Distributed Algorithms for Nonlinear Tree-Sparse Problems

Von der Fakultät für Mathematik und Physik
der Gottfried Wilhelm Leibniz Universität Hannover

zur Erlangung des Grades
Doktor der Naturwissenschaften
Dr. rer. nat.

genehmigte Dissertation
von

Dipl.-Math. Jens Hübner

geboren am 23.10.1984 in Bremen

2016

Referent:	Prof. Dr. Marc Steinbach, Gottfried Wilhelm Leibniz Universität Hannover
Korreferent:	Prof. Dr. Rüdiger Schultz, Universität Duisburg-Essen
Tag der Promotion:	15. Januar 2016

Abstract

Motivated by huge-scale portfolio problems and by controlling dynamic processes with stochastic disturbances, this thesis considers specific formulations of tree-structured problems that are attributed to the class of multistage problems in stochastic optimization. These tree-sparse problems are large but structured nonlinear optimization problems that are solved in a generic primal-dual interior-point framework employing a filter line-search globalization. Common approaches in this algorithmic framework are tailored to the specific problem structures. The arising KKT systems are solved by a direct structure-exploiting method performed as recursions over the tree. Dealing with rank-deficiencies and nonconvexities in nonlinear optimization, a problem-tailored inertia correction heuristic is developed that is incorporated into the KKT algorithm to avoid expensive refactorizations of the KKT matrix. In a structured quasi-Newton approach, second-order derivatives are generated based on partially separable Lagrangians. Numerical results are presented showing that the quasi-Newton approach combined with inertia corrections can be used to regulate dynamic processes modeled by perturbed ordinary differential equations and, additionally, is also a competitive alternative to exact second-order evaluations.

Moreover, facing the computational demands of huge-scale problems, this thesis presents a complete concept of distribution for the tree-sparse problems and the interior-point framework. The solution procedure is completely distributed based on a static depth-first distribution of the tree nodes. Theoretical results of the depth-first distributed trees are presented and used to develop distributed versions of the tree-sparse algorithms with few communication overhead. Parallel performance results for huge-scale portfolio optimization problems are presented proving the practicability of the concept of distribution and showing the efficiency of this approach.

Keywords: stochastic optimization, nonlinear optimization, tree-sparse problems, interior-point methods, distributed programming

Kurzzusammenfassung

Motiviert durch großzahlige Portfoliooptimierungsprobleme und durch Steuerungen von dynamischen Prozessen mit stochastischen Störungen werden in dieser Arbeit spezielle Formulierungen von baumstrukturierten Problemen betrachtet, die der Klasse der mehrstufigen stochastischen Optimierungsprobleme zuzuordnen sind. Diese großen aber dünnbesetzten Baumprobleme werden mit einer generischen primal-dualen Innere-Punkte-Methode gelöst, die eine Filter-Line-Search-Strategie zur Globalisierung einsetzt. Allgemeine Ansätze in diesem algorithmischen Rahmen werden auf die speziellen Problemstrukturen zugeschnitten. Die auftretenden KKT-Systeme werden mit einem direkten strukturausnutzenden Verfahren gelöst, welches durch Rekursionen über den Baum realisiert wird. Zur Behandlung von Rangdefekten und Nichtkonvexitäten wird eine Heuristik zur Signaturkorrektur in den KKT-Algorithmus eingebaut, die erneute Faktorisierungen der KKT-Matrix vermeidet. In einem strukturierten Quasi-Newton-Ansatz werden zweite Ableitungen auf Basis von partiell separierbaren Lagrange-Funktionen erzeugt. Die numerischen Ergebnisse zeigen, dass der Quasi-Newton-Ansatz kombiniert mit der Signaturkorrektur zum einen eingesetzt werden kann, um dynamische Prozesse zu regulieren, die durch gestörte gewöhnliche Differentialgleichungen modelliert werden, zum anderen aber auch eine konkurrenzfähige Alternative zur Verwendung von exakten zweiten Ableitungen sein kann.

Diese Arbeit präsentiert zudem ein vollständiges Konzept zur Verteilung der Baumprobleme und zum verteilten Rechnen der Innere-Punkte-Methode. Die Verteilung des Lösungsprozesses basiert auf einer statischen Aufteilung der Baumknoten nach Tiefensuche. Theoretische Ergebnisse zu den resultierenden tiefensuchenverteilten Bäumen werden bewiesen und genutzt, um parallele Varianten der problemspezifischen Algorithmen zu entwickeln, die nur einen geringen Kommunikationsaufwand nach sich ziehen. Anhand von großzahligen Portfoliooptimierungsproblemen werden Rechenergebnisse zur parallelen Performanz präsentiert, die die Praktikabilität des Verteilungskonzepts bestätigen und die Effizienz dieses Ansatzes aufzeigen.

Schlagworte: Stochastische Optimierung, nichtlineare Optimierung, baumstrukturierte Probleme, Innere-Punkte-Methoden, verteilte Programmierung

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Stochastic Optimization and Tree-Sparse Problems

Numerical optimization is an area of research in applied mathematics with the goal of providing solution software that helps finding decisions in real-world applications. In a first step of developing a solution software, the application needs to be translated into a mathematical optimization model that simplifies the complexity of the real world and highlights only the relevant key characteristics of the application. Typically seeking for solutions in infinite dimensional spaces, the mathematical model then runs through several discretization procedures to obtain a computationally tractable optimization problem. Finally, this optimization problem needs to be solved by applying a suitable solution algorithm.

This thesis focuses on the development and implementation of solution approaches for very specific types of optimization problems that arise in controlling time-continuous dynamic processes such as managing a portfolio or regulating the cell production of a bioreactor. Decisions in these applications need to be made on a basis of incomplete information, meaning that some influencing factors such as the development of the stock prices or the specific reaction rate of the concentration in the tank are not fully known at the time of a decision. Incorporating these uncertainties into the mathematical model of an application attributes the optimization problems to the subfield of stochastic optimization. Discretizing the process in time as well as explicit modeling of the uncertainties by considering finite many possible events lead to a scenario tree that represents the development of the dynamic process. The resulting discretized problems are challenging large-scale nonlinear optimizations problems that may include computationally expensive evaluations of solutions of ordinary differential equations. However, the problems are very structured featuring problem data that reflect the topology of the scenario tree. Exploiting the inherent tree structures is mandatory for solving these tree-sparse problems efficiently in a reasonable amount of time.

1.2. Distributed Programming

In scientific computing such as numerical optimization, the solution software typically requires a lot of processing power and memory resources. Since mainly composed of arithmetic operations, a computer program is computationally intensive. Moreover, finer discretization used in the solution procedure leads to larger problem instances and, therefore, more data to be stored. Now, since the beginning of the 21st Century the development of computers has changed drastically from increasing the processing power and memory capacities of a single machine to using distributed computing systems comprising several machines. Hence, facing high computational demands and memory requirements in scientific computing means making use of those distributed platforms.

Simply starting a computer program on a distributed platform is not sufficient to exploit the available computational resources. Prior to this, the solution software needs to be arranged for distributed computing, beginning with a suitable concept that leads to good parallel performance while being scalable with respect to the problem size. Aspects of distributed programming such as dividing the problem data and the computational workload, allocating the respective parts among the participating working units as well as invoking communication between these units for data transmission not only become part of the chores of implementation but also play a key role in the design of the solution approach. In this thesis, the tree-structured problems and the presented problem-tailored algorithms are stated in a consistent node-wise presentation, and a concept of distribution is developed that is based on a static distribution of the tree nodes.

1.3. Contributions and Organization

Interior-point methods emerged in the late 1980s and quickly became one of the most popular algorithm classes for optimization problems for two reasons. First, they are applicable to a lot of problem classes providing a consistent algorithmic framework for the different classes. Second, they benefit greatly from exploiting problem-specific structures, which makes them advantageous for large-scale optimization problems and, thus, well-suitable for the considered tree-structured problems. This thesis establishes the class of the *nonlinear tree-sparse problems* and develops a problem-tailored interior-point approach including a structured quasi-Newton framework for generating second-order derivatives. Moreover, this thesis provides a complete *concept of distribution* allowing to solve these problems on distributed platforms.

After stating the fundamental theory of nonlinear optimization, Chapter 2 introduces the basic concepts of interior-point methods and outlines algorithmic extensions for nonlinear problems.

Next, Chapter 3 motivates the tree-sparse problems by the means of stochastic optimization and reviews the tree-sparse convex problems established by Steinbach. Generalizing these convex problems, Chapter 4 states the nonlinear tree-sparse problems studied in this thesis and establishes notation and theoretical foundation for them first. After presenting the complete recursive direct KKT algorithms for the convex case (Sect. 4.3), these algorithms are then extended to deal with rank-deficiencies and nonconvexities of nonlinear problems. For this, a problem-tailored inertia correction is developed and directly incorporated into the KKT solution procedure (Sect. 4.4). Dealing with problems that do not provide evaluations of second-order derivatives, Hessian approximations are generated in a structured quasi-Newton approach based on partially separable functions (Sect. 4.5).

Maintaining a node-wise presentation for problems and algorithms, the concept of distribution for the tree-sparse problems presented in Chap. 5 is based on a static depth-first distribution of the tree nodes. After stating the distributed programming model for the tree-sparse problems (Sect. 5.1), the concept of depth-first distributed trees is established and theoretical results of these types of trees are presented (Sect. 5.2). Based on these results, distributed versions of the tree-sparse algorithms with few communication overhead are developed (Sect. 5.3) and, finally, the distribution of the complete interior-point algorithm is discussed (Sect. 5.4).

The tree-sparse algorithms and the concept of distribution are implemented in two C++ libraries that are developed to solve the tree-sparse problems using a generic interior-point framework. Chapter 6 discusses the software design of these libraries and outlines their incorporation into the interior-point solver. In Chap. 7, numerical experiments with examples from robust model predictive control and financial engineering demonstrate the modeling possibilities of the tree-sparse formulations and the potentials of the tree-sparse algorithms. The structured quasi-Newton approach combined with inertia corrections is applied to control dynamic processes with stochastic disturbances (Sect. 7.1). Parallel performance results are presented for huge-scale problems in portfolio optimization (Sect. 7.2). Finally, Chapter 8 concludes this thesis and considers some directions for future work.

Chapter 2

Basic Concepts of Nonlinear Optimization

This chapter presents the fundamental concepts of nonlinear optimization that is studied in this work. Section 2.1 introduces smooth nonlinear optimization problems and states necessary and sufficient conditions that characterize their solutions. The subsequent section focuses on algorithmic approaches for computing these solutions.

Notation: Vectors $v \in \mathbb{R}^n$ are always column vectors, the gradient ∇f of a sufficiently smooth function $f : \mathbb{R}^n \times \mathbb{R}$ is a column vector and the Jacobian $\nabla c \in \mathbb{R}^{m \times n}$ of a sufficiently smooth function $c : \mathbb{R}^n \to \mathbb{R}^m$ comprises the transposes of the gradients ∇c_i.

2.1. Theory Of Nonlinear Optimization

Consider the continuous nonlinear optimization problem (NLP)

$$\min_{x \in \mathbb{R}^n} \quad f(x) \tag{2.1a}$$

$$\text{s.t.} \quad c_i(x) = 0, \ i \in \mathcal{E}, \tag{2.1b}$$

$$c_i(x) \geq 0, \ i \in \mathcal{I}, \tag{2.1c}$$

where the real-valued functions $f, c_i : \mathbb{R}^n \to \mathbb{R}$ are sufficiently smooth. The function f is called the *objective*. The index sets $\mathcal{E}$ for the *equality constraints* and $\mathcal{I}$ for the *inequality constraints* are finite and disjoint. With $|\mathcal{E}| = m$ and $|\mathcal{I}| = k$, the constraint functions (2.1b) and (2.1c) read

$$c_{\mathcal{E}} : \mathbb{R}^n \to \mathbb{R}^m \quad \text{with} \quad c_{\mathcal{E}}(x) = 0 \quad \text{as well as} \quad c_{\mathcal{I}} : \mathbb{R}^n \to \mathbb{R}^k \quad \text{with} \quad c_{\mathcal{I}}(x) \geq 0, \tag{2.2}$$

where the relations $=$ and $\geq$ are meant component-wise.

NLP (2.1) is called a *linear problem* (LP) if the objective and all constraint functions are linear. A *quadratic problem* (QP) has linear constraints and an objective of the form

$$f(x) = \frac{1}{2}x^T Hx + c^T x \quad \text{with} \quad H = H^T \in \mathbb{R}^{n\times n} \quad \text{and} \quad c \in \mathbb{R}^n. \tag{2.3}$$

A point x satisfying the constraints (2.2) is called *feasible* for the optimization problem (2.1). The feasible points form the *feasible set*

$$\mathcal{F} := \{x \in \mathbb{R}^n \colon c_{\mathcal{I}}(x) = 0 \text{ and } c_{\mathcal{E}}(x) \geq 0\}. \tag{2.4}$$

The following definition characterizes solutions of NLPs.

Definition 1 (Solution)**.** *A feasible point $x^* \in \mathcal{F}$ is a* local solution *or simply a* solution *of problem* (2.1) *if there exists a neighborhood $\mathcal{N}$ of x^* such that $f(x) \geq f(x^*)$ for all $x \in \mathcal{N} \cap \mathcal{F}$. A solution is called a* global solution *if the relation $f(x) \geq f(x^*)$ holds for all feasible points $x \in \mathcal{F}$.*

Optimality conditions for solutions of problem (2.1) can be stated by means of the so-called Lagrangian.

Definition 2 (Lagrangian Function)**.** *The function*

$$\mathcal{L}(x, z, v) := f(x) - \sum_{i\in\mathcal{E}} z_i c_i(x) - \sum_{i\in\mathcal{I}} v_i c_i(x) \tag{2.5}$$

is called the Lagrangian *to optimization problem* (2.1)*. The scalars $z_i, v_i \in \mathbb{R}$ are the so-called* Lagrange multipliers *or* dual variables *corresponding to the equality and inequality constraints, respectively.*

Combining the Lagrange multipliers into vectors $z = (z_i)_{i\in\mathcal{E}}$ and $v = (v_i)_{i\in\mathcal{I}}$ for the equality and inequality constraints, respectively, the Lagrangian (2.5) reads

$$\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^k \to \mathbb{R} \quad \text{with} \quad \mathcal{L}(x, z, v) = f(x) - z^T c_{\mathcal{E}}(x) - v^T c_{\mathcal{I}}(x). \tag{2.6}$$

The concept of active and inactive constraints is important for the theoretical concepts as well as for the algorithmic treatments of NLPs. It is subsumed into the definition of the active set.

Definition 3 (Active Set)**.** *The* active set $\mathcal{A}(x) \subseteq \mathcal{E} \cup \mathcal{I}$ *of a feasible point* x *contains all constraint indices* i *with* $c_i(x) = 0$*, i.e.*

$$\mathcal{A}(x) := \mathcal{E} \cup \{i \in \mathcal{I} \colon c_i(x) = 0\}. \tag{2.7}$$

A constraint $i \in \mathcal{E} \cup \mathcal{I}$ *is said to be* active *at a feasible point* x *if* $i \in \mathcal{A}(x)$ *and* inactive *if* $i \notin \mathcal{A}(x)$*.*

The subsequent presentation of optimality conditions require the definition of the LICQ.

Definition 4 (Linear Independence Constraint Qualification)**.** *At a feasible point* x*, the* linear independence constraint qualification (LICQ) *is said to hold if the set* $\{\nabla c_i(x) \colon i \in \mathcal{A}(x)\}$ *of active constraint gradients is linearly independent.*

The following theorems express necessary first-order and second-order optimality conditions for a solution of NLPs by means of the Lagrangian and Lagrange multipliers.

Theorem 1 (First-Order Necessary Conditions)**.** *Let* x^* *be a solution of problem* (2.1) *and the functions* f *and* c_i *be continuously differentiable. Suppose that the LICQ holds at* x^**. Then there exist Lagrange multipliers* z^* *and* v^* *such that the following conditions are satisfied:*

$$\nabla_x \mathcal{L}(x^*, z^*, v^*) = \nabla f(x^*) - \sum_{i \in \mathcal{E}} z_i^* \nabla c_i(x^*) - \sum_{i \in \mathcal{A} \cap \mathcal{I}} v_i^* \nabla c_i(x^*) = 0, \tag{2.8a}$$

$$-\nabla_z \mathcal{L}(x^*, z^*, v^*) = c_{\mathcal{E}}(x^*) = 0, \tag{2.8b}$$

$$-\nabla_v \mathcal{L}(x^*, z^*, v^*) = c_{\mathcal{I}}(x^*) \geq 0, \tag{2.8c}$$

$$v^* \geq 0, \tag{2.8d}$$

$$v_i \cdot c_i(x^*) = 0, \quad i \in \mathcal{I}. \tag{2.8e}$$

x^* *is then said to be a* stationary point *of problem* (2.1)*.*

The conditions (2.8) are known as *KKT conditions* (for *Karush*, *Kuhn* and *Tucker*) and a stationary point is also called a *KKT point*. Condition (2.8a) is often called *dual feasibility*, conditions (2.8b) and (2.8c) form the *primal feasibility*, (2.8d) will be addressed as *nonnegativity* condition (for the Lagrange multipliers of the inequality constraints), and (2.8e) is called *complementarity* condition.

Theorem 2 (Second-Order Necessary Conditions)**.** *Let x^* be a solution of problem* (2.1) *with corresponding Lagrange multipliers z^* and v^*, and let the functions f and c_i be twice continuously differentiable. Suppose that the LICQ holds at x^*. Then,*

$$p^T \nabla^2_{xx} \mathcal{L}(x^*, z^*, v^*) p \geq 0 \tag{2.9}$$

holds for all vectors p with

$$\begin{aligned} & \nabla c_i(x^*)^T p = 0, \quad i \in \mathcal{E}, \\ \text{or} \quad & \nabla c_i(x^*)^T p = 0, \quad i \in \mathcal{A}(x^*) \cap \mathcal{I} \text{ with } v_i^* > 0, \\ \text{or} \quad & \nabla c_i(x^*)^T p \geq 0, \quad i \in \mathcal{A}(x^*) \cap \mathcal{I} \text{ with } v_i^* = 0. \end{aligned} \tag{2.10}$$

For directions that satisfy one of the equations (2.10), first-order derivatives do not give enough information to determine whether the objective will increase or decrease. Theorem 2 states that at a KKT point (x^*, z^*, v^*), the Hessian of the Lagrangian must have nonnegative curvature along those directions. Condition (2.9) is even sufficient if strictly satisfied, which is stated by the next theorem.

Theorem 3 (Second-Order Sufficient Conditions)**.** *Let (x^*, z^*, v^*) satisfy the KKT conditions* (2.8) *and suppose that*

$$p^T \nabla^2_{xx} \mathcal{L}(x^*, z^*, v^*) p > 0 \tag{2.11}$$

holds for all vectors p that satisfy (2.10)*. x^* is then a local solution for problem* (2.1)*.*

The following convex problems play an important role in nonlinear optimization.

Definition 5 (Convex Problem)**.** *Let the set $M \subseteq \mathbb{R}^n$ be connected. M is a* convex set *if for all $x, y \in M$ it holds*

$$\tau x + (1 - \tau) y \in M \quad \text{for all} \quad \tau \in [0, 1]. \tag{2.12}$$

A real-valued function $g : M \to \mathbb{R}$ is called a convex function *if for all $x, y \in M$ it is*

$$g(\tau x + (1 - \tau) y) \leq \tau g(x) + (1 - \tau) g(y) \quad \text{for all} \quad \tau \in [0, 1]. \tag{2.13}$$

The optimization problem (2.1) *is said to be a* convex problem *if the objective function f in* (2.1a) *and the feasible set $\mathcal{F}$ defined in* (2.4) *are convex.*

The feasible set is convex if the equality functions $c_{\mathcal{E}}$ are (linear-)affine and the functions $-c_{\mathcal{I}}$ are convex. For convex problems it can be shown that each KKT point (x^*, z^*, v^*) satisfies the sufficient conditions of Thm. 3. Hence, the first-order necessary conditions (2.8) are already sufficient. Furthermore, any local solution of a convex problem is also a global solution.

A detailed discussion of optimality conditions for NLPs along with proofs for theorems 1 to 3 can be found in the textbook [66]. A comprehensive background on convex optimization is given, e.g., by Boyd and Vandenberghe [16].

2.2. Solution Algorithms for Nonlinear Problems

The most popular solution techniques for NLPs include *sequential quadratic programming*[1] (SQP) and *interior-point methods* (IPM). Both are iterative approaches based on the Newton's method that aim for a KKT point, meaning a primal-dual vector (x^*, z^*, v^*) that satisfies the KKT conditions (2.8). If necessary, so-called *globalization strategies* are incorporated to enforce convergence towards a local solution.

Considering optimization problems without inequality constraints, i.e. NLP (2.1) with $|\mathcal{I}| = 0$ and thus $\mathcal{I} = \emptyset$, the KKT conditions (2.8) reduce to a system of nonlinear equations that can be solved using the Newton's method. Inequality constraints complicate system (2.8) by adding further primal feasibility conditions without reducing the dimensions of the feasibility set $\mathcal{F}$ (2.4) and by causing the nondifferentiable complementarity conditions (2.8e). The phenomenon of having $2^{|\mathcal{I}|}$ possible combinations of active or inactive constraints is often referred to as the *combinatorial difficulty* of NLPs [66].

In each iteration of an SQP method, the original NLP is approximated by a quadratic model based on the Lagrangian (2.5). So-called *active-set strategies* are often used to solve these local quadratic approximations. They address the combinatorial difficulty by systematically guessing the active set $\mathcal{A}(x^*)$, where x^* is the solution of the local quadratic model. Usually, they solve a series of equality-constrained quadratic subproblems that include active constraints $i \in \mathcal{W}$, with the *working set* $\mathcal{W}$ being the current guess for $\mathcal{A}(x^*)$. For more details on SQP methods and active-set strategies the interested reader is referred to the textbook [66] and the references therein.

[1] For historical reasons, numerical optimization is often referred to as programming. In this thesis, a program is always a computer program and programming refers to designing such a program.

IPMs use so-called *barrier problems* to approximate the NLP (2.1) in each iteration. Such a barrier problem reads

$$\min_{x,s} \quad f(x) - \mu \sum_{i \in \mathcal{I}} \ln(s_i) \tag{2.14a}$$

$$\text{s.t.} \qquad c_{\mathcal{E}}(x) = 0, \tag{2.14b}$$

$$c_{\mathcal{I}}(x) - s = 0. \tag{2.14c}$$

An IPM solves a series of problems (2.14) while driving the so-called *barrier parameter* μ to zero. Rather than guessing the active-set, an IPM deals with the combinatorial difficulty by *relaxing* the complementarity conditions (2.8e).

IPMs are in the algorithmic focus of this thesis. Section 2.2.1 introduces the basic concepts of IPMs by means of the so-called *homotopy approach*, which is well-known for being equivalent to the previously described *barrier approach* [66]. Algorithmic extensions for nonconvex NLPs are considered in Sect. 2.2.2. Section 2.2.3 outlines the idea of quasi-Newton methods in nonlinear optimization.

2.2.1. Interior-Point Methods

In the homotopy approach, the basic concept of IPMs is established as follows. By introducing nonnegative slack variables $s \in \mathbb{R}^k$ for the inequality constraints (2.1c), problem (2.1) is reformulated as

$$\min_{x,s} \qquad f(x) \tag{2.15a}$$

$$\text{s.t.} \qquad c_{\mathcal{E}}(x) = 0, \tag{2.15b}$$

$$c_{\mathcal{I}}(x) - s = 0, \tag{2.15c}$$

$$s \geq 0. \tag{2.15d}$$

The Lagrangian to this slacked NLP version reads

$$\mathcal{L}(x, s, z, v, \xi) = f(x) - z^T c_{\mathcal{E}}(x) - v^T (c_{\mathcal{I}}(x) - s) - \xi^T s. \tag{2.16}$$

Using the dual feasibility conditions

$$\nabla_s \mathcal{L} = v - \xi = 0 \tag{2.17}$$

Algorithm 1: Basic Interior-Point Method

1 choose an initial point $(x, s, z, v)^{(0)}$ with $s^{(0)} > 0$ and $v^{(0)} > 0$;
2 select $\mu^{(0)} > 0$, $\tau \in (0, 1)$, $\varepsilon > 0$;
3 set $k \leftarrow 0$;
4 **while** $E(x^{(k)}, s^{(k)}, z^{(k)}, v^{(k)}; \mu^{(k)}) > \varepsilon$ **do**
5 solve (2.21) to obtain the search direction $\Delta(x, s, z, v)^{(k)}$;
6 evaluate $\alpha_{\text{prim}}^{\max,(k)}$ and $\alpha_{\text{dual}}^{\max,(k)}$ from (2.23);
7 set $(x, s, z, v)^{(k+1)}$ using (2.24);
8 choose $\mu^{(k+1)}$;
9 set $k \leftarrow k + 1$;

to eliminate the so-called *dual slacks* ξ, the KKT conditions for (2.15) are written as the nonlinear equations

$$\nabla f(x) - \nabla c_{\mathcal{E}}(x)^T z - \nabla c_{\mathcal{I}}(x)^T v = 0, \tag{2.18a}$$

$$SVe - \mu e = 0, \tag{2.18b}$$

$$c_{\mathcal{E}}(x) = 0, \tag{2.18c}$$

$$c_{\mathcal{I}}(x) - s = 0, \tag{2.18d}$$

with $\mu = 0$, together with the nonnegativity conditions

$$s \geq 0 \quad \text{and} \quad v \geq 0. \tag{2.19}$$

The capital letters in (2.18b) denote diagonal matrices consisting of the entries of the corresponding vectors, i.e. $S = \text{Diag}(s)$ and $V = \text{Diag}(v)$, and $e = (1, \ldots, 1)^T$ is the vector of all ones of appropriate dimension. Note that by multiplying (2.18b) with S^{-1}, conditions (2.18) and (2.19) coincide with the KKT conditions of the barrier problem (2.14).

Interior-point methods are iterative algorithms that solve a series of the perturbed systems (2.18) while driving the barrier parameter μ to zero. The Newton's method is applied to (2.18) for a fix μ and a damping strategy is used to remain strictly feasible with respect to the nonnegativity conditions (2.19). A basic IPM scheme is outlined in Alg. 1. Starting with a strictly feasibly initial point $(x, s, z, v)^{(0)}$ with respect to (2.19), iteration steps 4 to 7 are repeated until the *KKT error* measured by

$$E(x, s, z, v; \mu) = \max\{\|\nabla_x \mathcal{L}(x, s, z, v)\|, \|SVe - \mu e\|, \|c_{\mathcal{E}}(x)\|, \|c_{\mathcal{I}}(x)\|\} \tag{2.20}$$

is reduced to a given error tolerance $\varepsilon > 0$. In each iteration, the Newton's method applied

to (2.18) leads to the so-called *primal-dual system* that is transformed into the symmetric form

$$\begin{bmatrix} \nabla^2_{xx}\mathcal{L} & 0 & \nabla c_{\mathcal{E}}(x)^T & \nabla c_{\mathcal{I}}(x)^T \\ 0 & S^{-1}V & 0 & -I \\ \nabla c_{\mathcal{E}}(x) & 0 & 0 & 0 \\ \nabla c_{\mathcal{I}}(x) & -I & 0 & 0 \end{bmatrix} \begin{pmatrix} \Delta x \\ \Delta s \\ -\Delta z \\ -\Delta v \end{pmatrix} = - \begin{pmatrix} \nabla_x \mathcal{L} \\ Ve - \mu S^{-1} e \\ c_{\mathcal{E}}(x) \\ c_{\mathcal{I}}(x) - s \end{pmatrix}. \tag{2.21}$$

Solving (2.21) for $\Delta(x, s, z, v)$ provides the direction for the next iterate. If applying the full step violates the positivity constraints

$$s > 0 \quad \text{and} \quad v > 0, \tag{2.22}$$

the step is shortened so the next iterate remains feasible with respect to (2.22). The step length is controlled by the so-called *fraction-to-the-boundary rule*,

$$\alpha^{\max}_{\text{prim}} = \max\left\{\alpha \in (0,1] \colon s + \alpha \Delta s \geq (\tau - 1)s\right\}, \tag{2.23a}$$

$$\alpha^{\max}_{\text{dual}} = \max\left\{\alpha \in (0,1] \colon v + \alpha \Delta v \geq (\tau - 1)v\right\}, \tag{2.23b}$$

where $\tau \in (0,1)$ is the *fraction-to-the-boundary parameter*. The new iterate is given by

$$(x, s)^{(k+1)} = (x, s)^{(k)} + \alpha^{\max}_{\text{prim}} \Delta(x, s)^{(k)}, \tag{2.24a}$$

$$(z, v)^{(k+1)} = (z, v)^{(k)} + \alpha^{\max}_{\text{dual}} \Delta(z, v)^{(k)}. \tag{2.24b}$$

In each iteration, the barrier parameter $\mu^{(k)}$ is updated or stays the same. The applied *barrier update strategy* has to ensure that the sequence of barrier parameters converges to zero, i.e. $\mu^{(k)} \to 0$.

Comprehensive background on IPMs are given, e.g., in the textbooks [66, 107]. Theoretical convergence results can be found in these textbooks and, e.g., in [26, 92, 104]. For state-of-the-art implementation techniques the reader is referred, e.g., to [72, 103, 105].

2.2.2. Algorithmic Extensions for Nonconvex Problems

When dealing with nonconvex optimization problems (see Def. 5), a globalization strategy is incorporated into Alg. 1 to enforce the convergence towards a local solution. The problems in this thesis are solved using an IPM that employs a filter line-search approach as globalization strategy, which is outlined next.

Finding a solution for NLP (2.15) aims for the two goals of minimizing the objective f (2.15a)

and satisfying the constraints (2.15b) to (2.15d). Measuring the feasibility of a primal point (x, s) by a real-valued function

$$h(x,s) = \sum_{i\in\mathcal{E}} \|c_i(x)\| + \sum_{i\in\mathcal{I}} \|c_i(x) - s\|, \tag{2.25}$$

filter methods treat these two goals separately. A filter comprises pairs $(f(x^{(l)}), h(x^{(l)}, s^{(l)}))$ for some iterates $(x, s, z, v)^{(l)}$. An iterate $(x, s, z, v)^{(k+1)}$ leads to a new element in the filter if $(x, s)^{(k+1)}$ progresses in f or in h. The following definition of a filter is taken from [66].

Definition 6 (Filter and Acceptance)**.** *Let $f_k = f(x^{(k)})$ denote the value of the objective* (2.15a) *at the iterate $(x, s, z, v)^{(k)}$ and $h_k = h(x^{(k)}, s^{(k)})$ its value of the infeasibility function* (2.25)*.*

1. *A pair (f_k, h_k) is said to* dominate *another pair (f_l, h_l) if both $f_k \leq f_l$ and $h_k \leq h_l$.*
2. *A* filter *is a list of pairs (f_l, h_l) such that no pair dominates any other.*
3. *An iterate $(x, s, z, v)^{(k)}$ is said to be* acceptable *to the filter if (f_k, h_k) is not dominated by any pair in the filter.*

The iterate (2.24) is neglected if the corresponding pair (f_{k+1}, h_{k+1}) is dominated by another pair in the filter. In that case, a *line search* is performed along the primal direction $\Delta(x, s)^{(k)}$ searching for a step length $\alpha_{\text{prim}}^{(k)} \in (0, \alpha_{\text{prim}}^{\max})$ such that the resulting new iterate

$$(x,s)^{(k+1)} = (x,s)^{(k)} + \alpha_{\text{prim}}^{(k)} \Delta(x,s)^{(k)} \tag{2.26}$$

is accepted by the filter. Such a line-search is only worthwhile if the search direction $\Delta(x, s)^{(k)}$ is useful for the optimization algorithm. The concept of such a *descent direction*, which is motivated from solving convex problems, is presented next. The presentation requires the following definition of the inertia for symmetric matrices.

Definition 7 (Inertia)**.** *The* inertia *of a symmetric matrix $M \in \mathbb{R}^{N\times N}$ comprises its numbers of positive, negative and zero eigenvalues n^+, n^- and n^0, respectively:*

$$\text{inertia}(M) = (n^+, n^-, n^0) \quad \textit{with} \quad n^+ + n^- + n^0 = N. \tag{2.27}$$

Eliminating Δs in (2.21) reduces the primal-dual system to a KKT system of the form $\Omega\Delta = \omega$, where the *KKT matrix* reads

$$\Omega = \begin{bmatrix} \nabla^2_{xx}\mathcal{L} & \nabla c_{\mathcal{E}}^T & \nabla c_{\mathcal{I}}^T \\ \nabla c_{\mathcal{E}} & 0 & 0 \\ \nabla c_{\mathcal{I}} & 0 & -\Sigma^{-1} \end{bmatrix} \quad \text{with} \quad \Sigma = S^{-1}V. \tag{2.28}$$

Eliminating the dual direction Δv, the *reduced KKT matrix* of the resulting system has the form

$$\Omega_{\mathrm{r}} = \begin{bmatrix} \hat{H} & \nabla c_{\mathcal{E}}^T \\ \nabla c_{\mathcal{E}} & 0 \end{bmatrix} \quad \text{with} \quad \hat{H} = \nabla^2_{xx}\mathcal{L} + \nabla c_{\mathcal{I}}^T \Sigma \nabla c_{\mathcal{I}}. \tag{2.29}$$

For convex problems the reduced matrix (2.29) is known to be regular with n positive and m negative eigenvalues [66].

The inertia of Ω_{r} in the convex case motivates the following definition of a descent direction for nonconvex optimization.

Definition 8 (Descent Direction)**.** *A step $\Delta(x, s, z, v)$ obtained from system* (2.21) *is a* descent direction *for problem* (2.15) *(with respective to a filter) if the reduced KKT matrix* (2.28) *satisfies*

$$\text{inertia}(\Omega_r) = (n, m, 0). \tag{2.30}$$

The subsequent regularity assumptions are sufficient to obtain a descent direction from the primal-dual system (2.21).

Assumption 1 (Regularity Assumptions for the Reduced KKT Matrix)**.** *Let $\Omega_r^{(k)}$ be the reduced KKT matrix* (2.29) *for the iterate $(x, s, z, v)^{(k)}$ with $\hat{H}_k := \hat{H}(x^{(k)}, z^{(k)}, v^{(k)})$ and $A_k := \nabla c_{\mathcal{E}}(x^{(k)})$. The following conditions apply:*

(A1) A_k has full row rank.

(A2) $\hat{H}_k$ is positive definite on the null-space of A_k, i.e. $\hat{H}_k|_{\mathcal{N}(A_k)} > 0$.

When the reduced KKT matrix (2.29) does not satisfy As. 1, the search direction obtained from (2.21) may not be a descent direction. In that case, one can ensure to get a descent direction by replacing the reduced KKT matrix (2.29) in the step computation by a corrected version

$$\Omega_{\mathrm{r}}^{\mathrm{corr}} = \begin{bmatrix} \hat{H} + \gamma_c I & \nabla c_{\mathcal{E}}^T \\ \nabla c_{\mathcal{E}} & -\gamma_r I \end{bmatrix}. \tag{2.31}$$

Any *regularization parameter* $\gamma_r > 0$ leads to full row rank of the lower row block in $\Omega_{\mathrm{r}}^{\mathrm{corr}}$. Choosing the *convexification parameter* $\gamma_c > 0$ sufficiently large ensures $\hat{H} + \gamma_c I$ to be positive definite (on the null-space of $\nabla c_{\mathcal{E}}$). Applying such an *inertia correction strategy* implies computing the step (2.21) with an *approximation* of the primal-dual matrix rather than using the exact data.

More details on globalization strategies can be found in [66] and the references therein. Filter line-search approaches for nonlinear optimization are proposed by Wächter and Biegler [104]. Inertia corrections are discussed, e.g., in [72, 103, 105].

2.2.3. Quasi-Newton Methods in Nonlinear Optimization

Quasi-Newton methods refer to iterative approaches that solve nonlinear systems with the Newton's method using *approximations* of derivatives instead of *exact* derivatives in the linearized subsystems.

In unconstrained optimization, i.e. in solving nonlinear problems (2.1) with $\mathcal{E} \cup \mathcal{I} = \emptyset$, the nonlinear system under consideration is the first-order necessary condition $\nabla f(x) = 0$. The resulting Newton's system reads $\nabla^2 f(x)\Delta x = -\nabla f(x)$. Approximations $\mathfrak{B} \approx \nabla^2 f$ of the Hessian of the objective are computed by using update heuristics that include first-order derivative information of the current iterate.

Given an initial point $x^{(0)}$ and an initial approximation $\mathfrak{B}^{(0)}$, the subsequent approximations are obtained by using, for example, one of the symmetric update formulae

$$\text{(PSB)} \qquad \mathfrak{B}^{(k+1)} = \mathfrak{B}^{(k)} + \frac{\mathfrak{r}^{(k)}(\mathfrak{s}^{(k)})^T + \mathfrak{s}^{(k)}(\mathfrak{r}^{(k)})^T}{(\mathfrak{s}^{(k)})^T\mathfrak{s}^{(k)}} - \frac{\left((\mathfrak{r}^{(k)})^T\mathfrak{s}^{(k)}\right)\mathfrak{s}^{(k)}(\mathfrak{s}^{(k)})^T}{\left((\mathfrak{s}^{(k)})^T\mathfrak{s}^{(k)}\right)^2}, \tag{2.32a}$$

$$\text{(BFGS)} \qquad \mathfrak{B}^{(k+1)} = \mathfrak{B}^{(k)} - \frac{\mathfrak{B}^{(k)}\mathfrak{s}^{(k)}(\mathfrak{s}^{(k)})^T\mathfrak{B}^{(k)}}{(\mathfrak{s}^{(k)})^T\mathfrak{B}^{(k)}\mathfrak{s}^{(k)}} + \frac{\mathfrak{g}^{(k)}(\mathfrak{g}^{(k)})^T}{(\mathfrak{g}^{(k)})^T\mathfrak{s}^{(k)}}, \tag{2.32b}$$

$$\text{(SR1)} \qquad \mathfrak{B}^{(k+1)} = \mathfrak{B}^{(k)} + \frac{\mathfrak{r}^{(k)}(\mathfrak{r}^{(k)})^T}{(\mathfrak{r}^{(k)})^T\mathfrak{s}^{(k)}}, \tag{2.32c}$$

where the iteration data are given by

$$\mathfrak{s}^{(k)} := x^{(k+1)} - x^{(k)}, \quad \mathfrak{g}^{(k)} := \nabla f(x^{(k+1)}) - \nabla f(x^{(k)}), \quad \mathfrak{r}^{(k)} := \mathfrak{g}^{(k)} - \mathfrak{B}^{(k)}\mathfrak{s}^{(k)}. \tag{2.33}$$

Each of the formulae (2.32) estimates the curvature of f along the direction from $x^{(k)}$ to $x^{(k+1)}$ and updates the approximation with the new curvature information. The *Powell-symmetric-Broyden* formula (2.32a) and the *Broyden-Fletcher-Goldfarb-Shanno* formula (2.32b) are rank-two updates that—under certain conditions—guarantee the approximations $\mathfrak{B}^{(k)}$ to be positive definite. The *symmetric-rank-one* update formula (2.32c) uses only one dyadic product

for updating the Hessian approximation. Approximations using SR1 updates tend to be more reliable for nonconvex problems since they do not try to avoid indefinite $\mathfrak{B}^{(k)}$ that reflect the original curvature of $\nabla^2 f(x^{(k)}) \not\succ 0$ more appropriately than enforcing $\mathfrak{B}^{(k)} > 0$ [66].

In constrained optimization, i.e. in solving NLPs (2.1) with $\mathcal{E} \cup \mathcal{I} \neq \emptyset$, the nonlinear systems under consideration are the KKT conditions (2.8). Applying the Newton's method to (2.8)—as it is done in an IPM algorithm (cf. Sect. 2.2.1)—leads to linear systems involving blocks of the KKT matrix (2.28). Hessian update strategies are then used to approximate the Hessian of the *Lagrangian* (2.6) . The approximation $\mathfrak{B}^{(k+1)} \approx \nabla^2_{xx}\mathcal{L}(x^{(k+1)}, z^{(k+1)}, v^{(k+1)})$ is obtained by updating $\mathfrak{B}^{(k)}$ with an estimation of the curvature of the Lagrangian along the direction $\mathfrak{s}^{(k)} = x^{(k+1)} - x^{(k)}$. Recalling from Sect. 2.1 the gradient $\nabla_x \mathcal{L}$ of the Lagrangian (2.6),

$$\nabla_x \mathcal{L}(x, z, v) = \nabla f(x) - \nabla c_{\mathcal{E}}(x)^T z - \nabla c_{\mathcal{I}}(x)^T v, \tag{2.34}$$

and using the notation

$$\mathcal{L}^+(x^{(k+1)}) := \mathcal{L}(x^{(k+1)}, z^{(k+1)}, v^{(k+1)}) \quad \text{and} \tag{2.35a}$$

$$\mathcal{L}^-(x^{(k)}) := \mathcal{L}(x^{(k)}, z^{(k+1)}, v^{(k+1)}), \tag{2.35b}$$

quasi-Newton methods in constrained optimization use formulae like (2.32) where $\mathfrak{g}^{(k)}$ reads

$$\mathfrak{g}^{(k)} := \nabla_x \mathcal{L}^+(x^{(k+1)}) - \nabla_x \mathcal{L}^-(x^{(k)}), \tag{2.36}$$

and $\mathfrak{s}^{(k)}$ and $\mathfrak{r}^{(k)}$ are defined as before in (2.33).

Considerations for the choice of an update formula depend on the applied globalization strategy incorporated into the optimization algorithm. Using filter line-search approaches in IPMs, the desire to obtain a descent direction (see Def. 8) from (2.21) motivates to use rank-two update formulae like BFGS or PSB. With some adjustments for the constrained optimization case such as damping strategies, these formulae generate positive definite approximations $\mathfrak{B} \approx \nabla^2_{xx} f$ leading to the desired inertia (2.30) of the reduced KKT matrix (2.29).

An overview of quasi-Newton methods for unconstrained and constrained optimization including convergence analysis can be found in [66]. Standard references for quasi-Newton methods include [51, 21, 25]. A damping strategy for the BFGS update formula in constrained optimization was originally proposed by Powell [69]. Implementations of constrained optimization algorithms employing quasi-Newton approaches include KNITRO [17, 71] and SNOPT [32, 30].

Chapter 3

Applications of Tree-Sparse Problems

This thesis deals with nonlinear problems (NLPs) that arise in the optimization of dynamic processes with stochastic disturbances. Explicit modeling of uncertainties represented by a scenario tree leads to problem formulations that are attributed to the class of multistage problems in stochastic optimization. The resulting *nonlinear tree-sparse problems* (TSPs) are specific NLPs featuring problem data with characteristic structures that arise from the underlying tree topology. Exploiting the structures is the key to solving the TSPs efficiently. The formulations of the TSPs and their problem-tailored solution algorithms in Chap. 4 are based on convex counterparts, the so-called *tree-sparse convex problems* (TSCP) established by Steinbach. This chapter reviews the previous research in this field of tree-sparse optimization and motivates its expansion to the scope of nonconvex problems.

After introducing the multistage stochastic problems in Sect. 3.1, the convex case of tree-sparse optimization is reviewed in Sect. 3.2 and compared to related approaches developed by other research groups. Finally, Section 3.3 motivates the nonlinear TSPs by outlining the usage of nonlinear multistage stochastic problems in the context of model predictive control.

3.1. Stochastic Optimization

Problems in stochastic optimization are characterized by the presence of uncertain problem data that depend on a random variable ξ. The classical two-stage stochastic problem (SP) contains a first-stage problem that models the optimal choice of so-called first-stage decisions based on the expected value of a series of second-stage problems. Each second-stage subproblem corresponds to a realization of the random variable and models the optimal choice of the respective second-stage decisions with respect to the specific realization. Extending the two-stage approach to more than one observation of ξ in time leads to the multistage stochastic problems (MSPs). Solution approaches for both SPs and MSPs include NLP solvers that are

applied to the deterministic equivalent problems (DEPs). The DEP is either equivalent to a stochastic problem or it is an approximation resulting from a discrete approximation of a continuous random variable.

The following overview first introduces the classical two-stage problems (Sect. 3.1.1) and then extends the approach to the multistage case (Sect. 3.1.2), which is exemplified by the formulation of a portfolio selection problem (Sect. 3.1.3). Finally, solution approaches for stochastic problems with the focus on interior-point methods are discussed (Sect. 3.1.4).

Notation: In this overview, the capital letter T is used for the length of the planning horizon in the future forecast of stochastic problems. Avoiding notational conflicts, the transpose of a vector v will be denoted by v^*.

3.1.1. Two-Stage Stochastic Problems

The classical linear *two-stage stochastic problem* consists of the *first-stage problem*

$$\min_x \quad c^*x + \mathbb{E}_\xi[Q(x,\xi)] \tag{3.1a}$$

$$\text{s.t.} \quad Ax = b, \tag{3.1b}$$

$$x \geq 0, \tag{3.1c}$$

and a set of *second-stage subproblems*

$$Q(x,\xi) := \min_y \quad q(\xi)^* y(\xi) \tag{3.2a}$$

$$\text{s.t.} \quad T(\xi)x + W(\xi)y(\xi) = h(\xi), \tag{3.2b}$$

$$y(\xi) \geq 0. \tag{3.2c}$$

So-called *first-stage decisions* $x \in \mathbb{R}^{n_1}$ of problem (3.1) are made before the realization of the random variable ξ. The problem data $c \in \mathbb{R}^{n_1}$, $A \in \mathbb{R}^{m_1 \times n_1}$ and $b \in \mathbb{R}^{m_1}$ are *deterministic*, i.e. they are independent of ξ. With $\mathbb{E}_\xi[Q(x,\xi)]$ in (3.2a), an optimal choice of x takes the expected optimal value of the second-stage subproblems (3.2) into account. *Second-stage decisions* $y(\xi) \in \mathbb{R}^{n_2}$ as well as the problem data $q(\xi) \in \mathbb{R}^{n_2}$, $T(\xi) \in \mathbb{R}^{m_2 \times n_1}$, $W(\xi) \in \mathbb{R}^{m_2 \times n_2}$ and $h(\xi) \in \mathbb{R}^{m_2}$ are unknown and dependent on the random variable ξ. The second-stage constraint matrix $W(\xi)$ is called *recourse matrix* and $T(\xi)$ is referred to as *technology matrix*. A stochastic problem (3.1)–(3.2) is said to have *fixed recourse* if the matrix W is independent of the random variable, i.e. if $W(\xi) \equiv W$. The second-stage constraints (3.2b) and (3.2c) are supposed to

hold *almost surely*, i.e. for all events except for those with zero probability. Second-stage decisions $y(\xi)$ are determined after the observation of ξ and can be seen as corrections of the first-stage decisions x taking into account the additional knowledge from the outcome of the random experiments.

A compact presentation of problem (3.1)–(3.2) reads

$$\min_{x} \quad c^*x + \mathbb{E}_\xi[Q(x,\xi)] \tag{3.3a}$$

$$\text{s.t.} \quad Ax = b, \tag{3.3b}$$

$$T(\xi)x + W(\xi)y(\xi) = h(\xi), \tag{3.3c}$$

$$x, y(\xi) \geq 0. \tag{3.3d}$$

Although called linear, problem (3.3) is actually a nonlinear and possibly nonsmooth optimization problem. The terminology *linear* is justified by the so-called *deterministic equivalent problem*. In the presence of a continuous random variable, problem (3.3) is computationally intractable. The continuous random variable ξ needs to be approximated using a discrete representation $\bar{\xi}$ comprising finite many realizations $(\bar{\xi}_1, \ldots, \bar{\xi}_K)$ with $K \in \mathbb{N}$. With $y_k = y(\bar{\xi}_k)$ denoting the second-stage decisions y corresponding to the realization $\bar{\xi}_k$ and analogously using the second-stage data q_k, T_k, W_k and h_k, the DEP reads

$$\min_{x,y} \quad c^*x + \sum_{k \in K} p_k q_k^* y_k \tag{3.4a}$$

$$\text{s.t.} \quad Ax = b, \tag{3.4b}$$

$$T_k x + W_k y_k = h_k, \quad k = 1, \ldots K, \tag{3.4c}$$

$$x, y_1, \ldots, y_K \geq 0. \tag{3.4d}$$

The DEP (3.4) is an approximation of the linear stochastic problem (3.3). With its linear objective (3.4a) and its (linear-)affine constraints (3.4b) to (3.4d), the DEP is a standard LP. For more details on the classical two-stage problem the interested reader is referred to the textbook of Birge and Louveaux [8] and the references therein.

3.1.2. Multistage Stochastic Problems

Multistage models of stochastic problems are generalizations of the two-stage model described in Sect. 3.1.1. Several observations of the random experiment are made at different points in time. With each new observation the previous decisions are corrected. The following presentation follows the lines of Grothey [41].

In the multistage case, the uncertainty is described by a *stochastic process*, i.e. a sequence of random variables $\xi = (\xi_1, \dots, \xi_T)$. Subsequently, $\xi^t = (\xi_1, \dots, \xi_t)$ denotes the information available at time t. The decisions $x = (x_1, \dots, x_T)$ also form a stochastic process and are supposed to be *nonanticipative*, meaning a decision $x_t = x_t(\xi^t)$ only depends on past observations and not on future outcomes.

The linear *multistage stochastic problem* in compact form reads

$$\min_x \quad c_1^* x_1 + \mathbb{E}_{\xi^2}\left[Q_2(x_1, \xi^2) + \dots + \mathbb{E}_{\xi^T}[Q_T(x_{T-1}, \xi^T)]\dots\right] \tag{3.5a}$$

$$\text{s.t.} \quad T_{t-1}(\xi_t)x_{t-1}(\xi^{t-1}) + W_t(\xi_t)x_t(\xi^t) = h_t(\xi_t), \qquad t = 1, \dots, T, \tag{3.5b}$$

$$x_t(\xi^t) \geq 0, \qquad t = 1, \dots, T, \tag{3.5c}$$

where the functions $Q_t(x_{t-1}, \xi^t)$ in the objective (3.5a) represent optimal values of the *t-stage subproblems*

$$Q_t(x_{t-1}, \xi^t) := \min_{x_t} \quad c_t(\xi_t)^* x_t(\xi^t) \tag{3.6a}$$

$$\text{s.t.} \quad T_{t-1}(\xi_t)x_{t-1}(\xi^{t-1}) + W_t(\xi_t)x_t(\xi^t) = h(\xi_t), \tag{3.6b}$$

$$x_t(\xi^t) \geq 0. \tag{3.6c}$$

The first observation is deterministic, i.e. ξ_1 is a discrete random variable with only one realization. The decisions $x_1 \in \mathbb{R}^{n_1}$ correspond to the first-stage decisions in the two-stage case. The problem data $c_1 \in \mathbb{R}^{n_1}$, $W_1 \in \mathbb{R}^{m_1 \times n_1}$, $h_1 \in \mathbb{R}^{m_1}$ are deterministic and correspond to the first-stage data c, A, b in (3.1), respectively. For the sake of the compact problem representation (3.5), the undefined values x_0 and T_0 are formally set to $x_0 = \emptyset$ and $T_0 = \emptyset$. The defined uncertain data $c_t \in \mathbb{R}^{n_t}$, $T_{t-1} \in \mathbb{R}^{m_t \times n_{t-1}}$, $W_t \in \mathbb{R}^{m_t \times n_t}$ and $h_t \in \mathbb{R}^{m_t}$ depend on the realization of the current observation ξ_t. They can be chosen in different ways for each point of time t as indicated by the time subscript attached to the respective data.

A stochastic process ξ of discrete random variables ξ_t is represented by a *scenario tree* as shown in Fig. 3.1, where V is the numbered set of nodes of the tree. Each node $j \in V$ corresponds to a series of realizations $\bar{\xi}^t = (\bar{\xi}_1, \dots, \bar{\xi}_t)$ up to time $t = t(j)$, which coincides with the tree level of the node j. The predecessor of node j is denoted by $\pi(j)$ and its set of successors by $S(j)$. The root $0 \in V$ represents the current time $t(0) = 0$ and each leaf l in the set of leaves $L = \{l \in V \colon S(l) = \emptyset\}$ defines a *scenario*, i.e. a possible state $(\bar{\xi}_1, \dots, \bar{\xi}_T)$ of the process at the end of the *planning horizon* $t(j) = T$. A successor $k \in S(j)$ represents a possible future realization of the random variable ξ_t with an associated *transition probability* $\bar{p}_k$. All transition probabilities from a node to its successors sum up to one, i.e. $\sum_{k \in S(j)} \bar{p}_k = 1$. The

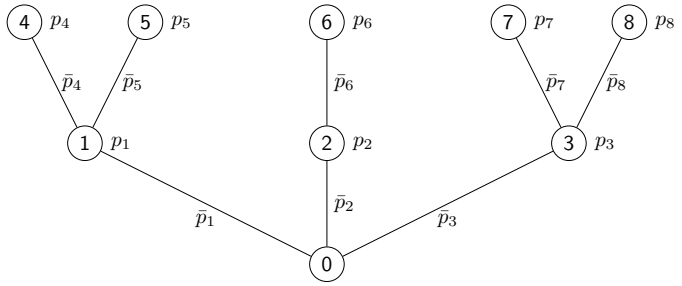


Figure 3.1.: Scenario tree with planning horizon $T = 2$ and $|V| = 9$ nodes

probability $p_j = \prod_{k \in \Pi(k)} \bar{p}_k$ associated with node j is the product of transition probabilities $\bar{p}_k$ on its path $\Pi(j)$ to the root 0.

Let the realizations corresponding to $j \in V$ be given by $\bar{\xi}_j = \bar{\xi}^{t(j)} = (\bar{\xi}_{1,j}, \ldots, \bar{\xi}_{t(j),j})$. The decision variables are denoted by $x_j = x(\bar{\xi}_j)$. The remaining problem data are mapped as follows: $c_j = c_t(\bar{\xi}_{t,j})$, $T_j = T_{t-1}(\bar{\xi}_{t,j})$, $W_j = W_t(\bar{\xi}_{t,j})$, $h_j = h_t(\bar{\xi}_{t,j})$. The deterministic version of the MSP then reads

$$\min_x \quad \sum_{j \in V} p_j c_j^* x_j \tag{3.7a}$$

$$\text{s.t.} \quad T_j x_{\pi(j)} + W_j x_j = h_j, \quad j \in V, \tag{3.7b}$$

$$x_j \geq 0, \qquad j \in V. \tag{3.7c}$$

In the case of discrete random variables, the DEP (3.7) is equivalent to the linear MSP (3.5). If the stochastic process is described by continuous random variables, each random variable ξ_t needs to be approximated by a discrete one. The resulting DEP is then an *approximation* of the original problem (3.5).

The extension of the linear MSP (3.7) to nonlinear multistage stochastic problems is straightforward. The objective of a quadratic MSP reads

$$\min_x \quad \sum_{j \in V} p_j \left(\frac{1}{2} x_j^* H_j x_j + c_j^* x_j \right) \tag{3.8}$$

with positive semidefinite matrices H_j. A simple nonlinear MSP formulation reads

$$\min_x \quad \sum_{j \in V} p_j \phi_j(x_j) \tag{3.9a}$$

$$\text{s.t.} \quad T_j(x_{\pi(j)}) + W_j(x_j) = 0, \quad j \in V, \tag{3.9b}$$

$$x_j \geq 0, \qquad j \in V. \tag{3.9c}$$

The objective function (3.9a) and the recourse (3.9b) are so-called *second-order decoupled* or *node-separable*, which reads

$$\nabla^2_{x_k x_l} \phi_j(x_j) = 0 \quad \text{and} \quad \nabla^2_{x_k x_l} \left(T_j(x_{\pi(j)}) + W_j(x_j) \right) = 0 \quad \text{for} \quad k \neq l. \tag{3.10}$$

3.1.3. A Portfolio Selection Problem

The following example is the multistage extension of the mean-variance approach for portfolio selection problems as introduced by Frauendorfer [27]. The presentation here follows the lines of Steinbach [78].

Consider a portfolio of n risky assets with the initial investment at $t = 0$. The portfolio is restructured at discrete times $t = 1, \ldots, T$, and redeemed one period later at time $T + 1$. The capitals of the assets at time t are gathered into the vector $x_t \in \mathbb{R}^n$. The capital already includes the transactions $v_t \in \mathbb{R}^n$ that are made in the time stage $[t, t+1)$. The initial wealth is normalized and fully invested, i.e. $e^* x_0 = 1$. New investments and partial redemptions within the period are not included. In the absence of transaction costs, the transaction conditions read $e^* v_t = 0$ for $t > 0$. The goal in the presented model is to attain a prescribed expected return $\rho > 1$ with minimal risk.

The development of the portfolio is described by a stochastic process $\xi = (\xi_0, \ldots, \xi_{T+1})$. The vector of returns in time stage $[t-1, t]$ is denoted by $r_t(\xi^t) \in \mathbb{R}^n$. Let $\bar{r}_T(\xi^T)$ be the expected return in the time stage $[T, T+1]$ conditioned on the history ξ^T, i.e. $\bar{r}_T(\xi^T) = \mathbb{E}(r_{T+1}(\xi^{T+1}|\xi^T))$. The covariance matrix associated with $\bar{r}_T(\xi^T)$ is given by $\Sigma_T(\xi^T) \in \mathbb{R}^{n \times n}$. At times $t > 0$, the portfolio is rebalanced after the observation of ξ_t, leading to the nonanticipative policy $x_t = r_t(\xi^t) x_{t-1} + v_t$. The risk is modeled as the variance of the expected return at the end of the period $T + 1$, which reads (omitting all stochastic dependencies) as follows:

$$\sigma^2_{\xi^{T+1}} \left(r^*_{T+1} x_T \right) = \mathbb{E} \left(x_T^* \left[\Sigma_T + \bar{r}_T^* \bar{r}_T \right] x_T \right) - \rho^2. \tag{3.11}$$

Assuming the returns r_j for $j \in V$ are given together with the expected returns $\bar{r}_k$ for $k \in L$ and the associated covariance matrices Σ_k, the DEP of the portfolio selection problem is a

convex QP and reads

$$\min_{x} \quad \sum_{j \in L} \frac{p_j}{2} x_j^* \left(\Sigma_j + \bar{r}_j \bar{r}_j^* \right) x_j - \rho^2 \tag{3.12a}$$

$$\text{s.t.} \quad e^* x_0 = 1, \tag{3.12b}$$

$$e^* x_j = r_j^* x_{\pi(j)}, \qquad j \in V \setminus \{0\}, \tag{3.12c}$$

$$\sum_{j \in L} p_j \bar{r}_j^* x_j = \rho. \tag{3.12d}$$

Extensions to the portfolio selection problem including cash flow, transaction costs and composition constraints can be found in the review paper [82]. The reviewed mean-variance approach is based on Markowitz' ideas on portfolio selection [63] and the utility of wealth [64]. A list of formulations and extensions leading to nonlinear MSPs is given, e.g., by Gondzio and Grothey [34]. Standard references for portfolio selection (also referred to as asset liability management) include [74, 110, 108].

3.1.4. Interior-Point Methods for Stochastic Problems

Many solution approaches for problems in stochastic optimization are sophisticated algorithms designed for solving deterministic equivalent counterparts. When dealing with continuous random variables, these first need to be approximated using discrete variables. This discretization process is often referred to as *scenario generation* and is not in the focus of this thesis.

Both the DEP (3.4) of the two-stage problem (3.3) and the DEP (3.7) of the multistage problem (3.5) are standard LPs. However, when accounting for many realizations of the random variables, both problems become very large. The problem sizes of the MSPs even grow exponentially with each stage. Solution algorithms have to exploit problem-specific characteristics in the solution process to keep the DEPs computationally tractable.

L-shaped methods for two-stage problems, for example, consider the first-stage problem (3.1) as so-called master problem and solve it by systematically evaluating second-stage problems (3.2) for some realizations of the random variable ξ. The second-stage problems to be evaluated are determined based on *Benders decompositions* [7] or alternatively *Dantzig-Wolfe decompositions* [20]. Extending this solution approach to the multistage case, the resulting so-called *nested L-shaped methods* solve each t-stage subproblem (3.6) with the L-shaped method. Informally speaking, nested L-shaped methods traverse the nodes of the scenario tree (cf. Sect. 3.1.2), starting with the root $0 \in V$, and solve the subproblem (3.6) corresponding to the current node j in the traversal. The branch that starts at j is cut when detecting that investigating the corresponding subproblem leads to no further progress towards the solution of the overall

problem, i.e. to the optimal choice of the first-stage decisions. More on (nested) L-shaped methods can be found, e.g., in [8] and the references therein.

When applying interior-point methods (cf. Sect. 2.2.1) to the DEP of a stochastic problem, one usually deals with large-scale KKT systems that, however, are very sparse and feature structured system matrices. The so-called *L-shaped* structure of the two-stage problem, for example, is best seen when restating the presentation (3.3) as

$$
\begin{aligned}
\min_{x,y} \quad & c^* x + \sum_{k \in K} p_k q_k^* y_k && (3.13a)\\
\text{s.t.} \quad & Ax = b, && (3.13b)\\
& T_1 x + W_1 y_1 = h_1, && (3.13c)\\
& T_2 x + W_2 y_2 = h_2, &&\\
& \vdots \qquad \ddots \qquad \vdots &&\\
& T_K x + W_K y_K = h_K, &&\\
& x, y_1, \ldots, y_K \geq 0. && (3.13d)
\end{aligned}
$$

In KKT matrices that arise when dealing with MSPs (3.7), the problem-specific structure reflects the underlying scenario tree. The recourse matrices W_j form a block-diagonal and the technology matrices T_j are located on the secondary diagonal corresponding to the predecessor node $\pi(j)$. For the example tree in Fig. 3.1, the equality constraint block of the KKT matrix (cf. Chap. 2) has the block structure

$$
\nabla c_{\mathcal{E}} = \begin{bmatrix}
W_0 & & & & & & & & \\
T_1 & W_1 & & & & & & & \\
T_2 & & W_2 & & & & & & \\
T_3 & & & W_3 & & & & & \\
& T_4 & & & W_4 & & & & \\
& T_5 & & & & W_5 & & & \\
& & T_6 & & & & W_6 & & \\
& & & T_7 & & & & W_7 & \\
& & & T_8 & & & & & W_8
\end{bmatrix}. \tag{3.14}
$$

In other words, the matrix (3.14) features a stochastic version of a so-called *staircase structure* that occurs in problems of dynamic optimization.

For IPM approaches in stochastic optimization, exploiting the problem-specific KKT structures

is mandatory to compete with other solution algorithms. Comparing an IPM with a nested L-shaped method, for example, the latter considers only some subsystems of the KKT system while one iteration of the IPM requires the solution of the *entire* system. However, handling the problem-specific structures properly, the otherwise cubic complexity of direct KKT solution algorithms can be reduced to linear complexity [82]. Such structure-exploiting direct methods for tree-structured KKT systems take center stage in tree-sparse optimization and are further discussed in Sect. 3.2.3.

Interior-point methods for LPs, QPs and NLPs are based on the same concepts [66]. Thus, IPM approaches for linear MSPs (3.7) are easily extended to the quadratic MSP case with objective (3.8) and to the nonlinear problem (3.9). Moreover, since the Jacobian of the nonlinear recourse constraints (3.9b) maintains the structure (3.14) due to the node-separability (3.10), the same structure-exploiting KKT algorithm can be used in all three cases.

3.2. Tree-Sparse Convex Problems

The tree-sparse convex problems (TSCP) established by Steinbach emphasize the dynamic nature of multistage stochastic problems (MSPs). The TSCP in implicit form (Sect. 3.2.1) is an MSP with convex objective and additional linear constraints. Distinguishing between dependent state variables and free control variables leads to the TSCPs with explicit controls (Sect. 3.2.2), which can be seen as convex problems in optimal control that take different developments of the controlled process explicitly into account. Solving large-scale optimization problems using interior-point methods requires problem-specific treatment of the arising KKT systems. For the TSCPs, the key to solving those systems efficiently is a hierarchical sparsity exploitation in a direct approach that is performed recursively over the tree (Sect. 3.2.3). Closely related IPM approaches of other research groups feature similar recursive KKT algorithms while dealing with coarser MSP formulations (cf. Sect. 3.2.4). Finally, the TSCPs are suitable for modeling problems in several applications of finance engineering and robust optimal control (Sect. 3.2.5).

3.2.1. Implicit Tree-Sparse Problems

The *tree-sparse convex problem in implicit form* reads

$$\min_{y} \quad \sum_{j \in V} \phi_j(y_j) \tag{3.15a}$$

$$\text{s.t.} \quad G_j y_i - P_j y_j + h_j = 0, \quad j \in V, \tag{3.15b}$$

$$F_j^r y_j \in [r_{lj}, r_{uj}], \qquad j \in V, \tag{3.15c}$$

$$y_j \in [b_{lj}, b_{uj}], \qquad j \in V, \tag{3.15d}$$

$$\sum_{j \in V} F_j y_j + e_V = 0, \tag{3.15e}$$

where $i = \pi(j)$ denotes the predecessor of node $j \in V$. The problem comprises a convex quadratic objective (3.15a) and the following linear constraints: *dynamic constraints* (3.15b), *range constraints* (3.15c), *simple bounds* (3.15d), and *global equality constraints* (3.15e). For the root $0 \in V$ it is $G_0 = \emptyset$. Problem (3.15) is an MSP (3.7) with a nonlinear convex objective function (3.9a) and additional linear equality and inequality constraints. The convexity of the objective (3.15a) implies that the Hessians $\nabla^2_{y_j y_j} \phi_j$ are positive semidefinite.

The implicit TSCP (3.15) is a *node-separable* convex optimization problem of the form

$$\min_{y} \quad \phi(y) \tag{3.16a}$$

$$\text{s.t.} \quad Ay = a, \tag{3.16b}$$

$$By \in [r_l, r_u], \tag{3.16c}$$

$$y \in [b_l, b_u]. \tag{3.16d}$$

Node-separability in the convex case means that mixed second-order derivatives of the objective (3.16) vanish, i.e. $\nabla_{y_k y_l} \phi(y) = 0$ for $k \neq l$, and the node variables y_j are at most *linearly coupled*. More precisely, the following three types of coupling occur in convex tree-sparse problems:

1. *Global linear coupling:* The objective (3.15a) and the equality constraints (3.15e) couple *all* node variables y_j linearly.

2. *Path coupling:* The dynamic constraints (3.15b) couple the variables of a node $j \in V$ with those of its predecessor i.

3. *Decoupled constraints:* The remaining constraints are so-called *local constraints*. They

involve only *local variables*, i.e. those node variables corresponding to the same node j as the constraint.

3.2.2. Tree-Sparse Problems with Explicit Controls

Splitting the node variables y_j into dependent *state variables* x_j and free *control variables* u_j, a first version of a tree-sparse problem with these explicit controls reads

$$\min_{x,u} \quad \sum_{j \in V} \phi_j(x_j, u_j) \tag{3.17a}$$

$$\text{s.t.} \quad G_j x_i + E_j u_i - x_j + h_j = 0, \quad j \in V, \tag{3.17b}$$

$$F_j^r x_j + D_j^r u_j \in [r_{lj}, r_{uj}], \qquad j \in V, \tag{3.17c}$$

$$x_j \in [b_{lj}^x, b_{uj}^x], \qquad j \in V, \tag{3.17d}$$

$$u_j \in [b_{lj}^u, b_{uj}^u], \qquad j \in V, \tag{3.17e}$$

$$\sum_{j \in V} (F_j x_j + D_j u_j) + e_V = 0. \tag{3.17f}$$

The former bounds (3.15d) are refined into *simple state bounds* (3.17d) and *simple control bounds* (3.17e). The remaining constraints and the objective are detailed versions of their corresponding implicit counterparts in (3.15). The explicit problem formulation (3.17) is mapped into the implicit problem (3.15) by combining vectors into

$$y_j = \begin{pmatrix} x_j \\ u_j \end{pmatrix}, \quad b_{lj} = \begin{pmatrix} b_{lj}^x \\ b_{lj}^u \end{pmatrix}, \quad b_{uj} = \begin{pmatrix} b_{uj}^x \\ b_{uj}^u \end{pmatrix}, \tag{3.18}$$

and by mapping matrix node subblocks by

$$\bar{G}_j = \begin{bmatrix} G_j \\ E_j \end{bmatrix}, \quad \bar{P}_j = \begin{bmatrix} I \\ 0 \end{bmatrix}, \quad \bar{F}_j^r = \begin{bmatrix} F_j^r \\ D_j^r \end{bmatrix}, \quad \bar{F}_j = \begin{bmatrix} F_j \\ D_j \end{bmatrix}, \tag{3.19}$$

where the barred letters denote the node subblocks of the implicit form (3.15).

In the dynamic constraints (3.17b), the impact of the controls influences the state of the dynamic system in the next time period, i.e. the controls u_i affect the states x_j of the successors $j \in S(i)$ in the time interval $[t(i), t(j))$, where $t(v)$ denotes the point of time corresponding to $v \in V$. For a successor $j \in S(i)$ of $i \in V$, the states x_j depend on the controls u_i as well as the preceding states x_i, and the applied controls u_i are the same for all successors $S(i)$. In the context of tree-sparse optimization, this control form is referred to as *outgoing control*

and problem (3.17) is a *tree-sparse problem in outgoing control form* or short *outgoing control problem.*

The dynamic constraints (3.17b) with so-called *incoming controls* read

$$G_j x_i + E_j u_j - x_j + h_j = 0, \quad j \in V. \tag{3.20}$$

In (3.20), the controls affect the state of the system in the same time period, meaning the controls u_j are applied at the beginning of the time interval $[t(j), t(j)+1)$ and affect the current states x_j. Speaking in terms of stochastic optimization, outgoing controls u_i and incoming controls u_j both are decisions that are made for the time period $[t(j), t(j)+1)$. The difference is that outgoing controls are decisions based on the knowledge up to history $\bar{\xi}^{t(i)}$ while decisions in incoming control form are made after the realization of $\xi_{t(j)}$, i.e. they are based on the knowledge up to history $\bar{\xi}^{t(j)}$.

For algorithmic reasons, the TSCP formulation with incoming controls requires some adjustments in comparison to the outgoing TSCP (3.17). First, the objective function with incoming controls reads

$$\min_{u,x} \quad \phi(u,x) = \sum_{j \in V} \left(\phi_{ij}(x_i, u_j) + \phi_j(x_j) \right). \tag{3.21}$$

Second, the range constraints (3.17c) are replaced by

$$F^r_{ij} x_i + D^r_j u_j \in [r^u_{lj}, r^u_{uj}], \quad j \in V, \tag{3.22a}$$

$$F^r_j x_j \in [r^x_{lj}, r^x_{uj}], \quad j \in V, \tag{3.22b}$$

where (3.22a) are *mixed ranges* and (3.22b) are pure *local state ranges.* The mixed ranges (3.22a) bear the same path coupling as the dynamics (3.20). Now, the so-called *tree-sparse problem in incoming control form* or short *incoming control problem* consists of the objective (3.21), the dynamics (3.20), the ranges (3.22), the bounds (3.17d) and (3.17e) as well as the equality constraints (3.17f).

With $\nabla^2_{x_i,u_j} \phi(x,u) \neq 0$, the incoming objective (3.21) is no longer node-separable in the sense of (3.10): the controls u_j are coupled nonlinearly with the predecessor's state variables x_i by the node functions ϕ_{ij}. The following motivation justifies to grant ϕ the property of *node-separability* nonetheless. In the outgoing control case, nonlinear couplings only occur between state-defining node variables, i.e. the states x_i and the controls u_i define the states x_j and are coupled nonlinearly by the node function ϕ_j. This observation still holds in the incoming control case where x_j is defined by the previous states x_i and the local controls u_j.

Refinement of Equality Constraints

In the implicit TSCP (3.15), nondynamic equalities are modeled as the globally coupled constraints (3.15e). A more detailed problem presentation includes so-called *local equality constraints*, i.e. decoupled equality constraints

$$F_j^y y_j + e_j^y = 0, \quad j \in V. \tag{3.23}$$

For explicit controls, these local equality constraints are refined into pure state constraints, pure control constraints and mixed equalities including both node variables. The refinement of the equality constraints for the outgoing TSCP (3.17) reads

$$F_j^x x_j + e_j^x = 0, \quad j \in V, \tag{3.24a}$$
$$D_j^u u_j + e_j^u = 0, \quad j \in V, \tag{3.24b}$$
$$F_j^c x_j + D_j^c u_j + e_j^c = 0, \quad j \in V, \tag{3.24c}$$

with *local state equalities* (3.24a), *local control equalities* (3.24b) and *mixed equality constraints* (3.24c). The equalities (3.24) are mapped into (3.23) as follows

$$F_j^y = \begin{bmatrix} F_j^x & \\ & D_j^u \\ F_j^c & D_j^c \end{bmatrix} \quad \text{and} \quad e_j^y = \begin{pmatrix} e_j^x \\ e_j^u \\ e_j^c \end{pmatrix}. \tag{3.25}$$

In the incoming control case, the mixed equalities (3.24c) are replaced by path coupling equations of the form

$$F_{ij}^c x_i + D_j^c u_j + e_j^c = 0, \quad j \in V. \tag{3.26}$$

Relations Between the Tree-Sparse Formulations

In [83], Steinbach presents all three tree-sparse convex problems in their most detailed forms showing that under certain conditions the problems are transformed into each other as illustrated in Fig. 3.2. An outgoing TSCP is transformed into an implicit problem by combining the states x_j and the controls u_j into the node variables y_j. In the reverse process, the outgoing TSCP is obtained from an implicit TSCP by variable splitting. An outgoing TSCP is embedded in the incoming control formulation and, in the reverse process, the incoming TSCP is transformed into an outgoing problem by collecting the children. Furthermore, Steinbach proves that the

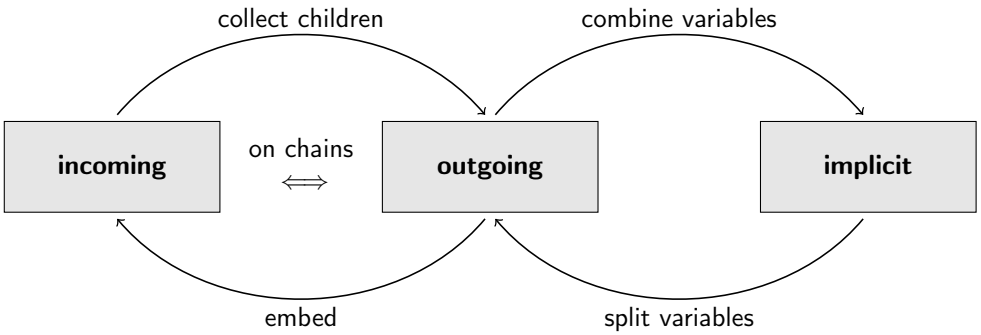


Figure 3.2.: Relations between the three control forms of the tree-sparse problems

explicit problem formulations are equivalent on chains as specific trees, i.e. in the deterministic case.

3.2.3. Hierarchical Sparsity and Recursive KKT Solution

Exploiting the sparsity in the arising KKT systems is mandatory when solving large-scale optimization problems. The KKT systems of tree-sparse problems benefit from four sources of sparsity that induce zero patterns on different levels in the KKT matrix. Hierarchically sorted from coarse to fine, these four levels of sparsity are as follows:

1. *Saddle point structure:* The KKT system (2.28) is a so-called *saddle point system* of the form

$$\begin{bmatrix} H & C^T \\ C & -M \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \quad \text{with} \quad C = \begin{bmatrix} A \\ B \end{bmatrix} \quad \text{and} \quad A = \begin{bmatrix} G \\ F^y \\ F \end{bmatrix}. \tag{3.27}$$

   The 2×2 system matrix is symmetric and indefinite, and the lower right block M usually is of diagonal form. The constraint matrix C comprises the inequality block B and the equality block A. The latter is further split into a dynamic block G, a block F^y corresponding to the local equality constraints and a global constraint block F.

2. *Tree structure:* As it is characteristic for multistage stochastic problems, the tree-sparse matrix blocks in the saddle point system (3.27) reflect the underlying tree topology (cf. Sect. 3.1.4). The matrix blocks have either block-diagonal form, the *stochastic staircase form* as in (3.14), or they are block-dense rows.

3. *Control refinement:* Compared with the implicit form, using explicit controls provides further structural information in the node subblocks, i.e. the matrix blocks with node

subscripts. The mappings (3.19) and (3.25) demonstrate the control refinement for the outgoing control case.

4. *Local sparsity:* The node subblocks may have problem-specific entry patterns that result from the specific optimization model at hand. For example, in a tree-sparse incoming control formulation of the portfolio selection problem (3.12), the node subblock G_j is of diagonal form and E_j is made of identity matrices, i.e. $G_j = \mathrm{diag}(r_j)$ and $E_j = [I, -I]$.

Neglecting the saddle point structure in optimization algorithms more than doubles the memory requirement for the KKT matrices as well as the computational effort for solving the KKT system. By definition , a solution algorithm designed for KKT systems takes this first sparsity level into account. In tree-sparse optimization, the second and the third sparsity level are addressed in a direct approach iterating recursively over the nodes of the underlying scenario tree. This approach is outlined subsequently. Finally, exploiting local sparsities requires solution algorithms that are tailored to the specific tree-sparse model at hand. Analyzing and exploiting local sparsities is discussed in [80] and [49]. An overview of the tree structures in matrix blocks and a detailed discussion on the control refinement can be found in [83]. Descriptions of the second sparsity level for explicit control problems without local equality constraints are also given in Sect. 4.2 of this thesis.

KKT Solution Algorithm for the Implicit Case

When solving convex problems of the form (3.16) with a primal-dual IPM (cf. Sect. 2.2.1), each IPM iteration requires one or several solutions of the KKT system

$$\begin{bmatrix} H+\Phi & A^T & B^T \\ A & & \\ B & & -\Psi^{-1} \end{bmatrix} \begin{pmatrix} \Delta y \\ -\Delta z \\ -\Delta v \end{pmatrix} = - \begin{pmatrix} \nu \\ a \\ \rho \end{pmatrix}, \tag{3.28}$$

where H denotes the Hessian of the Lagrangian corresponding to problem (3.16a), Φ and Ψ are positive diagonal matrices, z and v are the duals for the equalities (3.16b) and ranges (3.16c), respectively, and ν and ρ are concentrated right-hand side terms. Using the notation $\nu = (f_j)_{j\in V}$ and $\rho = (v_j)_{j\in V}$ as well as the Lagrange multipliers λ_j, μ_j^x and μ for the dynamics (3.15b), the local equalities (3.23) and the global equality constraints (3.15e), respectively, the KKT system (3.28) for the implicit TSP (3.15) with local equalities (3.23) can be stated as the

following set of equations:

$$
\begin{aligned}
&(H_j + \Phi_j)y_j + P_j^T \lambda_j - \sum_{k \in S(j)} G_k^T \lambda_k \\
&-(F_j^x)^T \mu_j^x - (F_j^r)^T v_j - F_j^T \mu + f_j = 0, \quad j \in V, && \text{(3.29a)} \\
&G_j y_i - P_j y_j + h_j = 0, \quad j \in V, && \text{(3.29b)} \\
&F_j^x y_j + e_j^x = 0, \quad j \in V, && \text{(3.29c)} \\
&F_j^r y_j + \Psi_j^{-1} v_j + r_j = 0, \quad j \in V, && \text{(3.29d)} \\
&\sum_{j \in V} F_j y_j + e_V = 0. && \text{(3.29e)}
\end{aligned}
$$

This KKT system is solved efficiently by applying the following three algorithm phases:

1. *Local projections:* Use the local equality constraints (3.29c) to eliminate as many primal variables y_j as possible. Project the remaining system, i.e. (3.29) without (3.29c), onto the null-space $\mathcal{N}(F_j^x)$ of the local equalities (3.29c).

2. *Elimination of range duals:* Solve (3.29d) for the range duals v_j and substitute this expression into the dual feasibility condition (3.29a).

3. *Recursive variable elimination:* Eliminate the remaining primal variables and the dynamic duals λ_j recursively over the tree.

After the first two steps, the tree-sparse system (3.29) is reduced to

$$
\begin{aligned}
&\bar{H}_j y_j^1 + \bar{P}_j^T \lambda_j - \sum_{k \in S(j)} \bar{G}_k^T \lambda_k - \bar{F}_j^T \mu + \bar{f}_j = 0, \quad j \in V, && \text{(3.30a)} \\
&\bar{G}_j y_i - \bar{P}_j y_j^1 + \bar{h}_j = 0, \quad j \in V, && \text{(3.30b)} \\
&\sum_{j \in V} \bar{F}_j y_j^1 + \bar{e}_V = 0, && \text{(3.30c)}
\end{aligned}
$$

where the barred subblocks are projected modifications of their original counterparts in (3.29) and y_j^1 are the remaining primal variables. In the third step, equations (3.30a) and (3.30b) are used in an inward recursion over the tree to eliminate y_j^1 and λ_j, respectively. Substituting the expressions of the eliminated variables into the global equation (3.30c) leads to a linear system that is solved for the global multiplier μ. The node variables are evaluated during an outward recursion over the tree using the expressions obtained during the elimination processes. The recursive elimination and evaluation procedure of the third algorithm step is referred to as *basic recursion*.

A detailed presentation of the implicit basic recursion for system (3.30) is given in [79], the local projections of (3.29c) can be found in [80].

Algorithmic Refinements for Explicit Control

Solution algorithms for tree-sparse KKT systems corresponding to problems in explicit control forms follow the same three phases as in the implicit case. The first algorithm phase for explicit controls include three local projections: projections for the local state equalities (3.24a), for the local control equalities (3.24b) as well as for the mixed equalities (3.24c) and (3.26), respectively. Local projections correpsonding to a node $j \in V$ may induce additional local constraints or dynamic constraints for its predecessor i. Hence, in the explicit case, the projection phase of the solution algorithm is also performed recursively over the tree.

Tree-sparse problems with outgoing controls are stochastic extensions of the optimal control problems considered in [77]. There, Steinbach provides a complete discussion of the KKT solution algorithm for outgoing controls on chains. The extension on trees of the second and the third phase of this algorithm is supplemented in this thesis (cf. Sect. 4.3.2). For the incoming control case, the basic recursion is presented in [79] and can also be found in Sect. 4.3.3. A complete discussion of the local projections are provided in [83]. Detailed presentations of the second phases of the tree-sparse KKT algorithms cannot be found in the literature. For both explicit control cases, those elimination phases are discussed in this work (cf. Sect. 4.3). Finally, regularity conditions for the KKT solution algorithms of all three problem formulations are discussed in detail in [83].

3.2.4. Related Approaches

In the literature, there are two structured IPM approaches for MSPs that are closely related to the one for tree-sparse problems and which feature similar recursive KKT solution algorithms. First, Gondzio and Grothey consider quadratic MSPs with global inequality constraints. They base their recursive approach on factorizing subblock prototypes that are obtained from reordering the structured KKT matrix [34, 35]. Their parallel C++-implementation OOPS [1] exploits this subblock presentation further for scheduling corresponding computations dynamically in the parallel computational environment. Gondzio and Grothey solve nonlinear extensions of the quadratic MSPs with a textbook SQP framework using OOPS as the underlying QP solver [33, 36].

Second, Blomvall and Lindberg consider convex MSPs with explicit controls, which basically are outgoing TSCPs without global and local equality constraints. They solve these problems using a *Ricatti-based IPM solver* [9, 11, 12, 13]. In [10], Blomvall proposes a parallel approach

for the structured KKT solution algorithm based on a depth-first distribution of the tree nodes that is also considered in [46, 50] in the context of tree-sparse optimization. Blomvall's idea also takes center stage in the distribution of the nonlinear tree-sparse problems in Chap. 5.

In the context of portfolio optimization, Grothey [41] provides a review of IPM approaches for MSPs (cf. Sect. 3.1.3) together with comparisons of the previously described three approaches including the tree-sparse optimization.

3.2.5. Applications and Software

In [77], Steinbach establishes solution algorithms for KKT systems that arise in nonlinear optimization approaches for optimal control problems (OCP). He solves the discretized OCPs in an SQP framework that employs an IPM approach for the occurring quadratic subproblems. The considered OCPs arise from controlling industrial robots [76, 87, 88, 86]. Steinbach compares his Fortran 77 implementations of the sophisticated KKT solution algorithms called MSKKT with the (at that time) state of the art sparse solvers MA28 [23] and LAPACK's BAND [3].

Motivated by applications in financial engineering, the TSCPs lead back to stochastic extensions of Steinbach's KKT solution algorithms. Tree-sparse formulations of the portfolio selection problem (3.12) are established in [78, 81]. In [89], Steinbach and Vollbrecht study the valuation of swing options at energy markets. They present a multistage stochastic model of the valuation problem formulated as an incoming control problem. Outgoing control formulations are used in [29] to model the separation of methanol and water in a binary mixture. Related articles that provide further modeling aspects and additional computational results for this distillation process include Henrion et al. [44] and Steinbach [84]. Implemented software for these applications is written in C++.

Research work with the focus on software implementation include the three diploma theses of Hutanu [49], Hofmann [46] and the author [50]. Hutanu addresses the potential of exploiting local sparsity (cf. Sect. 3.2.3) and develops a software tool that generates C++-code tailored to specific node subblock structures. With this code generator, Hutanu provides a fair compromise between the time-consuming writing of an efficient handcrafted source code and the usage of an inefficient implementation without local sparsity exploitation [49]. In [46] and [50], first attempts are made to parallelize the tree-sparse KKT solution algorithms. In this thesis, the distribution of the complete algorithm for (nonlinear) tree-sparse problems is subject of Chap. 5. Detailed information on the implementation together with discussions of previous software development in this context are provided in Chap. 6.

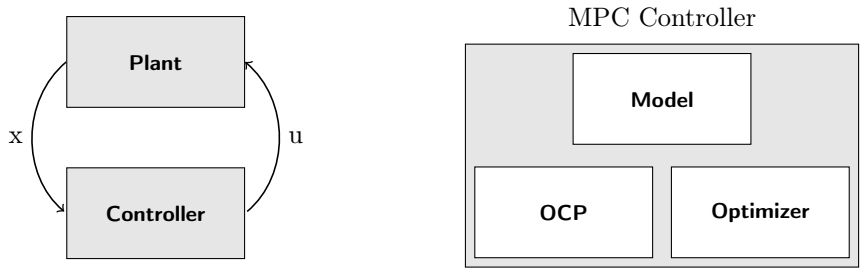


Figure 3.3.: Closed-loop control system (left) and MPC controller (right)

3.3. Robust Model Predictive Control

The following presentation outlines the basic ideas of *model predictive control* (MPC) and sketches a *robust* MPC approach that incorporates nonlinear MSPs in its procedure. For a comprehensive background on MPC the interested reader is referred to the textbook [43] and the references therein.

Researchers in the area of optimal control study dynamic processes and develop *controllers* that manipulate these processes in a desired way. The task of such a controller is to bring a real-world instance of the process called *plant* into a reference state and keep it there. In an *open-loop control system*, the controller forecasts the behavior of the plant over a certain *prediction horizon* and generates a control signal such that the dynamic process develops in an optimal way. However, these predictions are seldom exact or to say it with Moiraine's[1] words, "The Wheel (of time) weaves as the Wheel wills." [53]. In a so-called *closed-loop control system*, information about the state of the plant is constantly being fed to the controller that in turn is adjusting its control signal (see Fig. 3.3 on the left-hand side). This signal is then given in *feedback form* $u(t) = \mu(x(t))$, where the *feedback law* μ defined by the controller determines the control signal $u(t)$ based on the current state $x(t)$.

Now, model predictive control is an optimization-based approach for the feedback law μ [43]. An MPC controller comprises a dynamic *model* of the plant, an *optimal control problem* (OCP) and a solver for the OCP called *optimizer* (see Fig. 3.3 on the right-hand side). First, the dynamic model, which is here assumed to be given as a system of ordinary differential equations (ODEs), is a simplified version of the plant that predicts its change of state with respect to an applied control signal. Next, the OCP incorporates the dynamic model as constraints and defines an objective that penalizes deviations of the current state from the reference state as well as costs caused by the applied control. Finally, the optimizer fills the OCP with current data, solves it

[1] Moiraine Damodred, daughter of Dalresin Damodred of House Damodred in Cairhien, Aes Sedai of the Blue Ajah of the White Tower in Tar Valon

over the prediction horizon T and, this way, determines the control signal that is returned next to the plant. Thus, an MPC controller in a closed-loop system repeats the following steps:

1. Receive a measurement of the current state $x(t = t_0)$ of the plant.
2. Solve the OCP over the prediction horizon $t_0 + T$.
3. Define the feedback law $\mu(x(t)) = u(t_0)$ and apply this control in the next period.

In this scheme of a so-called *moving horizon controller*, the length of the prediction horizon T remains the same in each run. Thus, the prediction horizon is moving forward in time.

Generally, the MPC controller is error-prone with several sources of error. First, *model errors* arise due to the simplification of the real-world process that is considered in the OCP. Second, the state of the plant is observed by taking measurements of only a small number of characteristics. Additionally to the *measurement errors* caused by characteristics that are not observed, also, the taken measurements cannot be expected to be exact but rather feature some error margins. Finally, the scheme above does not account for the communication times of the exchanged signals, i.e. the measurements and the control signals. *Time delays* are, therefore, another possible source of errors.

By the continuing adjustment of the control signal, a closed-loop control system is already designed for taking errors into account. In *robust* MPC, however, errors are in addition incorporated explicitly into the dynamic model of the plant. Here, it is assumed that model errors are represented by uncertain parameters with possibly different realizations in time. Then the dynamic model of the plant reads

$$\dot{x}(t) = g(x(t), u(t), \xi(t)), \tag{3.31}$$

where $\xi(t) \equiv \xi$ is a continuous random variable.

Multistage Stochastic Problems in Robust MPC

The simplest version of a multistage stochastic problem used as OCP in the MPC controller reads

$$\min_{x,u} \quad \sum_{j \in V} p_j \phi_j(x_j, u_j) \tag{3.32a}$$

$$\text{s.t.} \quad x_j - g_j(x_i, u_i) = 0, \quad j \in V, \tag{3.32b}$$

where $i = \pi(j)$ is the predecessor of node $j \in V$ (cf. Sect. 3.2). The node functions ϕ_j are non-negative functions representing costs. The node functions g_j in the dynamic constraints (3.32b)

have the form

$$g_j(x_i, u_i) = x_i + \int_{t(i)}^{t(j)} g(x(t), u_i, \xi_j)dt, \tag{3.33}$$

where $t(j)$ denotes the point of time corresponding to node j. Thus, in (3.33) the trajectory x is described as solution of the initial value problem

$$\dot{x}(t) = g(x(t), u(t) \equiv u_i, \xi(t) \equiv \xi_j) \quad \text{with} \quad x(t(i)) = x_i, \tag{3.34}$$

and g_j represents the value of x at time $t = t(j)$.

The MSP (3.32) is obtained by approximating the uncertain parameter ξ by a stochastic process of discrete random variables and by applying a multiple shooting approach to the dynamic model (3.31). In more detail, the steps to formulation (3.32) are as follows:

- Discretize the time by using the time grid $\Gamma = \{[t_0, t_1], \ldots, [t_{m-1}, t_m = T]\}$.
- For each point of time t_k, approximate the continuous random variable ξ by a discrete one $\bar{\xi}_{t_k}$. The resulting stochastic process is represented by a scenario tree with tree nodes V and scenarios L (cf. Sect. 3.1.2).
- For each scenario $l \in L$, approximate the control function u by a piecewise constant function $\bar{u}^l$ with $\bar{u}^l(t(j)) = u_j$ for all ancestors $j \in \Pi(l)$ of l.
- For each scenario $l \in L$, approximate the trajectory x by a continuous and piecewise linear function $\bar{x}^l$ with $\bar{x}^l(t(j)) = x_j$ for all ancestors $j \in \Pi(l)$ of l.
- Include the states x_j explicitly as optimization variables into the optimization problem and ensure the continuity of all functions $\bar{x}^l$ by incorporating the dynamic constraints (3.32b).

Now, solving problem (3.32) leads to solution functions $\bar{u}^l$ and $\bar{x}^l$ such that the expected value of the costs (3.32a) is minimized. This way, the control signal $u(t_0) = u_0$ that is sent back to the plant is optimal with respect to all considered scenarios.

Chapter 4

Algorithms for Nonlinear Tree-Sparse Problems

The nonlinear tree-sparse problems (TSPs) studied in this work generalize the tree-sparse convex problems established by Steinbach (cf. Sect. 3.2). They are solved using a primal-dual interior-point method (IPM) that employs a filter line-search globalization (cf. Sect. 2.2). Common algorithmic approaches in this optimization framework are tailored to the specific presentation of the TSPs. The resulting tree-sparse algorithms feature the same node-wise presentation as the TSPs and, in doing so, preserve the sparsity pattern originating from the underlying tree topology. Difficulties arising for nonlinear problems as well as for problems that do not provide evaluations of second-order derivatives are also addressed in this node-wise manner.

This chapter is organized as follows. Section 4.1 introduces the TSPs that are studied in this work and highlights their key features that are exploited in the subsequent sections. Section 4.2 tailors the general discussions of NLPs in the context of IPMs to the specific case of the TSPs. The tree-sparse KKT algorithms for convex TSPs are presented in Sect. 4.3. By introducing a problem-specific inertia correction strategy, Section 4.4 extends these KKT algorithms for their application in solving nonlinear problems. TSPs that do not provide evaluations of second-order derivatives are addressed in Sect. 4.5. There, a structured quasi-Newton approach based on Hessian update strategies for partially separable functions is proposed. Finally, Section 4.6 presents further tree-sparse algorithms and discusses some numerical issues in the context of tree-sparse optimization.

Notation: Considering tree-sparse problems in the context of interior-point methods, notational conflicts are entailed by bringing two sets of notation together, i.e. the notation used for nonlinear optimization and IPMs (cf. Chap. 2) are mixed with the one used for the tree-sparse problems

(cf. Sect. 3.2). Remaining consistent with previous used notation and with the relevant notation in the referred literature, possible confusions are clarified at the beginning of each section.

4.1. Nonlinear Tree-Sparse Problems

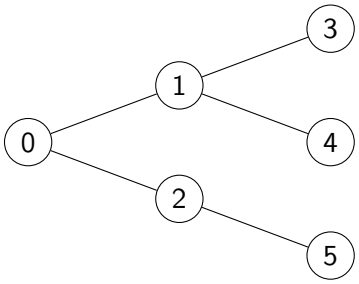
Nonlinear tree-sparse problems are NLPs that originate from optimizing dynamic processes with stochastic disturbances (cf. Sect. 3.1). They feature an underlying tree topology that results from a time discretization and an explicit modeling of the uncertainties. Each node in the tree represents an event with an associated probability. Distinguishing between free control variables and dependent state variables, the studied TSPs express their dynamic nature explicitly. TSPs in implicit form are not considered in this thesis (cf. Sect. 3.2.1).

In the following, Section 4.1.1 states the tree notation used to describe the TSPs and their algorithms. Section 4.1.2 introduces the studied tree-sparse problems, i.e. explicit TSPs in outgoing and in incoming control form, and maps them into a standard NLP formulation. Afterwards, Section 4.1.3 highlights the key characteristics of those TSPs.

Notation: The primal variables of an NLP are denoted by y and the objective function by ϕ whereas f refers to the global equality constraints of the TSPs. Fixed subscripts are l (lower) and u (upper) as well as the node subscripts i, j, k. Fixed superscripts are x and u denoting state-related and control-related dimensions and functions, respectively. Then, u_j are always the control node variables whereas u, u_l and u_u refer to Lagrange multipliers corresponding to simple bounds.

4.1.1. Tree Notation

The considered trees have numbered *node sets* V, are rooted at node $0 \in V$ and have the *tree depth* T, which originates from discretizing a time-continuous process. Each node $j \in V$ is associated with a *tree level* $t(j)$ indicating its distance to the root. Thus, it is $t(0) = 0$. The unique predecessor of a node j is given by $i = \pi(j)$, and for the root it is $\pi(0) = \emptyset$. A node j has a *set of successors* $S(j)$, and the *leaves* L are those nodes without successors, i.e. $S(j) = \emptyset$ for $j \in L$. The *level set* L_t comprises all nodes on level t whereas V_t forms the set of all nodes *up to level* t, i.e. $V_t = \bigcup_{\tau=0}^{t} L_\tau$. Figure 4.1 shows a simple example tree with its corresponding node sets and level sets.



Node sets	Level sets
$V_0 = \{0\}$	$L_0 = \{0\}$
$V_1 = \{0, 1, 2\}$	$L_1 = \{1, 2\}$
$V_2 = \{0, \ldots, 5\}$	$L_2 = \{3, 4, 5\}$
$V = \{0, \ldots, 5\}$	$L = \{3, 4, 5\}$

Figure 4.1.: Example tree with corresponding node sets and level sets

4.1.2. Nonlinear Tree-Sparse Problems with Explicit Controls

Two formulations of tree-sparse problems with explicit controls are studied in this thesis. Both TSPs are formulated as NLPs in the presentation

$$\min_{y \in \mathbb{R}^n} \quad \phi(y) \tag{4.1a}$$

$$\text{s.t.} \quad c_{\mathcal{E}}(y) = 0, \tag{4.1b}$$

$$c_{\mathcal{R}}(y) \in [r_l, r_u], \tag{4.1c}$$

$$y \in [b_l, b_u]. \tag{4.1d}$$

The former inequalities of problem (2.1) are subclassified into *range constraints* (4.1c) with $|\mathcal{R}| = k$ and $r_l, r_u \in \mathbb{R}^k$ as well as *bound constraints* or simply *bounds* (4.1d) with $b_l, b_u \in \mathbb{R}^n$. Problem (4.1) is equivalent to (2.1) with $|\mathcal{I}| = 2k + 2n$ using the mapping

$$c_{\mathcal{I}}(y) = \left(c_{\mathcal{R}}(y) - r_l, -c_{\mathcal{R}}(y) + r_u, y - b_l, -y + b_u\right)^T. \tag{4.2}$$

The *nonlinear tree-sparse problem in outgoing control form* studied in this thesis reads

$$\min_{x,u} \quad \sum_{j \in V} \phi_j(x_j, u_j) \tag{4.3a}$$

$$\text{s.t.} \quad g_j(x_i, u_i) - x_j = 0, \qquad j \in V, \tag{4.3b}$$

$$r_j(x_j, u_j) \in [r_{lj}, r_{uj}], \quad j \in V, \tag{4.3c}$$

$$x_j \in [b^x_{lj}, b^x_{uj}], \qquad j \in V, \tag{4.3d}$$

$$u_j \in [b^u_{lj}, b^u_{uj}], \qquad j \in V, \tag{4.3e}$$

$$\sum_{j \in V} f_j(x_j, u_j) = 0. \tag{4.3f}$$

The outgoing control problem consists of the *objective function* (4.3a) and the following five constraints: *dynamics* (4.3b), *local ranges* (4.3c), *local state bounds* (4.3d), *local control bounds* (4.3e) and *global equalities* (4.3f).

The *nonlinear tree-sparse problem in incoming control form* reads

$$\min_{u,x} \quad \sum_{j\in V} \phi_{ij}(x_i, u_j) + \sum_{j\in V} \phi_j(x_j) \tag{4.4a}$$

$$\text{s.t.} \quad g_j(x_i, u_j) - x_j = 0, \qquad j \in V, \tag{4.4b}$$

$$r_{ij}(x_i, u_j) \in [r^u_{lj}, r^u_{uj}], \qquad j \in V, \tag{4.4c}$$

$$r_j(x_j) \in [r^x_{lj}, r^x_{uj}], \qquad j \in V, \tag{4.4d}$$

$$u_j \in [b^u_{lj}, b^u_{uj}], \qquad j \in V, \tag{4.4e}$$

$$x_j \in [b^x_{lj}, b^x_{uj}], \qquad j \in V, \tag{4.4f}$$

$$\sum_{j\in V} f_{ij}(x_i, u_j) + \sum_{j\in V} f_j(x_j) = 0. \tag{4.4g}$$

The incoming control problem includes the *objective function* (4.4a) and is constrained by *dynamic equations* (4.4b), *mixed ranges* (4.4c), *local ranges* (4.4d), *control bounds* (4.4e), *state bounds* (4.4f) and *global equality constraints* (4.4g).

Problem Dimensions

The problem dimensions for the TSPs are denoted as follows. The numbers of state and control node variables are n^x_j and n^u_j, respectively. Their sum forms the number of primal node variables n^v_j. In both control cases, m^g denotes the number of global equality constraints (4.3f) and (4.4g), respectively. The number of range constraints (4.3c) for node $j \in V$ is denoted by l^r_j. In the incoming control case, the number of range constraints is the sum $l^r_j = l^{ru}_j + l^{rx}_j$ where l^{ru}_j denotes the number of mixed ranges (4.4c) and l^{rx}_j the number of pure state ranges (4.4d). Dimensions with a node subscript are referred to as *node dimensions* and corresponding *total dimensions* are obtained by summing up these node dimensions. A total dimension is denoted by the same letter as the corresponding node dimension without the node subscript of the latter. Table 4.1 lists all occurring dimensions and their compositions.

NLP Mapping of the Outgoing Control Problem

In the outgoing control case, primal node variables are combined into the variable vector $y \in \mathbb{R}^{n^v}$,

$$y = ((x_j, u_j))_{j\in V}\,. \tag{4.5}$$

Table 4.1.: Node dimensions and total dimensions of tree-sparse problems

Dimension	Composition	Control form	Description
n_j^x	-	outgoing and incoming	state node variables
n_j^u	-	outgoing and incoming	control node variables
n_j^v	$n_j^x + n_j^u$	outgoing and incoming	primal node variables
l_j^r	-	outgoing	node range constraints
l_j^{ru}	-	incoming	local node range constraints
l_j^{rx}	-	incoming	mixed node range constraints
l_j^r	$l_j^{ru} + l_j^{rx}$	incoming	node range constraints
m^g	-	outgoing and incoming	global constraints
n^x	$\sum_{j\in V} n_j^x$	outgoing and incoming	state variables
n^u	$\sum_{j\in V} n_j^u$	outgoing and incoming	control variables
n^v	$\sum_{j\in V} n_j^v$	outgoing and incoming	primal variables
l^r	$\sum_{j\in V} l_j^r$	outgoing and incoming	range constraints
l^{ru}	$\sum_{j\in V} l_j^{ru}$	incoming	local range constraints
l^{rx}	$\sum_{j\in V} l_j^{rx}$	incoming	mixed range constraints
n	n^v	outgoing and incoming	primal variables
m	$n^x + m^g$	outgoing and incoming	equality constraints
k	l^r	outgoing and incoming	range constraints

The outgoing control problem (4.3) is an NLP (4.1) with objective $\phi : \mathbb{R}^{n^v} \to \mathbb{R}$,

$$\phi(y) = \sum_{j\in V} \phi_j(x_j, u_j), \tag{4.6}$$

equality constraints $c_{\mathcal{E}} : \mathbb{R}^{n^v} \to \mathbb{R}^{n^x+m^g}$,

$$c_{\mathcal{E}}(y) = \left(\left(g_j(x_i, u_i) - x_j\right)_{j\in V}, \sum_{j\in V} f_j(x_j, u_j) \right), \tag{4.7}$$

and range constraints $c_{\mathcal{R}} : \mathbb{R}^{n^v} \to \mathbb{R}^{l^r}$,

$$c_{\mathcal{R}}(y) = \left(r_j(x_j, u_j)\right)_{j\in V}. \tag{4.8}$$

The lower and upper bounds $b_l, b_u \in \mathbb{R}^{n^v}$ are given by

$$b_l = \left((b_{lj}^x, b_{lj}^u)\right)_{j\in V} \quad \text{and} \quad b_u = \left((b_{uj}^x, b_{uj}^u)\right)_{j\in V}, \tag{4.9}$$

and the lower and upper range bounds $r_l, r_u \in \mathbb{R}^{l^r}$ are defined as

$$r_l = \left(r_{lj}\right)_{j\in V} \quad \text{and} \quad r_u = \left(r_{uj}\right)_{j\in V}. \tag{4.10}$$

NLP Mapping of the Incoming Control Problem

In the incoming control case, the order of the primal node variables switches and the remaining mappings are adjusted to this changed order. The vectors $y, b_l, b_u \in \mathbb{R}^{n^v}$ read

$$y = ((u_j, x_j))_{j\in V}\,, \quad b_l = \left((b^u_{lj}, b^x_{lj})\right)_{j\in V} \quad \text{and} \quad b_u = \left((b^u_{uj}, b^x_{uj})\right)_{j\in V}. \tag{4.11}$$

The incoming control problem (4.4) is an NLP (4.1) with objective $\phi : \mathbb{R}^{n^v} \to \mathbb{R}$,

$$\phi(y) = \sum_{j\in V} \phi_{ij}(x_i, u_j) + \sum_{j\in V} \phi_j(x_j), \tag{4.12}$$

equality constraints $c_{\mathcal{E}} : \mathbb{R}^{n^v} \to \mathbb{R}^{n^x+m^g}$,

$$c_{\mathcal{E}}(y) = \left((g_j(x_i, u_j) - x_j)_{j\in V}\,, \sum_{j\in V} f_{ij}(x_i, u_j) + \sum_{j\in V} f_j(x_j) \right), \tag{4.13}$$

and range constraints $c_{\mathcal{R}} : \mathbb{R}^{n^v} \to \mathbb{R}^{l^r}$,

$$c_{\mathcal{R}}(y) = ((r_{ij}(x_j, u_j), r_j(x_j)))_{j\in V}\,. \tag{4.14}$$

The lower and upper range bounds $r_l, r_u \in \mathbb{R}^{l^r}$ are mapped accordingly by

$$r_l = \left((r^u_{lj}, r^x_{lj})\right)_{j\in V} \quad \text{and} \quad r_u = \left((r^u_{uj}, r^x_{uj})\right)_{j\in V}. \tag{4.15}$$

4.1.3. Characteristics of Tree-Sparse Problems

TSPs are NLPs (4.1) with the specific forms of problem functions where f, $c_{\mathcal{E}}$ and $c_{\mathcal{R}}$ consist of nonlinear node functions and are either sums of the latter or pure node functions themselves. More precisely, TSPs are *sufficiently smooth node-separable* optimization problems that *couple* node variables in two ways: all variables are coupled *linearly* and, additionally, state-defining variables are coupled *nonlinearly*. In the following, these key characteristics as well as the stochastic background are explained in more detail.

Smoothness

For a TSP to be smooth, all node functions are assumed to be twice continuously differentiable,

$$\phi_j, \phi_{ij}, g_j, r_j, r_{ij}, f_j, f_{ij} \in \mathcal{C}^2. \tag{4.16}$$

To approximate the tree-sparse Hessians of the Lagrangians in the context of a quasi-Newton approach (cf. Sect. 4.5), it is assumed that at least first-order derivatives of the functions (4.16) are available.

Node-Separability

A *partially separable* function $\zeta : \mathbb{R}^N \to \mathbb{R}$ is a sum of *contributions* $\zeta_\kappa : \mathbb{R}^N \to \mathbb{R}$ that depend only on a small subset of the variables $y \in \mathbb{R}^N$, i.e.

$$\zeta(y) = \sum_{\kappa=1}^{M} \zeta_\kappa \left((y_\iota)_{\iota \in \mathcal{J}_\kappa} \right) \quad \text{with} \quad \mathcal{J}_\kappa \subseteq \{1, \dots, N\} \quad \text{for} \quad \kappa = 1, \dots, M. \tag{4.17}$$

Node-separabilities—as defined by (3.10) and further discussed in Sect. 3.2.1—are special forms of partial separability. In the outgoing control case, the objective function (4.3a) and the global constraint functions (4.3f) are partially separable functions of the form

$$\zeta(y) = \sum_{j \in V} \zeta_j \left((y_\iota)_{\iota \in \mathcal{J}_j} \right), \tag{4.18}$$

where $\mathcal{J}_j$ comprises the indices of $(x_{j\tau})_{\tau=1,\dots,n_j^x}$ and $(u_{j\tau})_{\tau=1,\dots,n_j^u}$ in the primal vector $y \in \mathbb{R}^n$. In the incoming control case, the objective (4.4a) and the global constraints (4.4g) have the form

$$\zeta(y) = \sum_{j \in V} \zeta_{ij} \left((y_\iota)_{\iota \in \mathcal{J}_{ij}} \right) + \sum_{j \in V} \zeta_j \left((y_\iota)_{\iota \in \mathcal{J}_j} \right). \tag{4.19}$$

The index set $\mathcal{J}_{ij}$ consists of the indices of $(x_{i\tau})_{\tau=1,\dots,n_i^x}$ and $(u_{j\tau})_{\tau=1,\dots,n_j^u}$ in y, whereas $\mathcal{J}_j$ comprises those of $(x_{j\tau})_{\tau=1,\dots,n_j^x}$. The forms (4.18) and (4.19) take center stage in designing tree-sparse Hessian update strategies in Sect. 4.5.

Coupling of Variables

Let v_{j_1} and w_{j_2} be two (vectors of) node variables corresponding to the nodes $j_1, j_2 \in V$. A function ζ is said to *couple* v_{j_1} and w_{j_2} if ζ depends on both variables. The variables are coupled *linearly* by ζ if mixed second-order derivatives vanish, i.e. if $\nabla^2_{v_{j_1} w_{j_2}} \zeta = 0$, and they are coupled *nonlinearly* if $\nabla^2_{v_{j_1} w_{j_2}} \zeta \neq 0$. In the TSPs, only state-defining variables are coupled nonlinearly, that is the node variables x_j and u_j by the node functions ϕ, g_j, r_j, f_j in the outgoing control problem (4.3) as well as the node variables x_i and u_j by the node functions ϕ_{ij}, g_j, r_{ij}, f_{ij} in the incoming control problem (4.4). The objectives (4.3a) and (4.4a) as well as the global constraints (4.3f) and (4.4g) couple all node variables linearly. Additionally, the

dynamics (4.3b) and (4.4b) also couple the states x_j linearly with the state-defining variables (see global linear coupling and path coupling in Sect. 3.2.2).

Stochastic Background

Motivated by nonlinear multistage stochastic problems (cf. Sect. 3.1.2), each node $j \in V$ of the tree corresponding to a TSP represents an event with an associated probability p_j, and the objective as well as the global constraint functions of the TSP are expected values. For the node probabilities it holds

$$p_j = \sum_{k \in S(j)} p_k \quad \text{for} \quad j \in V \quad \text{and} \quad 1 = \sum_{j \in L_t} p_j \quad \text{for} \quad t = 0, \ldots, T. \tag{4.20}$$

Node functions in expected values consume the node probabilities. Hence, the objectives (4.3a) and (4.3f) read

$$\phi(y) = \sum_{j \in V} p_j \tilde{\phi}_j(x_j, u_j) \quad \text{and} \quad \phi(y) = \sum_{j \in V} p_j \left(\tilde{\phi}_{ij}(x_i, u_j) + \tilde{\phi}_j(x_j) \right) \tag{4.21}$$

with $\phi_j = p_j \tilde{\phi}_j$ and $\phi_{ij} = p_j \tilde{\phi}_{ij}$, and the global constraints (4.4a) and (4.4g) read

$$f(y) = \sum_{j \in V} p_j \tilde{f}_j(x_j, u_j) \quad \text{and} \quad f(y) = \sum_{j \in V} p_j \left(\tilde{f}_{ij}(x_i, u_j) + \tilde{f}_j(x_j) \right) \tag{4.22}$$

with $f_j = p_j \tilde{f}_j$ and $f_{ij} = p_j \tilde{f}_{ij}$.

4.2. Perturbed KKT Conditions and the Primal-Dual System

The TSPs are solved by a primal-dual interior-point method that uses a filter line-search approach as globalization strategy (cf. Sect. 2.2.2). In each iteration of the IPM, the search direction is obtained from the primal-dual system (2.21). By slacking the inequalities of a TSP and perturbing the KKT conditions corresponding to this reformulation, the primal-dual system results from applying the Newton's methods to the perturbed nonlinear system (see homotopy approach in Sect. 2.2.1).

Next, the tree-sparse primal-dual systems for the TSPs are stated for both explicit control cases. Section 4.2.1 follows the lines of Sect. 2.2.1 and applies the homotopy approach to the general NLP (4.1). Using the NLP mappings in Sect. 4.1.2, the tree-sparse primal-dual systems

are presented by means of the previous considerations. Section 4.2.2 states the perturbed KKT conditions and the primal-dual system for the outgoing control TSP. The incoming control case is discussed in Sect. 4.2.3.

Notation: In the following, the letter ϕ is used with node subscripts (i, j, k) to denote objective node functions (ϕ_i, ϕ_j, ϕ_k) as well as with the fixed subscripts u and l to denote right-hand side barrier terms (ϕ_l and ϕ_u). The letter μ is used as Lagrange multiplier whereas μ^{bp} refers to the barrier parameter in IPMs.

4.2.1. The General NLP Case

The difference between the NLP (2.1) introduced in Chap. 2 and the formulation (4.1) used in this chapter is that the latter incorporates inequality constraints in a more detailed presentation. However, both NLP formulations are equivalent (cf. Sect. 4.1.2). Thus, only the presentation of the primal-dual system becomes more cumbersome for (4.1) while the homotopy approach presented in Sect. 2.2.1 is the same for both NLPs and consists of the following four steps:

1. Restate the considered NLP with slacks for the inequality constraints, i.e. for the ranges and the simple bounds in (4.1).
2. Derive the KKT conditions from the Lagrangian corresponding to the slacked NLP.
3. Perturb the KKT conditions by relaxing the complementarity conditions.
4. Apply the Newton's method to the perturbed KKT conditions.

Following this guideline, the NLP with slacked ranges and bounds, the corresponding perturbed KKT conditions and, finally, the resulting primal-dual system are described next.

NLP with Slacks and the Lagrangian

Introducing slack variables $t_l, t_u \in \mathbb{R}^k$ for the range constraints (4.1c) as well as $s_l, s_u \in \mathbb{R}^n$ for the bounds (4.1d), the slacked reformulation of problem (4.1) reads

$$\begin{aligned}
\min_{y} \quad & \phi(y) && \text{(4.23a)}\\
\text{s.t.} \quad & c_{\mathcal{E}}(y) = 0, && \text{(4.23b)}\\
& c_{\mathcal{R}}(y) - t_l - r_l = 0, && \text{(4.23c)}\\
& -c_{\mathcal{R}}(y) - t_u + r_u = 0, && \text{(4.23d)}\\
& y - s_l - b_l = 0, && \text{(4.23e)}\\
& -y - s_u + b_u = 0, && \text{(4.23f)}\\
& t_l, t_u, s_l, s_u \geq 0. && \text{(4.23g)}
\end{aligned}$$

The Lagrange multipliers for the equality constraints (4.23b) are denoted by $z \in \mathbb{R}^m$, $(v_l, v_u) \in \mathbb{R}^{2k}$ are the duals corresponding to the range constraints (4.23c) and (4.23d), and $(u_l, u_u) \in \mathbb{R}^{2n}$ are the associated duals to the bounds (4.23e) and (4.23f). Combining the primal slacks into $s = (s_l, s_u, t_l, t_u)$ and using $\xi \in \mathbb{R}^{2k+2n}$ to denote the corresponding dual slacks, the Lagrangian to (4.23) reads

$$\begin{aligned}
\mathcal{L}(y, \eta) = f(y) &- z^T c_{\mathcal{E}}(y) - \xi^T s \\
&- v_l^T \left(c_{\mathcal{R}}(y) - t_l - r_l\right) - v_u^T \left(-c_{\mathcal{R}}(y) - t_u + r_u\right) \\
&- u_l^T \left(y - s_l - b_l\right) - u_u^T \left(-y - s_u + b_u\right),
\end{aligned} \tag{4.24}$$

where the vector η contains the primal slacks s and all dual variables, i.e.

$$\eta = (z, s, v_l, v_u, u_l, u_u, \xi). \tag{4.25}$$

Hence, η subsumes all variables except for the primal variables y. This notation is frequently used in this section and in the context of the quasi-Newton approach in Sect. 4.5 to avoid cumbersome specifications of variable dependencies.

Perturbed KKT Conditions

The μ^{bp}-perturbed KKT conditions for NLP (4.23) are derived from its corresponding Lagrangian (4.24). They comprise the dual feasibility conditions

$$\nabla_y \mathcal{L}(y,\eta) = \nabla_y \phi(y) - \nabla c_{\mathcal{E}}(y)^T z - \nabla c_{\mathcal{R}}(y)^T (v_l - v_u) - (u_l - u_u) = 0, \tag{4.26}$$

the primal feasibility conditions (4.23b) to (4.23g), the μ^{bp}-perturbed complementarity conditions

$$S_l V_l e = S_u V_u e = T_l U_l e = T_u U_u e = \mu^{bp} e, \tag{4.27}$$

and the nonnegativity conditions

$$u_l, u_u, v_l, v_u \geq 0. \tag{4.28}$$

The capital letters S, T, U, V denote diagonal matrices to the corresponding vectors s, t, u, v, and e is the vector of ones in appropriate dimension. The dual slacks ξ are already eliminated from the conditions above (cf. Sect. 2.2.1).

Primal-Dual System

The *primal infeasibilities* from (4.23b) to (4.23f) are denoted as follows:

$$\alpha(y) := c_{\mathcal{E}}(y), \tag{4.29a}$$

$$\rho_l(y, t_l) := c_{\mathcal{R}}(y) - t_l - r_l, \tag{4.29b}$$

$$\rho_u(y, t_u) := -c_{\mathcal{R}}(y) - t_u + r_u, \tag{4.29c}$$

$$\beta_l(y, s_l) := y - s_l - b_l, \tag{4.29d}$$

$$\beta_u(y, s_u) := y - s_u + b_u. \tag{4.29e}$$

The so-called *barrier diagonal matrices* are given by

$$\Phi_l := S_l^{-1} U_l, \quad \Phi_u := S_u^{-1} U_u, \tag{4.30a}$$

$$\Psi_l := T_l^{-1} V_l, \quad \Psi_u := T_u^{-1} V_u, \tag{4.30b}$$

and the corresponding *barrier right-hand side terms* read

$$\phi_l := u_l - \mu^{bp} S_l^{-1} e, \quad \phi_u := u_u - \mu^{bp} S_u^{-1} e, \tag{4.31a}$$

$$\psi_l := v_l - \mu^{bp} T_l^{-1} e, \quad \psi_u := v_u - \mu^{bp} T_u^{-1} e. \tag{4.31b}$$

Omitting the variable dependencies, the primal-dual system (2.21) resulting from problem (4.23) takes the form

$$\Omega_{\mathrm{pd}} \omega_{\mathrm{pd}} = -w_{\mathrm{pd}}, \tag{4.32}$$

where the system matrix reads

$$\Omega_{\mathrm{pd}} = \left[\begin{array}{ccccc|ccccc}
H & & & & & \nabla c_{\mathcal{E}}^T & I & -I & \nabla c_{\mathcal{R}}^T & -\nabla c_{\mathcal{R}}^T \\
& \Phi_l & & & & & -I & & & \\
& & \Phi_u & & & & & -I & & \\
& & & \Psi_l & & & & & -I & \\
& & & & \Psi_u & & & & & -I \\
\hline
\nabla c_{\mathcal{E}} & & & & & & & & & \\
I & -I & & & & & & & & \\
-I & & -I & & & & & & & \\
\nabla c_{\mathcal{R}} & & & -I & & & & & & \\
-\nabla c_{\mathcal{R}} & & & & -I & & & & &
\end{array}\right], \tag{4.33}$$

the right-hand side is given by

$$w_{\mathrm{pd}}^T = \left(\nabla_y \mathcal{L}, \phi_l, \phi_u, \psi_l, \psi_u, \alpha, \beta_l, \beta_u, \rho_l, \rho_u\right), \tag{4.34}$$

and the search direction reads

$$\omega_{\mathrm{pd}}^T = \left(\Delta y, \Delta s_l, \Delta s_u, \Delta t_l, \Delta t_u, -\Delta z, -\Delta u_l, -\Delta u_u, -\Delta v_l, -\Delta v_u\right). \tag{4.35}$$

The block H in the upper left of Ω_{pd} (4.33) denotes the Hessian of the Lagrangian (4.24), i.e.

$$H := \nabla_{yy}^2 \mathcal{L}(y, \eta) = \nabla_{yy}^2 \phi(y) - \sum_{\tau \in \mathcal{E}} z_\tau \nabla_{yy}^2 c_\tau(y) - \sum_{\tau \in \mathcal{R}} \left(v_{l\tau} - v_{u\tau}\right) \nabla_{yy}^2 c_\tau(y). \tag{4.36}$$

4.2.2. The Outgoing Control Case

The TSPs are NLPs of the form (4.1), making the previous discussions in Sect. 4.2.1 for the general NLP case also applicable to the TSPs. Hence, from a global point of view the tree-sparse primal-dual matrices Ω_{pd} take the form (4.33). For solving the primal-dual system efficiently, a more detailed presentation of (4.33) is of interest that highlights the characteristic structures of the TSPs. These tree-sparse presentations, which are based on the mappings introduced in Sect. 4.1.2, are established in the following three steps:

1. Supplementing mappings for slack variables and Lagrange multipliers.
2. Introducing a presentation of the Lagrangian based on node functions and notation for the node subblocks in the tree-sparse primal-dual system.
3. Presenting the specific tree structures in the Hessian of the Lagrangian and the Jacobians of the constraints.

The TSP in the outgoing control form (4.3) is considered first, the incoming TSP is discussed in Sect. 4.3.3. For the outgoing control case, recall the mappings (4.6) to (4.10) as well as the order of primal node variables (x_j, u_j) in the variable vector y (4.5).

Mapping of Slacks and Duals

Slack variables are denoted by $t_{lj}, t_{uj} \in \mathbb{R}^{l_j^r}$ for the range constraints (4.3c) as well as by $s_{lj}^x, s_{uj}^x \in \mathbb{R}^{n_j^x}$ and $s_{lj}^u, s_{uj}^u \in \mathbb{R}^{n_j^u}$ for the simple bounds (4.3d) and (4.3e), respectively. The node variables are mapped into the vectors s_l, s_u, t_l, t_u by

$$s_l = \left((s_{lj}^x, s_{lj}^u)\right)_{j\in V}, \quad s_u = \left((s_{uj}^x, s_{uj}^u)\right)_{j\in V}, \quad t_l = (t_{lj})_{j\in V}, \quad t_u = (t_{uj})_{j\in V}. \tag{4.37}$$

The duals corresponding to the equality constraints are denoted by $\lambda_j \in \mathbb{R}^{n_j^x}$ for the dynamics (4.3b) and by $\mu \in \mathbb{R}^{m^g}$ for the global constraints (4.3f), hence

$$z = \left((\lambda_j)_{j\in V}, \mu\right) \in \mathbb{R}^{n^x + m^g}. \tag{4.38}$$

The Lagrange multipliers for the slacked range constraints and bounds are $v_{lj}, v_{uj} \in \mathbb{R}^{l_j^r}$, $u_{lj}^x, u_{uj}^x \in \mathbb{R}^{n_j^x}$ as well as $u_{lj}^u, u_{uj}^u \in \mathbb{R}^{n_j^u}$. They are mapped by

$$u_l = \left((u_{lj}^x, u_{lj}^u)\right)_{j\in V}, \quad u_u = \left((u_{uj}^x, u_{uj}^u)\right)_{j\in V}, \quad v_l = (v_{lj})_{j\in V}, \quad v_u = (v_{uj})_{j\in V}. \tag{4.39}$$

Table 4.2.: Matrix node subblock dimensions – Outgoing control case

Subblock	H_j	K_j	J_j	G_j	E_j	F_j^r	D_j^r	F_j	D_j
Number of rows	n_j^x	n_j^u	n_j^u	n_j^x	n_j^x	l_j^r	l_j^r	m^g	m^g
Number of columns	n_j^x	n_j^u	n_j^x	n_i^x	n_i^u	n_j^x	n_j^u	n_j^x	n_j^u

Lagrangian and Derivatives

In the outgoing control case, the Lagrangian (4.24) reads

$$\mathcal{L} = \sum_{j \in V} \mathcal{L}_j - \xi^T s, \tag{4.40}$$

where for each node $j \in V$ the introduced node functions $\mathcal{L}_j$ are defined by

$$\begin{aligned}\mathcal{L}_j(x_j, u_j, \eta) := \phi_j(x_j, u_j) + \lambda_j^T x_j - \sum_{k \in S(j)} \lambda_k^T g_k(x_j, u_j) - \mu^T f_j(x_j, u_j) \\ - v_{lj}^T \left(r_j(x_j, u_j) - t_{lj} - r_{lj}\right) - v_{uj}^T \left(-r_j(x_j, u_j) - t_{uj} + r_{uj}\right) \\ - \left(u_{lj}^x\right)^T \left(x_j - s_{lj}^x - b_{lj}^x\right) - \left(u_{uj}^x\right)^T \left(-x_j - s_{uj}^x + b_{uj}^x\right) \\ - \left(u_{lj}^u\right)^T \left(u_j - s_{lj}^u - b_{lj}^u\right) - \left(u_{uj}^u\right)^T \left(-u_j - s_{uj}^u + b_{uj}^u\right),\end{aligned} \tag{4.41}$$

with $\eta = (z, s, v_l, v_u, u_l, u_u, \xi)$ as defined in (4.25). The Jacobians of the constraint node functions, which dimensions are listed in Table 4.2, read

$$G_j := \nabla_{x_i} g_j(x_i, u_i), \quad F_j^r := \nabla_{x_j} r_j(x_j, u_j), \quad F_j := \nabla_{x_j} f_j(x_j, u_j), \tag{4.42a}$$

$$E_j := \nabla_{u_i} g_j(x_i, u_i), \quad D_j^r := \nabla_{u_j} r_j(x_j, u_j), \quad D_j := \nabla_{u_j} f_j(x_j, u_j). \tag{4.42b}$$

The gradient of the Lagrangian (4.40) with respect to the primal variables y reads

$$\nabla_y \mathcal{L} = \left((\nabla_{x_j} \mathcal{L}, \nabla_{u_j} \mathcal{L})\right)_{j \in V} = \left((\nabla_{x_j} \mathcal{L}_j, \nabla_{u_j} \mathcal{L}_j)\right)_{j \in V}, \tag{4.43}$$

where the partial derivatives of the node functions are given by

$$\nabla_{x_j} \mathcal{L}_j = \nabla_{x_j} \phi_j + \lambda_j - \sum_{k \in S(j)} \lambda_k^T G_k - F_j^T \mu - \left(F_j^r\right)^T (v_{lj} - v_{uj}) - \left(u_{lj}^x - u_{uj}^x\right), \tag{4.44a}$$

$$\nabla_{u_j} \mathcal{L}_j = \nabla_{u_j} \phi_j \qquad - \sum_{k \in S(j)} \lambda_k^T E_k - D_j^T \mu - \left(D_j^r\right)^T (v_{lj} - v_{uj}) - \left(u_{lj}^u - u_{uj}^u\right). \tag{4.44b}$$

To state the Hessian of the Lagrangian (4.40) with respect to the primal variables y, the following three matrix node subblocks are introduced:

$$H_j := \nabla^2_{x_j x_j} \mathcal{L}_j, \quad K_j := \nabla^2_{u_j u_j} \mathcal{L}_j, \quad J_j := \nabla^2_{u_j x_j} \mathcal{L}_j. \tag{4.45}$$

Omitting the variable dependencies, the Hessian node subblocks (4.45) are explicitly given by

$$H_j = \nabla^2_{x_j x_j} \phi_j - \sum_{\tau=1}^{m^g} \mu_\tau \nabla^2_{x_j x_j} f_{j\,\tau} - \sum_{k \in S(j)} \sum_{\tau=1}^{n_j^x} \lambda_{k\tau} \nabla^2_{x_j x_j} g_{k\tau} - \sum_{\tau=1}^{l_j^r} (v_{lj} - v_{uj})_\tau \nabla^2_{x_j x_j} r_{j\,\tau}, \tag{4.46a}$$

$$K_j = \nabla^2_{u_j u_j} \phi_j - \sum_{\tau=1}^{m^g} \mu_\tau \nabla^2_{u_j u_j} f_{j\,\tau} - \sum_{k \in S(j)} \sum_{\tau=1}^{n_j^x} \lambda_{k\tau} \nabla^2_{u_j u_j} g_{k\tau} - \sum_{\tau=1}^{l_j^r} (v_{lj} - v_{uj})_\tau \nabla^2_{u_j u_j} r_{j\,\tau}, \tag{4.46b}$$

$$J_j = \nabla^2_{x_j u_j} \phi_j - \sum_{\tau=1}^{m^g} \mu_\tau \nabla^2_{x_j u_j} f_{j\,\tau} - \sum_{k \in S(j)} \sum_{\tau=1}^{n_j^x} \lambda_{k\tau} \nabla^2_{x_j u_j} g_{k\tau} - \sum_{\tau=1}^{l_j^r} (v_{lj} - v_{uj})_\tau \nabla^2_{x_j u_j} r_{j\,\tau}, \tag{4.46c}$$

The dimensions of the Hessian node subblocks are also listed in Table 4.2.

Tree-Structured Block Matrices

The KKT matrix blocks resulting from tree-sparse problems reflect the underlying tree topology (cf. Sect. 3.2.3). In the following, the tree structures of the KKT matrix blocks are demonstrated with respect to the tree in Fig. 4.1.

The Hessian of the Lagrangian (4.40) and the Jacobian of the range constraints (4.3c) are block-diagonal matrices. With the node subblock notation (4.45), the Hessian of the Lagrangian takes the form

$$\nabla^2_{yy} \mathcal{L} = \begin{bmatrix}
H_0 & J_0^T & & & & & & & & & \\
J_0 & K_0 & & & & & & & & & \\
 & & H_1 & J_1^T & & & & & & & \\
 & & J_1 & K_1 & & & & & & & \\
 & & & & H_2 & J_2^T & & & & & \\
 & & & & J_2 & K_2 & & & & & \\
 & & & & & & H_3 & J_3^T & & & \\
 & & & & & & J_3 & K_3 & & & \\
 & & & & & & & & H_4 & J_4^T & \\
 & & & & & & & & J_4 & K_4 & \\
 & & & & & & & & & & H_5 & J_5^T \\
 & & & & & & & & & & J_5 & K_5
\end{bmatrix}. \tag{4.47}$$

The Hessian $\nabla^2_{yy} f$ of the objective function (4.6) has the same block-diagonal structure as the Hessian of the Lagrangian. By means of the node subblocks (4.42), the Jacobian $\nabla c_{\mathcal{R}}$ of the range constraints (4.8) reads

$$\nabla c_{\mathcal{R}} = \begin{bmatrix} F_0^r & D_0^r & & & & & & & & & & \\ & & F_1^r & D_1^r & & & & & & & & \\ & & & & F_2^r & D_2^r & & & & & & \\ & & & & & & F_3^r & D_3^r & & & & \\ & & & & & & & & F_4^r & D_4^r & & \\ & & & & & & & & & & F_5^r & D_5^r \end{bmatrix}. \tag{4.48}$$

The Jacobian $\nabla c_{\mathcal{E}}^T = [G^T \; F^T]$ of the equality constraints (4.7) comprises a dynamic block $G \in \mathbb{R}^{n^x \times n^v}$ corresponding to the dynamic constraints (4.3b) and a global constraint block $F \in \mathbb{R}^{m^g \times n^v}$ corresponding to the global equalities (4.3f). The global constraint part is a block-dense row with node subblocks $[F_j \; D_j]$,

$$F = \begin{bmatrix} F_0 & D_0 & F_1 & D_1 & F_2 & D_2 & F_3 & D_3 & F_4 & D_4 & F_5 & D_5 \end{bmatrix}. \tag{4.49}$$

The dynamic part of $\nabla c_{\mathcal{E}}$ features the stochastic staircase structure (cf. Sect. 3.2.3) with node subblocks $[-I \; 0]$ on the diagonal and $[G_j \; E_j]$ on the secondary diagonal corresponding to the predecessor i. For the tree in Fig. 4.1, the dynamic part reads

$$G = \begin{bmatrix} -I & & & & & & & & & & & \\ G_1 & E_1 & -I & & & & & & & & & \\ G_2 & E_2 & & & -I & & & & & & & \\ & & G_3 & E_3 & & & -I & & & & & \\ & & G_4 & E_4 & & & & & -I & & & \\ & & & & G_5 & E_5 & & & & & -I & \end{bmatrix}. \tag{4.50}$$

4.2.3. The Incoming Control Case

For the TSP in the incoming control form (4.4), the tree-sparse presentation of the primal-dual system (4.32) proceeds analogously to the outgoing control TSP in Sect. 4.2.2. However, a separate discussion is necessary since the nonlinear coupling of node variables is different. In the incoming control case, the controls u_j are coupled nonlinearly with the states x_i of the predecessor $i = \pi(j)$ instead of being coupled nonlinearly with the states x_j of the same node j.

For the subsequent discussions, recall the mappings (4.12) to (4.15) and the changed order of primal node variables (u_j, x_j) in the vector y (4.11).

Mapping of Slacks and Duals

The slacks for the bounds (4.4e) and (4.4f) are $s^u_{lj}, s^u_{uj} \in \mathbb{R}^{n^u_j}$ and $s^x_{lj}, s^x_{uj} \in \mathbb{R}^{n^x_j}$, respectively. Slacks for mixed ranges (4.4c) and local ranges (4.4d) are denoted by $t^u_{lj}, t^u_{uj} \in \mathbb{R}^{l^{ru}_j}$ and $t^x_{lj}, t^x_{uj} \in \mathbb{R}^{l^{rx}_j}$, respectively. The aggregation of the slacks is based on (4.11) and reads

$$s_l = \left((s^u_{lj}, s^x_{lj})\right)_{j\in V}, \quad s_u = \left((s^u_{uj}, s^x_{uj})\right)_{j\in V}, \tag{4.51a}$$

$$t_l = \left((t^u_{lj}, t^x_{lj})\right)_{j\in V}, \quad t_u = \left((t^u_{uj}, t^x_{uj})\right)_{j\in V}. \tag{4.51b}$$

Dual variables are denoted the same way as in the outgoing control case: λ_j corresponds to the dynamics (4.4b), μ to the global constraints (4.4g), v^u_{lj}, v^u_{uj} and v^x_{lj}, v^x_{uj} to the slacked range constraints (4.4c) and (4.4d), respectively, and u^u_{lj}, u^u_{uj} and u^x_{lj}, u^x_{uj} to the slacked versions of the simple bounds (4.4e) and (4.4f), respectively. Dual slacks are combined analogously to their primal counterparts in (4.51), i.e.

$$u_l = \left((u^u_{lj}, u^x_{lj})\right)_{j\in V}, \quad u_u = \left((u^u_{uj}, u^x_{uj})\right)_{j\in V}, \tag{4.52a}$$

$$v_l = \left((v^u_{lj}, v^x_{lj})\right)_{j\in V}, \quad v_u = \left((v^u_{uj}, v^x_{uj})\right)_{j\in V}. \tag{4.52b}$$

Lagrangian and Derivatives

In the incoming control case, the Lagrangian (4.24) is stated as

$$\mathcal{L} = \sum_{j\in V} \mathcal{L}_{ij} + \sum_{j\in V} \mathcal{L}_j - \xi^T s, \tag{4.53}$$

where $\mathcal{L}_{ij}$ includes all nonlinear node functions that couple x_i and u_j,

$$\begin{aligned}\mathcal{L}_{ij}(x_i, u_j, \eta) := \phi_{ij}(x_i, u_j) &- \lambda_j^T g_j(x_i, u_j) - \mu^T f_{ij}(x_i, u_j) \\ &- \left(v^u_{lj}\right)^T \left(r_{ij}(x_i, u_j) - t^u_{lj} - r^u_{lj}\right) - \left(v^u_{uj}\right)^T \left(-r_{ij}(x_i, u_j) - t^u_{uj} + r^u_{uj}\right) \\ &- \left(u^u_{lj}\right)^T \left(u_j - s^u_{lj} - b^u_{lj}\right) - \left(u^u_{uj}\right)^T \left(-u_j - s^u_{uj} + b^u_{uj}\right),\end{aligned} \tag{4.54}$$

Table 4.3.: Matrix node subblock dimensions – Incoming control case

Subblock	K_j	H_j	J_j	G_j	E_j	F_{ij}^r	D_j^r	F_j^r	F_{ij}	D_j	$\bar{F}_j$
Number of rows	n_j^u	n_j^x	n_j^u	n_j^x	n_j^x	l_j^{ru}	l_j^{ru}	l_j^{rx}	m^g	m^g	m^g
Number of columns	n_j^u	n_j^x	n_j^x	n_i^x	n_j^u	n_i^x	n_j^u	n_j^x	n_i^x	n_j^u	n_j^x

and the node function $\mathcal{L}_j$ comprises the decoupled terms,

$$\begin{aligned}\mathcal{L}_j(x_j,\eta) := \phi_j(x_j) + \lambda_j^T x_j - \mu^T f_j(x_j) \\ - \left(v_{lj}^x\right)^T \left(r_j(x_j) - t_{lj}^x - r_{lj}^x\right) - \left(v_{uj}^x\right)^T \left(-r_j(x_j) - t_{uj}^x + r_{uj}^x\right) \\ - \left(u_{lj}^x\right)^T \left(u_j - s_{lj}^x - b_{lj}^x\right) - \left(u_{uj}^x\right)^T \left(-u_j - s_{uj}^x + b_{uj}^x\right). \end{aligned} \tag{4.55}$$

The Jacobian node subblocks of the dynamics (4.4b) are given by

$$G_j := \nabla_{x_i} g_j(x_i,u_j) \quad \text{and} \quad E_j := \nabla_{u_j} g_j(x_i,u_j). \tag{4.56}$$

The Jacobian node subblocks of the ranges (4.4c) and (4.4d) as well as of the global constraints (4.4g) read

$$F_{ij}^r := \nabla_{x_i} r_{ij}(x_i,u_j), \quad D_j^r := \nabla_{u_j} r_{ij}(x_i,u_j), \quad F_j^r := \nabla_{x_j} r_j(x_j), \tag{4.57a}$$

$$F_{ij} := \nabla_{x_i} f_{ij}(x_i,u_j), \quad D_j := \nabla_{u_j} f_{ij}(x_i,u_j), \quad \bar{F}_j := \nabla_{x_j} f_j(x_j). \tag{4.57b}$$

The respective numbers of rows and columns can be taken from Table 4.3. The gradient of the Lagrangian (4.53) with respect to the primal variables y reads

$$\nabla_y \mathcal{L} = \left((\nabla_{u_j}\mathcal{L}, \nabla_{x_j}\mathcal{L})\right)_{j\in V} = \left((\nabla_{u_j}\mathcal{L}_{ij}, \nabla_{x_j}\mathcal{L}_j + \sum_{k\in S(j)} \nabla_{x_j}\mathcal{L}_{jk})\right)_{j\in V} \tag{4.58}$$

with the three explicit expressions

$$\nabla_{x_i}\mathcal{L}_{ij} = \nabla_{x_i}\phi_{ij} - G_j^T\lambda_j - (F_{ij})^T\mu - \left(F_{ij}^r\right)^T\left(v_{lj}^u - v_{uj}^u\right), \tag{4.59a}$$

$$\nabla_{u_j}\mathcal{L}_{ij} = \nabla_{u_j}\phi_{ij} - E_j^T\lambda_j - (D_j)^T\mu - \left(D_j^r\right)^T\left(v_{lj}^u - v_{uj}^u\right), \tag{4.59b}$$

$$\nabla_{x_j}\mathcal{L}_j = \nabla_{x_j}\phi_j + \lambda_j - \left(\bar{F}_j\right)^T\mu - \left(F_j^r\right)^T\left(v_{lj}^x - v_{uj}^x\right). \tag{4.59c}$$

To state the Hessian of the Lagrangian (4.53) with respect to y, the following four matrix node subblocks are introduced:

$$H_{ij} := \nabla^2_{x_i x_i} \mathcal{L}_{ij}, \quad K_j := \nabla^2_{u_j u_j} \mathcal{L}_{ij}, \quad J_j := \nabla^2_{u_j x_i} \mathcal{L}_{ij}, \quad \bar{H}_j := \nabla^2_{x_j x_j} \mathcal{L}_j. \tag{4.60}$$

Omitting the variable dependencies, these four node subblocks are given explicitly by

$$H_{ij} = \nabla^2_{x_i x_i} \phi_{ij} - \sum_{\tau=1}^{m^g} \mu_\tau \nabla^2_{x_i x_i} f_{ij\,\tau} - \sum_{\tau=1}^{n_j^x} \lambda_{j\,\tau} \nabla^2_{x_i x_i} g_{j\,\tau} - \sum_{\tau=1}^{l_j^{ru}} \left(v^u_{lj\,\tau} - v^u_{uj\,\tau} \right) \nabla^2_{x_i x_i} r_{ij\,\tau}, \tag{4.61a}$$

$$K_j = \nabla^2_{u_j u_j} \phi_{ij} - \sum_{\tau=1}^{m^g} \mu_\tau \nabla^2_{u_j u_j} f_{ij\,\tau} - \sum_{\tau=1}^{n_j^x} \lambda_{j\,\tau} \nabla^2_{u_j u_j} g_{j\,\tau} - \sum_{\tau=1}^{l_j^{ru}} \left(v^u_{lj\,\tau} - v^u_{uj\,\tau} \right) \nabla^2_{u_j u_j} r_{ij\,\tau}, \tag{4.61b}$$

$$J_j = \nabla^2_{u_j x_i} \phi_{ij} - \sum_{\tau=1}^{m^g} \mu_\tau \nabla^2_{u_j x_i} f_{ij\,\tau} - \sum_{\tau=1}^{n_j^x} \lambda_{j\,\tau} \nabla^2_{u_j x_i} g_{j\,\tau} - \sum_{\tau=1}^{l_j^{ru}} \left(v^u_{lj\,\tau} - v^u_{uj\,\tau} \right) \nabla^2_{u_j x_i} r_{ij\,\tau}, \tag{4.61c}$$

$$\bar{H}_j = \nabla^2_{x_j x_j} \phi_j - \sum_{\tau=1}^{m^g} \mu_\tau \nabla^2_{x_j x_j} f_{j\,\tau} - \sum_{\tau=1}^{l_j^{rx}} \left(v^x_{lj\,\tau} - v^x_{uj\,\tau} \right) \nabla^2_{x_j x_j} r_{j\,\tau}. \tag{4.61d}$$

The corresponding dimensions are listed in Table 4.3.

Tree-Structured Block Matrices

In the incoming control case, most of the KKT matrix blocks feature stochastic staircase structures. The Hessian $\nabla^2_{yy} \mathcal{L}$ even takes a symmetric form of this structure. It consists of node subblocks K_j and J_j as defined in (4.61) as well as the accumulations

$$H_j = \bar{H}_j + \sum_{k \in S(j)} H_{jk}, \quad j \in V. \tag{4.62}$$

The symmetric node subblocks K_j and H_j are located on the diagonal, the node subblocks J_j are located on the secondary diagonals corresponding to the predecessor node i. For the tree

in Fig. 4.1, the Hessian of the Lagrangian (4.53) reads

$$\nabla^2_{yy}\mathcal{L} = \begin{bmatrix}
K_0 & & & & & & & & & & \\
 & H_0 & J_1^T & & J_2^T & & & & & & \\
 & J_1 & K_1 & & & & & & & & \\
 & & & H_1 & & & J_3^T & & J_4^T & & \\
 & J_2 & & & K_2 & & & & & & \\
 & & & & & H_2 & & & & J_5^T & \\
 & & & J_3 & & & K_3 & & & & \\
 & & & & & & & H_3 & & & \\
 & & & J_4 & & & & & K_4 & & \\
 & & & & & & & & & H_4 & \\
 & & & & & J_5 & & & & K_5 & \\
 & & & & & & & & & & H_5
\end{bmatrix}. \tag{4.63}$$

Using the node subblock notation (4.57a), the Jacobian $\nabla c_{\mathcal{R}}$ of the range constraints (4.14) has node subblocks $[D_j^r \; F_j^r]$ on the diagonal and $[0 \; F_{ij}^r]$ on the secondary diagonals. For the tree in Fig. 4.1, the tree-sparse Jacobian $\nabla c_{\mathcal{R}}$ reads

$$\nabla c_{\mathcal{R}} = \begin{bmatrix}
D_0^r & & & & & & & & & & \\
 & F_0^r & & & & & & & & & \\
 & F_{01}^r & D_1^r & & & & & & & & \\
 & & & F_1^r & & & & & & & \\
 & F_{02}^r & & & D_2^r & & & & & & \\
 & & & & & F_2^r & & & & & \\
 & & & F_{13}^r & & & D_3^r & & & & \\
 & & & & & & & F_3^r & & & \\
 & & & F_{14}^r & & & & & D_4^r & & \\
 & & & & & & & & & F_4^r & \\
 & & & & & F_{25}^r & & & & D_5^r & \\
 & & & & & & & & & & F_5^r
\end{bmatrix}. \tag{4.64}$$

The dynamic block G of the equality Jacobian $\nabla c_{\mathcal{E}}^T = [G^T \; F^T]$ features the same tree structure as the range block (4.64). The diagonal node subblocks have the form $[E_j \; -I]$ and the node subblocks on the secondary diagonals are $[0 \; G_j]$. The global constraint block F is a block-dense row with node subblocks $[D_j \; F_j]$. For the tree in Fig. 4.1, the blocks G and F of the tree-sparse

Jacobian $\nabla c_{\mathcal{E}}$ read

$$G = \begin{bmatrix} E_0 & -I & & & & & & & & & & \\ & G_1 & E_1 & -I & & & & & & & & \\ & G_2 & & & E_2 & -I & & & & & & \\ & & & G_3 & & & E_3 & -I & & & & \\ & & & G_4 & & & & & E_4 & -I & & \\ & & & & & G_5 & & & & & E_5 & -I \end{bmatrix}, \tag{4.65}$$

and

$$F = \begin{bmatrix} D_0 & F_0 & D_1 & F_1 & D_2 & F_2 & D_3 & F_3 & D_4 & F_4 & D_5 & F_5 \end{bmatrix}. \tag{4.66}$$

4.3. Tree-Sparse KKT Algorithms

The step computation in IPMs is based on solving a series of KKT systems (cf. Sect. 2.2.1). In this thesis, direct approaches for the KKT systems are considered, i.e. the solutions are determined by factorizing the KKT matrix first and then solving two linear systems with the resulting factors in respective substitutions. Solving KKT systems by a direct approach is dominated by the complexity of the factorization, which is of cubic order with respect to the problem dimensions when neglecting any sparsities. Hence, exploiting problem-specific sparsity patterns in the KKT matrix is mandatory to improve the efficiency of the solution procedure.

The KKT solution algorithms for the tree-sparse problems exploit the underlying tree topology, which manifests itself into the structures of the KKT matrix blocks (cf. Sect. 4.2). Benefiting from the node-wise presentations of TSPs and the resulting KKT data, these *tree-sparse KKT algorithms* are designed as *traversals* of tree nodes that fulfill their overall tasks by performing a series of *node operations*. This way, the KKT solution is evaluated with linear complexity [83]. Moreover, the node-wise presentations of tree-sparse problems and algorithms are conducive to distributing them (cf. Chap. 5).

The tree-sparse KKT solution procedure is divided into *phases* and *stages*. The stages refer to the three steps of the direct approach: the *factorization* stage, the *inward substitution* and the *outward substitution*. Each of these stages is composed of two phases. First, by eliminating the duals corresponding to the range constraints in the *elimination phase*, the KKT system is transformed into so-called basic form, which corresponds to the KKT system of a TSP only consisting of the objective, the dynamics and the global constraints. This basic KKT system is then solved in the second phase of the KKT solution procedure referred to as *basic recursion*.

Extending the tree-sparse KKT algorithms by inertia corrections in the subsequent section, the basic recursions are presented here for the sake of completeness. Detailed presentations of the elimination phases for both control cases are—to the best of the author's knowledge—not provided in the literature and, therefore, supplemented here.

In the following, Section 4.3.1 provides a global point of view on the tree-sparse KKT solution procedure. That section covers the reduction of the primal-dual system to the KKT system and, moreover, provides refined regularity assumptions for the reduced KKT system (cf. Sect. 2.2.2). In the subsequent sections, the tree-sparse KKT algorithms are presented for the outgoing control case and the incoming control case, respectively.

4.3.1. Global Point of View

The KKT system $\Omega w = -\omega$ that is solved by means of the tree-sparse KKT algorithms reads

$$\begin{bmatrix} H + \Phi & \nabla c_{\mathcal{E}}^T & \nabla c_{\mathcal{R}}^T \\ \nabla c_{\mathcal{E}} & & \\ \nabla c_{\mathcal{R}} & & -\Psi^{-1} \end{bmatrix} \begin{pmatrix} \Delta y \\ -\Delta z \\ -\Delta v \end{pmatrix} = - \begin{pmatrix} \nu \\ \alpha \\ \rho \end{pmatrix}, \tag{4.67}$$

where α represents the vector of infeasibilities of the equality constraints (4.29a), and the dual vector v is the difference $v_l - v_u$ of the duals corresponding to the lower and upper ranges (4.23c) and (4.23d), respectively. The KKT system (4.67) is obtained from the primal-dual system (4.32) by eliminating the bound slacks s_l, s_u and the range slacks t_l, t_u. The concentrated matrix barrier terms Φ and Ψ are given by

$$\Phi := \Phi_l + \Phi_u \quad \text{and} \quad \Psi := \Psi_l + \Psi_u. \tag{4.68}$$

The right-hand side barrier terms of the bounds are concentrated into

$$\nu := \nabla_y \mathcal{L} + \Phi_l \beta_l + \phi_l - \Phi_u \beta_u - \phi_u, \tag{4.69}$$

and the right-hand side barrier terms of the ranges read

$$\rho := \rho_l + \Psi^{-1} \left(\psi_l - \Psi_u \left(\rho_l + \rho_u \right) + \psi_u \right). \tag{4.70}$$

Detailed listings of the elimination steps are given in [72].

In the first phase of the presented KKT algorithms, the range duals v in (4.67) are eliminated by using

$$\Delta v = -\Psi \left(\rho + \nabla c_{\mathcal{R}} \Delta y\right). \tag{4.71}$$

The reduced KKT system $\Omega_{\mathrm{r}} w_r = -\omega_{\mathrm{r}}$ without range duals reads

$$\begin{bmatrix} \hat{H} & \nabla c_{\mathcal{E}}^T \\ \nabla c_{\mathcal{E}} & \end{bmatrix} \begin{pmatrix} \Delta y \\ -\Delta z \end{pmatrix} = - \begin{pmatrix} \hat{\nu} \\ \alpha \end{pmatrix}, \tag{4.72}$$

where the concentrated data are given by

$$\hat{H} := H + \Phi + \nabla c_{\mathcal{R}}^T \Psi \nabla c_{\mathcal{R}} \quad \text{and} \quad \hat{\nu} := \nu + \nabla c_{\mathcal{R}}^T \Psi \rho. \tag{4.73}$$

In the second phase of the KKT algorithms, which is referred to as basic recursion, the KKT system (4.72) is projected onto the null-space of the dynamics, and then the Schur-Complement method is applied to the projected system. For this, the dynamic part G of the Jacobian $\nabla c_{\mathcal{E}}$ is required to have full rank, i.e. $\operatorname{rank}(G) = n^x$, which is, however, already ensured by the explicit formulation of the dynamics. To apply the Schur-Complement method, the global equality part F of the Jacobian $\nabla c_{\mathcal{E}}$ is required to have full rank, and, additionally, the projection of the Hessian $\hat{H}$ must be positive definite. These requirements for the tree-sparse KKT solution procedure are subsumed into the following assumptions.

Assumption 2 (Regularity Assumptions for the Reduced KKT System)**.** *Consider the system matrix of* (4.72) *with* $\hat{H}$ *as defined in* (4.73)*. Let* $G \in \mathbb{R}^{n^x \times n^v}$ *be the dynamic block and* $F \in \mathbb{R}^{m^g \times n^v}$ *the global constraint block of the tree-sparse Jacobian* $\nabla c_{\mathcal{E}}$*. The following conditions apply:*

(A1) F *has full row rank on the null-space of* G*, i.e.* $\operatorname{rank}(F_{|\mathcal{N}(G)}) = m^g$*.*

(A2) $\hat{H}$ *is positive definite on the null-space of* $\nabla c_{\mathcal{E}}$*, i.e.* $\hat{H}_{|\mathcal{N}(G)\cap\mathcal{N}(F)} > 0$*.*

Note that As. 2 is a refined version of As. 1 introduced to guarantee descent directions in the IPM framework (cf. Sect. 2.2.2).

4.3.2. Solution of the Tree-Sparse KKT System - Outgoing Control

The tree-sparse KKT algorithms are established in two steps. First, the KKT system (4.67) is restated in a node-wise presentation, i.e. as a set of node-wise defined conditions as well as an additional equation representing the linearized global constraints. Afterwards, the elimination of the range duals as presented in Sect. 4.3.1 is tailored to the TSP in the considered control form. The presentation of the basic recursions first covers the dual feasibility conditions and the dynamics. Treating the global constraint equation is supplemented afterwards. The order of evaluating the primal and dual node variables in the outward substitution is discussed separately. Dynamic algorithm tables are introduced for each phase of the tree-sparse solution procedure. Their reading is discussed at the end of this section.

Node-Wise Presentation of the KKT System

Using the right-hand side notation

$$\nu = \left(\left(f_j, d_j\right)\right)_{j\in V}, \quad \alpha = \left(\left(h_j\right)_{j\in V}, e_V\right), \quad \rho = \left(r_j\right)_{j\in V}, \tag{4.74}$$

and dropping Δ from all step variables, the node-wise presentation of KKT system (4.67) reads

$$\left(H_j + \Phi_j^x\right) x_j + J_j^T u_j + \lambda_j - \sum_{k\in S(j)} G_k^T \lambda_k - \left(F_j^r\right)^T v_j - F_j^T \mu + f_j = 0, \quad j \in V, \tag{4.75a}$$

$$J_j x_j + \left(K_j + \Phi_j^u\right) u_j - \sum_{k\in S(j)} E_k^T \lambda_k - \left(D_j^r\right)^T v_j - D_j^T \mu + d_j = 0, \quad j \in V, \tag{4.75b}$$

$$G_j x_i + E_j u_i - x_j + h_j = 0, \quad j \in V, \tag{4.75c}$$

$$F_j^r x_j + D_j^r u_j + \Psi_j^{-1} v_j + r_j = 0, \quad j \in V, \tag{4.75d}$$

$$\sum_{j\in V} F_j x_j + \sum_{j\in V} D_j u_j + e_V = 0. \tag{4.75e}$$

Elimination of the Dual Ranges

The dual range variables are eliminated from (4.75) by solving (4.75d) for v_j, i.e.

$$-v_j = \Psi_j \left(F_j^r x_j + D_j^r u_j + r_j\right). \tag{4.76}$$

Table 4.4.: Elimination of range duals – Outgoing control case

	Factorization ↓	Inward Subst. ↓	Outward Subst. ↑
1:			$-v_j \leftarrow \Psi_j(-v_j)$
2:	$H_j \mathrel{+}= {F_j^r}^T \Psi_j F_j^r + \Phi_j^x$	$f_j \mathrel{+}= {F_j^r}^T \Psi_j r_j$	$-v_j \mathrel{+}= F_j^r x_j$
3:	$K_j \mathrel{+}= {D_j^r}^T \Psi_j D_j^r + \Phi_j^u$	$d_j \mathrel{+}= {D_j^r}^T \Psi_j r_j$	$-v_j \mathrel{+}= D_j^r u_j$
4:	$J_j \mathrel{+}= {D_j^r}^T \Psi_j F_j^r$		

Using the node subblock notation

$$\hat{H}_j := H_j + \Phi_j^x + \left(F_j^r\right)^T \Psi_j F_j^r, \qquad \hat{f}_j := f_j + \left(F_j^r\right)^T \Psi_j r_j, \tag{4.77a}$$

$$\hat{K}_j := K_j + \Phi_j^u + \left(D_j^r\right)^T \Psi_j D_j^r, \qquad \hat{d}_j := d_j + \left(D_j^r\right)^T \Psi_j r_j, \tag{4.77b}$$

$$\hat{J}_j := J_j \qquad + \left(F_j^r\right)^T \Psi_j D_j^r, \tag{4.77c}$$

and substituting (4.76) into (4.75a) and (4.75b), the latter equations read

$$\hat{H}_j x_j + \hat{J}_j^T u_j + \lambda_j - \sum_{k \in S(j)} G_k^T \lambda_k - F_j^T \mu + \hat{f}_j = 0, \quad j \in V, \tag{4.78a}$$

$$\hat{J}_j x_j + \hat{K}_j u_j - \sum_{k \in S(j)} E_k^T \lambda_k - D_j^T \mu + \hat{d}_j = 0, \quad j \in V. \tag{4.78b}$$

The steps for the described elimination process are listed in Table 4.4, the reading of this table is described at the end of this section.

Basic Recursion

The presentation of the basic recursion in the outgoing control case extends the presentation in [77] from the deterministic case on chains to the stochastic case on trees. First, consider equations (4.78) and (4.75c) at a leaf $j \in L$. Dropping all previous used mathematical accents, these equations read

$$H_j x_j + J_j^T u_j + \lambda_j - F_j^T \mu + f_j = 0, \tag{4.79a}$$

$$J_j x_j + K_j u_j - D_j^T \mu + d_j = 0, \tag{4.79b}$$

$$G_j x_i + E_j u_i - x_j + h_j = 0. \tag{4.79c}$$

In the first step of the basic recursion, equation (4.79b) is solved for the controls u_j, i.e.

$$u_j = -K_j^{-1} \left(J_j x_j + D_j^T (-\mu) + d_j\right). \tag{4.80}$$

Assumption 2 implies that the symmetric node subblock K_j is positive definite. Hence, its inverse exists and the Cholesky factorization can be applied, i.e.

$$K_j = L_j L_j^T. \tag{4.81}$$

Using (4.80) to eliminate u_j from (4.79a) leads to

$$-\lambda_j = \bar{H}_j x_j + \bar{F}_j^T(-\mu) + \bar{f}_j, \tag{4.82}$$

where the modified matrix and vector node subblocks are defined as

$$\bar{J}_j := L_j^{-1} J_j, \qquad \bar{D}_j := L_j^{-1} D_j, \qquad \bar{d}_j := L_j^{-1} d_j, \tag{4.83a}$$

$$\bar{H}_j := H_j - \bar{J}_j^T \bar{J}_j, \qquad \bar{F}_j := F_j - \bar{D}_j \bar{J}_j, \qquad \bar{f}_j := f_j + \bar{J}_j^T \bar{d}_j. \tag{4.83b}$$

With the modified node subblocks (4.83), equation (4.80) reads

$$u_j = -L_j^{-T} \left(\bar{J}_j x_j + \bar{D}_j^T (-\mu) + \bar{d}_j \right). \tag{4.84}$$

Solving the linearized dynamics (4.79c) for the local states x_j,

$$x_j = G_j x_i + E_j u_i + h_j, \tag{4.85}$$

the resulting expression is substituted into (4.82), which then reads

$$-\lambda_j = \bar{H}_j G_j x_i + \bar{H}_j E_j u_i + \bar{F}_j(-\mu) + \bar{f}_j + \bar{H}_j h_j. \tag{4.86}$$

In (4.86), the local dynamic duals λ_j depend only on state and control variables of the predecessor $i = \pi(j)$ as well as the dual multiplier μ for the global constraints (4.3f).

Now, considering (4.78) for a node $j \in V$ that has only leaves as successors ($S(j) \subseteq L$), using (4.86) to eliminate the successor's dynamic duals λ_k and dropping the mathematical accents used during the elimination of the range duals, equations (4.86) transform into

$$\tilde{H}_j x_j + \tilde{J}_j^T u_j + \lambda_j - \tilde{F}_j^T \mu + \tilde{f}_j = 0, \tag{4.87a}$$

$$\tilde{J}_j x_j + \tilde{K}_j u_j - \tilde{D}_j^T \mu + \tilde{d}_j = 0, \tag{4.87b}$$

where the modified matrix and vector node subblocks are given by

$$\tilde{H}_j := H_j + \sum_{k\in S(j)} G_k^T \bar{H}_k G_k, \qquad \tilde{f}_j := f_j + \sum_{k\in S(j)} G_k^T \left(\bar{f}_k + \bar{H}_k h_k\right), \tag{4.88a}$$

$$\tilde{K}_j := K_j + \sum_{k\in S(j)} E_k^T \bar{H}_k E_k, \qquad \tilde{d}_j := d_j + \sum_{k\in S(j)} E_k^T \left(\bar{f}_k + \bar{H}_k h_k\right), \tag{4.88b}$$

$$\tilde{J}_j := J_j + \sum_{k\in S(j)} E_k^T \bar{H}_k G_k, \tag{4.88c}$$

$$\tilde{F}_j := F_j + \sum_{k\in S(j)} \bar{F}_k G_k, \qquad \tilde{D}_j := D_j + \sum_{k\in S(j)} \bar{F}_k E_k. \tag{4.88d}$$

In (4.87), all variables corresponding to the successors $S(j)$ are eliminated. These equations read the same as the former leaf equations (4.79a) and (4.79b). The previous elimination steps are now applied to (4.87) and (4.79c) for the predecessor i. Proceeding this way in an inward sweep over the tree, variables u_j, x_j and λ_j are eliminated recursively from the linearized dual feasibility conditions (4.75a) and (4.75b) as well as the dynamics (4.75c) of the tree-sparse KKT system.

In the presence of global equality constraints (4.3f), the corresponding linearization (4.75e) is rewritten as

$$\sum_{j\in V_{T-1}} F_j x_j + \sum_{j\in V_{T-1}} D_j u_j + \sum_{j\in L_T} F_j x_j + \sum_{j\in L_T} D_j u_j + X_{V_T}\mu + e_{V_T} = 0 \tag{4.89}$$

with a fill-in matrix $X_{V_T} := 0$. Using (4.84) to eliminate u_j corresponding to the nodes $j \in L_T$, equation (4.89) reads

$$\sum_{j\in V_{T-1}} F_j x_j + \sum_{j\in V_{T-1}} D_j u_j + \sum_{j\in L_T} \bar{F}_j x_j + X_{V_{T-1}}\mu + \bar{e}_{V_T} = 0. \tag{4.90}$$

The modified node subblocks are defined as

$$X_{V_{T-1}} := X_{V_T} + \sum_{j\in L_T} \bar{D}_j \bar{D}_j^T \quad \text{and} \quad \bar{e}_{V_{T-1}} := e_{V_T} - \sum_{j\in L_T} \bar{D}_j \bar{d}_j. \tag{4.91}$$

Next, expression (4.85) is used to eliminate x_j for $j \in L_T$ from (4.90). Defining

$$e_{V_{T-1}} := \bar{e}_{V_{T-1}} + \sum_{j\in L_T} \bar{F}_j h_j, \tag{4.92}$$

Table 4.5.: Basic KKT recursion – Outgoing control case

	Factorization ↓	Inward Subst. ↓	Outward Subst. ↑
1:	$K_j \leftarrow L_j L_j^T$		
2:	$D_j \leftarrow L_j^{-1} D_j$	$d_j \leftarrow L_j^{-1} d_j$	$u_j \leftarrow -L_j^{-T} u_j$
3:	$J_j \leftarrow L_j^{-1} J_j$		$u_j \mathrel{+}= J_j x_j$
4:	$H_j \mathrel{-}= J_j^T J_j$	$f_j \mathrel{+}= J_j^T d_j$	$-\lambda_j \mathrel{+}= H_j x_j$
5:	$F_j \mathrel{-}= D_j J_j$		
6:	$X \mathrel{+}= D_j D_j^T$	$e \mathrel{-}= D_j d_j$	
7:	$H_i \mathrel{+}= G_j^T H_j G_j$	$f_i \mathrel{+}= G_j^T (f_j + H_j h_j)$	$x_j \mathrel{+}= G_j x_i$
8:	$K_i \mathrel{+}= E_j^T H_j E_j$	$d_i \mathrel{+}= E_j^T (f_j + H_j h_j)$	$x_j \mathrel{+}= E_j u_i$
9:	$J_i \mathrel{+}= E_j^T H_j G_j$		
10:	$F_i \mathrel{+}= F_j G_j$	$e \mathrel{+}= F_j h_j$	$-\lambda_j \mathrel{+}= F_j^T (-\mu)$
11:	$D_i \mathrel{+}= F_j E_j$	$h_j \leftrightarrow f_j$	$u_j \mathrel{+}= D_j^T (-\mu)$
12:	$X \leftarrow LL^T$	$e \leftarrow L^{-1} e$	$-\mu \leftarrow L^{-T} (-\mu)$

the resulting expression reads

$$\sum_{j \in V_{T-2}} F_j x_j + \sum_{j \in V_{T-2}} D_j u_j + \sum_{j \in L_{T-1}} \tilde{F}_j x_j + \sum_{j \in L_{T-1}} \tilde{D}_j u_j + X_{V_{T-1}} \mu + e_{V_{T-1}} = 0. \tag{4.93}$$

Equation (4.93) has the same form as (4.89) but with level T eliminated. Repeating the elimination of u_j and x_j from (4.93) inwardly to the root, the global constraints reduce to

$$X_\emptyset(-\mu) = e_\emptyset, \tag{4.94}$$

where $X_\emptyset$ is as sum of symmetric products $\bar{D}_j \bar{D}_j^T$ also symmetric itself and at least positive semidefinite. With As. 2 satisfied, $X_\emptyset$ is also positive definite and the Cholesky factorization is applied, i.e.

$$X_\emptyset = L_\emptyset L_\emptyset^T. \tag{4.95}$$

The steps of the basic recursion are listed in Table 4.5.

Outward Substitution

The subblock modifications (4.77), (4.83), (4.88) and (4.95) described above correspond to the factorization and the inward substitution stages of the tree-sparse KKT algorithms. Those steps can be performed during one single inward recursion over the tree or in two separate

recursions. In the latter case, the matrix subblock modifications are applied in the factorization stage and the vector subblock modifications in the inward substitution stage.

The variables of the KKT system (4.67) are computed in the outward substitution stage of the tree-sparse KKT algorithm, and that node-wise in an outward sweep over the tree from its root 0 to its leaves L. The global multiplier μ is obtained from (4.94) using the Cholesky factors (4.95). The node variables x_j, u_j and λ_j are computed in this very order using expressions (4.85), (4.84) and (4.82), respectively. The range duals v_j are evaluated afterwards using (4.76).

Dynamic Algorithm Tables

In the introduced dynamic algorithm tables for the tree-sparse KKT algorithms, i.e. tables 4.5 and 4.4 in this section and tables 4.7 and 4.6 in Sect. 4.3.3, the mathematical accents used in the discussions are omitted. Instead, these tables make use of the following dynamic operations:

+= Add the right-hand side result to the left-hand side value.

-= Subtract the right-hand side result from the left-hand side value.

$\leftarrow$ Overwrite the left-hand side value with the right-hand side result.

$\leftrightarrow$ Swap values of the left-hand side and the right-hand side.

In Chap. 5, the operations and data in the algorithm tables are described and classified in more detail. For now, the following terminology is provided beforehand to explain the reading of the specific KKT algorithm tables. First, the involved data are subsumed in sets DA(j) for each node $j \in V$. There are common *node data* labeled with a node subscript (e.g. K_j, d_j) that exist for each node and can be found in each set DA(j). There are also specific *global data* that exist only once and are attributed to the set DA(0) of the tree root 0. In algorithm tables, global data have no subscripts, e.g. $X_\emptyset$ and $e_\emptyset$ read X and e in Table 4.5. Second, the respective operations of each stage are subsumed in sets OP(j) for each node j. Common *node operations* are performed for each node in the tree, i.e. these are listed in each set OP(j). So-called *global operations* are performed only once and are supplemented to the set OP(0). Global are those operations that involve only global data, e.g. item 12 in Table 4.5.

Now, the KKT algorithm tables read as follows. For each algorithm, the operations in a set OP(j_j) are performed one after another without performing operations of another node $j_2 \neq j_1$ in between. In the factorization stage and the inward substitution, the operations in one set OP(j) are performed from top to bottom as listed in the table ($\downarrow$). Item 12 of the basic recursion in Table 4.5 is as global operation only applied at the root 0. The nodes of the tree are processed in an inward sweep, meaning that the operations OP(j) for a node j are executed only after those of all its successors $k \in S(j)$ were performed. The outward

substitution proceeds completely opposite to the other two stages. The nodes are processed in an outward sweep, i.e. the operations OP(j) are executed not until completing OP(i) of the predecessor i, and the operations in one set are performed from bottom to top as listed in the respective column of the table (↑).

In the implementation, the respective stages of the two phases of the tree-sparse KKT algorithms are performed in one single traversal over the tree. In doing so, the items of the tables for the range eliminations are placed on top of the items of the respective basic recursions, e.g. items 1 to 4 in Table 4.4 are placed on top of item 1 of Table 4.5. Alternatively, the two phases can be performed one after another in two separate traversals. Doing the latter and in the absence of range constraints, one needs to account for the remaining barrier diagonal terms within the basic recursion, i.e. in the outgoing control case one needs to add Φ_j^x and Φ_j^u to H_j and K_j, respectively.

4.3.3. Solution of the Tree-Sparse KKT System - Incoming Control

In the incoming control case, the tree-sparse KKT algorithms are established analogously to the outgoing control case. The presentation here follows the guideline at the beginning of Sect. 4.3.2. The reading of the subsequently introduced dynamic algorithm tables is already explained at the end of Sect. 4.3.2.

Node-wise Presentation of the KKT System

The right-hand side of the primal-dual system (4.32) is divided into

$$\nu := ((d_j, f_j))_{j\in V}\,, \quad \alpha := \Big((h_j)_{j\in V}\,, e_V\Big)\,, \quad \rho := \big((r_j^u, r_j^x)\big)_{j\in V}\,. \tag{4.96}$$

Dropping Δ from all step variables, the node-wise presentation of KKT system (4.67) reads

$$J_j x_i + \left(K_j + \Phi_j^u\right) u_j - E_j^T \lambda_j - \left(D_j^r\right)^T v_j^u - D_j^T \mu + d_j = 0, \quad j \in V, \tag{4.97a}$$

$$\begin{aligned}\left(H_j + \Phi_j^x\right) x_j + \sum_{k\in S(j)} J_j^T u_k + \lambda_j - \sum_{k\in S(j)} G_j^T \lambda_k \\ - \sum_{k\in S(j)} \left(F_{ij}^r\right)^T v_k^u - \left(F_j^r\right)^T v_j^x - F_j^T \mu + f_j = 0, \quad j \in V,\end{aligned} \tag{4.97b}$$

$$G_j x_i + E_j u_j - x_j + h_j = 0, \quad j \in V, \tag{4.97c}$$

$$F_{ij}^r x_i + D_j^r u_j + \left(\Psi_j^u\right)^{-1} v_j^u + r_j^u = 0, \quad j \in V, \tag{4.97d}$$

$$F_j^r x_j + \left(\Psi_j^x\right)^{-1} v_j^x + r_j^x = 0, \quad j \in V, \tag{4.97e}$$

$$\sum_{j\in V} D_j u_j + \sum_{j\in V} F_j x_j + e_V = 0. \tag{4.97f}$$

Elimination of the Dual Ranges

Solving (4.97d) for v_j^u as well as (4.97e) for v_j^x lead to

$$-v_j^u = \Psi_j^u \left(F_{ij}^r x_i + D_j^r u_j + r_j^u\right), \tag{4.98a}$$

$$-v_j^x = \Psi_j^x \left(F_j^r x_j + r_j^x\right). \tag{4.98b}$$

Equations (4.97a) and (4.97b) in reduced forms then read

$$\hat{J}_j x_i + \hat{K}_j u_j - E_j^T \lambda_j - D_j^T \mu + \hat{d}_j = 0, \quad j \in V, \tag{4.99a}$$

$$\hat{H}_j x_j + \sum_{k\in S(j)} \hat{J}_k^T u_k + \lambda_j - \sum_{k\in S(j)} G_k^T \lambda_k - F_j^T \mu + \hat{f}_j = 0, \quad j \in V, \tag{4.99b}$$

Table 4.6.: Elimination of range duals – Incoming control case

	Factorization ↓	Inward Subst. ↓	Outward Subst. ↑
1:			$-v_j^u \leftarrow \Psi_j^u(-v_j^u)$
2:			$-v_j^x \leftarrow \Psi_j^x(-v_j^x)$
3:	$K_j \mathrel{+}= D_j^{rT}\Psi_j^u D_j^r + \Phi_j^u$	$d_j \mathrel{+}= D_j^{rT}\Psi_j^u r_j^u$	$-v_j^u \mathrel{+}= D_j^r u_j$
4:	$J_j \mathrel{+}= D_j^{rT}\Psi_j^u F_{ij}^r$		
5:	$H_i \mathrel{+}= F_{ij}^{rT}\Psi_j^u F_{ij}^r$	$f_i \mathrel{+}= F_{ij}^{rT}\Psi_j^u r_j^u$	$-v_j^u \mathrel{+}= F_{ij}^r x_i$
6:	$H_j \mathrel{+}= F_j^{rT}\Psi_j^x F_j^r + \Phi_j^x$	$f_j \mathrel{+}= F_j^{rT}\Psi_j^x r_j^x$	$-v_j^x \mathrel{+}= F_j^r x_j$

where the modified node subblocks are given by

$$\hat{K}_j := K_j + \left(D_j^r\right)^T \Psi_j^u D_j^r, \qquad \hat{d}_j := d_j + \left(D_j^r\right)^T \Psi_j^u r_j^u \tag{4.100a}$$

$$\hat{J}_j := J_j + \left(D_j^r\right)^T \Psi_j^u F_{ij}^r, \tag{4.100b}$$

and

$$\hat{H}_j := H_j + \left(F_j^r\right)^T \Psi_j^x F_j^r + \sum_{k\in S(j)} \left(F_{jk}^r\right)^T \Psi_j^u F_{jk}^r, \tag{4.101a}$$

$$\hat{f}_j := f_j + \left(F_j^r\right)^T \Psi_j^x r_j^x + \sum_{k\in S(j)} \left(F_{jk}^r\right)^T \Psi_j^u r_k^x. \tag{4.101b}$$

The computation steps presented above are listed in Table 4.6, the reading of this table is described at the end of Sect. 4.3.2.

Basic Recursion

The presentation of the basic recursion in the incoming control case follows the lines of Steinbach [79]. At a leaf $j \in L$, the equations (4.99) and (4.97c) read

$$J_j x_i + K_j u_j + E_j^T(-\lambda_j) + D_j^T(-\mu) + d_j = 0, \tag{4.102a}$$

$$H_j x_j + \lambda_j + F_j^T(-\mu) + f_j = 0, \tag{4.102b}$$

$$G_j x_i + E_j u_j - x_j + h_j = 0. \tag{4.102c}$$

For the sake of clarity, the mathematical accents used during the first elimination process are dropped. Equations (4.102b) and (4.102c) are solved for λ_j and x_j, respectively, then reading

$$-\lambda_j = H_j x_j + F_j^T(-\mu) + f_j, \tag{4.103a}$$

$$x_j = G_j x_i + E_j u_j + h_j. \tag{4.103b}$$

Substituting (4.103b) into (4.103a) leads to

$$-\lambda_j = H_j G_j x_i + H_j E_j u_j + H_j h_j + F_j^T(-\mu) + f_j. \tag{4.104}$$

Using (4.104) to eliminate λ_j from (4.102a), the latter equation reads

$$\bar{J}_j x_i + \bar{K}_j u_j + \bar{D}_j^T(-\mu) + \bar{d}_j = 0, \tag{4.105}$$

where the modified node subblocks are defined as

$$\bar{K}_j := K_j + E_j^T H_j E_j, \qquad \bar{d}_j := d_j + E_j^T \left(H_j h_j + f_j\right), \tag{4.106a}$$

$$\bar{J}_j := J_j + E_j^T H_j G_j, \qquad \bar{D}_j := D_j + F_j E_j. \tag{4.106b}$$

Due to As. 2, the node subblocks $\bar{K}_j$ are symmetric and positive definite and the Cholesky factorization $\bar{K}_j = L_j L_j^T$ is applied. In doing so and solving (4.105) for the local controls u_j, expression (4.105) is further transformed into

$$u_j = -L_j^{-T}\left(\check{J}_j x_i + \check{D}_j^T(-\mu) + \check{d}_j\right), \tag{4.107}$$

where the merged KKT node subblocks read

$$\check{J}_j := L_j^{-1}\bar{J}_j, \quad \check{D}_j := \bar{D}_j L_j^{-T}, \quad \check{d}_j := L_j^{-1}\bar{d}_j. \tag{4.108}$$

In (4.107), the local control variables u_j depend only on the state variables x_i of the predecessor i as well as the global multiplier μ. Therefore, consider (4.99b) for a node j with $S(j) \in L$. Using (4.104) to eliminate the dynamic duals λ_k of the successors $k \in S(j)$ and dropping the accents used during the elimination of the range duals, equation (4.99b) reduces to

$$\bar{H}_j x_j + \sum_{k \in S(j)} \bar{J}_k^T u_k + \lambda_j + \bar{F}_j^T(-\mu) + \bar{f}_j = 0 \tag{4.109}$$

where the modified node subblocks are given by

$$\bar{H}_j := H_j + \sum_{k \in S(j)} G_k^T H_k G_k, \qquad \bar{f}_j := f_j + \sum_{k \in S(j)} G_k^T \left(H_k h_k + f_k\right), \tag{4.110a}$$

$$\bar{F}_j := F_j + \sum_{k \in S(j)} F_k G_k. \tag{4.110b}$$

The controls u_k of the successors $k \in S(j)$ are eliminated using (4.107). Defining

$$\tilde{H}_j := \bar{H}_j - \sum_{k \in S(j)} \check{J}_k^T \check{J}_k, \qquad \tilde{f}_j := \bar{f}_j - \sum_{k \in S(j)} \check{J}_k^T \check{d}_k, \tag{4.111a}$$

$$\tilde{F}_j := \bar{F}_j - \sum_{k \in S(j)} \check{D}_k \check{J}_k, \tag{4.111b}$$

the modified equation (4.109) reads

$$\tilde{H}_j x_j + \lambda_j + \tilde{F}_j^T(-\mu) + \tilde{f}_j = 0. \tag{4.112}$$

With the last step the nodes $S(j)$ are eliminated from the tree. The equations (4.99a), (4.112) and (4.97c) now have the same form as the former leaf equations (4.102). Therefore, the elimination process can be repeated in an inward recursion over the tree.

For computing the global multiplier μ, the global equations (4.97f) are restated as

$$\sum_{j \in V_{T-1}} D_j u_j + \sum_{j \in V_{T-1}} F_j x_j + \sum_{j \in L_T} D_j u_j + \sum_{j \in L_T} F_j x_j + X_{V_T} \mu + e_{V_T} = 0 \tag{4.113}$$

with the fill-in block $X_{V_T} := 0$. Using expression (4.103b) to eliminate x_j for $j \in L_T$ leads to

$$\sum_{j \in V_{T-2}} F_j x_j + \sum_{j \in V_{T-1}} D_j u_j + \sum_{j \in L_{T-1}} \bar{F}_j x_j + \sum_{j \in L_T} \bar{D}_j u_j + X_{V_T} \mu + \bar{e}_{V_T} = 0, \tag{4.114}$$

where the modified right-hand side subblock is defined as

$$\bar{e}_{V_T} := e_{V_T} + \sum_{j \in L_T} F_j h_j. \tag{4.115}$$

Next, equation (4.107) is used to eliminate u_j of the same nodes $j \in L_T$. Defining

$$X_{V_{T-1}} := X_{V_T} + \sum_{j \in L_T} \check{D}_j \check{D}_j^T \quad \text{and} \quad e_{V_{T-1}} := \bar{e}_{V_T} - \sum_{j \in L_T} \check{D}_j \check{d}_j, \tag{4.116}$$

the global constraints then read

$$\sum_{j \in V_{T-2}} D_j u_j + \sum_{j \in V_{T-2}} F_j x_j + \sum_{j \in L_{T-1}} \bar{D}_j u_j + \sum_{j \in V_{T-1}} \tilde{F}_j x_j + X_{V_{T-1}} \mu + e_{V_{T-1}} = 0. \tag{4.117}$$

Table 4.7.: Basic KKT recursion – Incoming control case

	Factorization ↓	Inward Subst. ↓	Outward Subst. ↑
1:	$K_j \mathrel{+}= E_j^T H_j E_j$	$d_j \mathrel{+}= E_j^T(H_j h_j + f_j)$	$-\lambda_j \mathrel{+}= H_j x_j$
2:	$K_j \leftarrow L_j L_j^T$		
3:	$J_j \mathrel{+}= E_j^T H_j G_j$		$x_j \mathrel{+}= E_j u_j$
4:	$H_i \mathrel{+}= G_j^T H_j G_j$	$f_i \mathrel{+}= G_j^T(H_j h_j + f_j)$	$x_j \mathrel{+}= G_j x_i$
5:	$F_i \mathrel{+}= F_j G_j$	$e \mathrel{+}= F_j h_j$	$-\lambda_j \mathrel{+}= F_j^T(-\mu)$
6:	$D_j \mathrel{+}= F_j E_j$		
7:	$D_j \leftarrow D_j L_j^{-T}$	$h_j \leftrightarrow f_j$	
8:	$J_j \leftarrow L_j^{-1} J_j$	$d_j \leftarrow L_j^{-1} d_j$	$u_j \leftarrow -L_j^{-T} u_j$
9:	$H_i \mathrel{-}= J_j^T J_j$	$f_i \mathrel{-}= J_j^T d_j$	$u_j \mathrel{+}= J_j x_i$
10:	$F_i \mathrel{-}= D_j J_j$		
11:	$X \mathrel{+}= D_j D_j^T$	$e \mathrel{-}= D_j d_j$	$u_j \mathrel{+}= D_j^T(-\mu)$
12:	$X \leftarrow LL^T$	$e \leftarrow L^{-1} e$	$-\mu \leftarrow L^{-T}(-\mu)$

Repeating the eliminations of u_j and x_j inwardly to the root leads to

$$X_\emptyset(-\mu) = e_\emptyset. \tag{4.118}$$

With As. 2 satisfied, $X_\emptyset$ is symmetric positive definite and can be factorized ($X_\emptyset = L_\emptyset L_\emptyset^T$). The steps of the basic recursion above are listed in Table 4.7.

Outward Substitution

As in the outgoing control case, the global Lagrange multiplier μ is computed first from (4.118) using the factors of $X_\emptyset$. The computation order for the node variables is then u_j, x_j, λ_j, v_j^x and v_j^u, using the equations (4.107), (4.103b), (4.103a) and (4.98), respectively.

4.4. Tree-Sparse Inertia Correction

The TSPs are solved by an interior-point method that incorporates a filter line-search approach as globalization strategy (cf. Sect. 2.2.2). In this algorithmic framework, obtaining a descent direction (Def. 8) from the primal-dual system (2.21) requires the reduced KKT Matrix,

$$\Omega_\mathrm{r} = \begin{bmatrix} \hat{H} & \nabla c_\mathcal{E}^T \\ \nabla c_\mathcal{E} & 0 \end{bmatrix} \quad \text{with} \quad \hat{H} \in \mathbb{R}^{n\times n} \quad \text{and} \quad \nabla c_\mathcal{E} \in \mathbb{R}^{m\times n}, \tag{4.119}$$

to be regular with m positive and n negative eigenvalues. This inertia condition, i.e. $\text{inertia}(\Omega_\text{r}) = (m, n, 0)$, is satisfied if the Jacobian $\nabla c_\mathcal{E}$ has full row rank m and the Hessian of the Lagrangian $\hat{H}$ projected onto the null-space $\mathcal{N}(\nabla c_\mathcal{E})$ is positive definite (As. 1).

For well-formulated convex problems, i.e. for convex problems (Def. 5) with $\{\nabla c_i(x) \colon i \in \mathcal{E}\}$ being linearly independent, Assumption 1 is always satisfied [66]. Nonlinear problems, however, lack the benefit of convex problems. Using inertia corrections to overcome the drawback of an undesired inertia, the reduced KKT matrix (4.119) is replaced by a corrected version

$$\Omega_\text{r}^\text{corr} = \begin{bmatrix} \hat{H} + \Gamma_c & \nabla c_\mathcal{E}^T \\ \nabla c_\mathcal{E} & -\Gamma_r \end{bmatrix} \quad \text{with} \quad \Gamma_c, \Gamma_r \geq 0, \tag{4.120}$$

which satisfies the condition $\text{inertia}(\Omega_\text{r}^\text{corr}) = (n, m, 0)$. Typically, the correction terms Γ_c and Γ_r are multiples of the identity and determined in a trial-and-error approach based on attempts at factorizing $\Omega_\text{r}^\text{corr}$ [72, 103, 105]. When solving tree-sparse problems, such correction terms are not suitable for two reasons. First, a regularization of the form $\Gamma_r = \gamma_r I$ destroys the sparsity pattern of the tree-sparse KKT matrix for which the solution algorithms in Sect. 4.3 are designed. Second, adjusting the parameters γ_r and γ_c requires expensive refactorizations of the entire KKT matrix.

Addressing both drawbacks of the typical approach, a problem-tailored inertia correction heuristic for the TSPs is developed that is directly incorporated into the tree-sparse KKT solution procedure. This way, the tree-sparse KKT algorithms in Sect. 4.3 are extended to deal with rank-deficiencies and nonconvexities in the arising KKT systems. The proposed *tree-sparse inertia correction* is—to the best of the author's knowledge—not considered in the literature. Also, for nonlinear multistage stochastic problems fitting into the formulations of the TSPs, problem-tailored inertia corrections are—again, to the best of the author's knowledge—not considered in the literature.

Subsequently, Section 4.4.1 outlines the basic idea of the tree-sparse inertia correction. Afterwards, the regularization heuristic is discussed in Sect. 4.4.2 and Sect. 4.4.3 presents the problem-tailored convexification heuristic.

4.4.1. Extension of the Tree-Sparse KKT Algorithms

The tree-sparse KKT algorithms discussed in Sect. 4.3 are originally designed for well-formulated tree-sparse convex problems (cf. Sect. 3.2). For these problems, the required regularity assumptions (As. 2) are always satisfied guaranteeing that the tree-sparse factorization succeeds. This success depends on the successes of the Cholesky factorizations that are performed during the

tree-sparse factorization, i.e. the factorizations of the Hessian subblocks K_j and the fill-in block $X_\emptyset$. The main idea of the inertia correction is to ensure that these Cholesky factorizations are still performed even if the regularity assumptions are not satisfied. This is achieved by replacing the corresponding operations, i.e. items 1 and 12 in Table 4.5 as well as items 2 and 12 in Table 4.7, with the modifications

$$(X_\emptyset + \gamma_r I) = L_\emptyset L_\emptyset^T \quad \text{and} \quad (K_j + \gamma_j^c I) = L_j L_j^T \quad \text{for} \quad j \in V. \tag{4.121}$$

Setting the parameters $\gamma_r, \gamma_j^c \geq 0$ sufficiently large ensures that the respective Cholesky factors are evaluated successfully.

Basically, the parameters γ_r and γ_j^c are set when the respective Cholesky factorizations fail. In doing so, the correction terms are directly incorporated into the factorization stage of the tree-sparse KKT algorithms. This way, the tree-sparse KKT algorithms are extended to *modifying* solution approaches for KKT systems, i.e. the KKT matrix may be modified during the solution procedure.

4.4.2. Regularization Strategy

The regularization term Γ_r in (4.120) is used to clear out the zero eigenvalues in the inertia of $\Omega_\mathrm{r}^\mathrm{corr}$ that are caused by a rank-deficiency in the Jacobian $\nabla c_\mathcal{E}$ of the equality constraints. Using a multiple of the identity as regularizing term, i.e. $\Gamma_r = \gamma_r I$, any regularization parameter $\gamma_r > 0$ leads to full row rank in the lower block row of $\Omega_\mathrm{r}^\mathrm{corr}$ [105]. However, this common approach is incompatible with the tree-sparse KKT algorithms. Recall, for example, the linearized dynamics (4.75c) of the outgoing control TSP (4.3) and consider the following modification that accounts for the regularization:

$$G_j x_i + E_j u_i - x_j + \gamma_r \lambda_j + h_j = 0, \quad j \in V. \tag{4.122}$$

The regularization causes fill-in with the dynamic duals λ_j for which the tree-sparse KKT algorithms are not designed (cf. Sect. 4.3). In fact, the incorporated regularizing term in (4.122) is uncalled-for since the dynamic block G of the Jacobian $\nabla c_\mathcal{E}$ is already regular (cf. Sect. 4.3.1). More precisely, (A1) of As. 2 implies that only the global block F may cause a rank-deficiency in the tree-sparse Jacobian $\nabla c_\mathcal{E}$. Hence, regularizing the corresponding block row in $\Omega_\mathrm{r}^\mathrm{corr}$ (4.120) is sufficient to clear out the resulting zero eigenvalues in the inertia, i.e. the corrected version

of the reduced tree-sparse KKT matrix reads

$$\Omega_{\mathrm{r}}^{\mathrm{corr}} = \begin{bmatrix} \hat{H} + \Gamma_c & G^T & F^T \\ G & & \\ F & & -\gamma_r I \end{bmatrix} \quad \text{with} \quad \Gamma_c \geq 0 \quad \text{and} \quad \gamma_r \geq 0. \tag{4.123}$$

In the tree-sparse KKT algorithms, a rank-deficiency in $\nabla c_{\mathcal{E}}$ is detected at the attempt of factorizing the fill-in subblock $X_\emptyset$. In both control cases, the subblock $X_\emptyset$ is as sum of symmetric products at least positive semidefinite (see (4.91) and (4.116)), but it may be singular if (A1) of As. 2 is unsatisfied. Then, any parameter $\gamma_r > 0$ regularizes the subblock $X_\emptyset$ such that the Cholesky factorization

$$(X_\emptyset + \gamma_r I) = L_\emptyset L_\emptyset^T \tag{4.124}$$

exists. The operation (4.124) is performed at the very end of the factorization stage (see tables 4.5 and 4.7). Adjusting the regularization parameter γ_r does not affect previous performed operations and, therefore, does not require a refactorization of the entire matrix $\Omega_{\mathrm{r}}^{\mathrm{corr}}$ (4.123).

4.4.3. Convexification Strategy

In Sect. 4.4.1, the node subblocks K_j are modified using a *node convexification parameter* $\gamma_j^c > 0$ if the Cholesky factorization $K_j = L_j L_j^T$ fails. This way, the correction term Γ_c in (4.119) is of diagonal form, which reads in the outgoing control case as follows:

$$\Gamma_c = \mathrm{Diag}\left(\Gamma_j^c\right)_{j \in V} \quad \text{with} \quad \Gamma_j^c = \begin{bmatrix} & \\ & \gamma_j^c I \end{bmatrix}. \tag{4.125}$$

Hence, compared to the typical approach described in Sect. 2.2.2, the convexification Γ_c (4.125) is no longer a multiple of the identity. Moreover, Γ_c does not convexify the entire Hessian $\hat{H}$ but only its projection onto the null-space $\mathcal{N}(G)$ of the dynamic part G of the tree-sparse Jacobian $\nabla c_{\mathcal{E}}$ (cf. Sect. 4.2). The key advantage of this convexification strategy is that the inertia of the Hessian $\hat{H}$ can be adjusted without needing to refactorize the entire KKT matrix $\Omega_{\mathrm{r}}^{\mathrm{corr}}$. For a node $j \in V$, a modification of K_j affects only the node subblocks of its ancestors and not those of its descendants. Hence, the node convexification parameter γ_j^c can be determined at the attempt of factorizing the node subblock K_j during the factorization stage of the tree-sparse KKT algorithm (see item 1 in Table 4.5 as well as item 2 in Table 4.7).

Thus, convexifying the KKT matrix $\Omega_{\mathrm{r}}^{\mathrm{corr}}$ (4.119) using Γ_c (4.125) avoids expensive refac-

torizations of the KKT matrix in one iteration of the IPM algorithm. However, since the KKT matrix is modified the resulting primal-dual search direction ω_{pd} (4.35) might not be useful. On the other hand, using a multiple of the identity as correction term in (4.119) requires refactorizing $\Omega_{\text{r}}^{\text{corr}}$ each time the parameter γ_c is adjusted but is successfully applied in the literature [72, 103, 105]. Both arguments motivate the flexible convexification framework presented next, which allows correcting $\Omega_{\text{r}}^{\text{corr}}$ in either one of the two ways or to find a fair comprise between the both of them.

Convexification Framework

The convexification framework for the tree-sparse KKT system comprises an *outer convexification* as well as *local convexifications* for each node $j \in V$. The outer convexification initiates the factorization of the KKT matrix $\Omega_{\text{r}}^{\text{corr}}$ with $\gamma_j^c \equiv \bar{\gamma}_c$ for all $j \in V$. Starting with $\bar{\gamma}_c = 0$, the factorization of $\Omega_{\text{r}}^{\text{corr}}$ is tried for increasing values of $\bar{\gamma}_c$. Additionally, the inertia of $\Omega_{\text{r}}^{\text{corr}}$ can be adjusted by the local convexifications. Those try to apply the Cholesky factorizations $K_j + \gamma_j^c I = L_j L_j^T$ for increasing node convexification parameters

$$\gamma_j^c = \kappa_c^l \bar{\gamma}_c \quad \text{with} \quad \kappa_c > 1 \quad \text{and} \quad l = 0, \ldots, n_c^{\max}. \tag{4.126}$$

It is also possible to apply the local convexifications in a uniform way for each node. Activating this *uniform node convexification* means that the state node subblocks H_j of the tree-sparse Hessian are corrected with the same terms as the control node subblocks K_j, i.e. $H_j + \gamma_j^c I$. This way, for each node j the convexification term Γ_j^c becomes a multiple of the identity.

Now, in this convexification framework, setting $n_c^{\max} = \infty$ means disabling the outer convexification and use only local convexifications. On the other hand, returning to the typical approach of using a multiple of the identity $\Gamma_c = \gamma_c I$ is achieved by setting $n_c^{\max} = 0$ and activating the uniform node convexification.

4.5. Quasi-Newton Methods for Tree-Sparse Problems

Motivated from optimizing dynamic processes modeled by ordinary differential equations (ODEs) (cf. Sect. 3.3), this section considers TSPs without using explicit evaluations of second-order derivatives. In a quasi-Newton approach, approximations of the Hessian of the Lagrangian $\mathcal{L}$ (4.24) are generated using Hessian update strategies (cf. Sect. 2.2.3). Without accounting for the sparsity pattern of the problem at hand, quasi-Newton methods lead to block-dense systems. The sparsity pattern of the problem is destroyed, making standard sparse

solvers highly inefficient and problem-tailored ones like the tree-sparse KKT algorithms inapplicable. In the following, a problem-tailored quasi-Newton approach for tree-sparse problems is proposed. Update formulae are applied node-wise to the Hessian of the Lagrangian, i.e. to each set of node subblocks that belong together, instead of applying the formulae to the entire Hessian at once.

Quasi-Newton methods for tree-sparse problems are—to the best of the author's knowledge—not considered in the literature. Also, for nonlinear multistage stochastic problems that are covered in the formulation of the TSPs (cf. Chap. 3), problem-tailored quasi-Newton approaches are—again, to the best of the author's knowledge—not considered in the literature. Reports of quasi-Newton approaches for deterministic optimal control problems (OCPs) are given, for example, by Culver and Shoemaker [19] and by Asprion, Chinellato and Guzzella [52, 4]. The latter consider dynamic processes modeled by ODEs and formulate OCPs featuring discretized dynamics in outgoing control form.

The TSPs are solved using an IPM framework that employs a line-search approach as globalization strategy [72]. To obtain descent directions from the reduced KKT systems (cf. Sect. 2.2.2), update formulae such as SR1 leading to indefinite approximations are usually dismissed. In [52], for example, the authors apply a BFGS-based update strategy to solve the considered OCPs using Ipopt [104, 105]. Indefinite rank-one updates, on the other hand, are indeed used in [19], but the authors avoid indefinite Hessian approximations by skipping updates that cause indefiniteness. However, undesired indefinite approximations can actually be considered in a line-search IPM framework by using the inertia corrections at hand (cf. Sect. 4.4). Relying on the tree-sparse inertia corrections, the proposed quasi-Newton approach explicitly do include indefinite Hessian approximations in general and the SR1 update rule in specific.

In the following, Sect. 4.5.1 provides a short overview of quasi-Newton approaches for sparse problems in general and reviews Hessian update strategies for partially separable functions (cf. Sect. 4.1.3) in specific. Based on the discussions on partially separable functions, sections 4.5.2 and 4.5.3 present tree-sparse Hessian update strategies tailored to the problems in their respective control forms.

4.5.1. Hessian Updates for Partially Separable Functions

To maintain the computational tractability of a problem, quasi-Newton methods in large-scale optimization must not alter its sparsity pattern too much, i.e. the applied Hessian update strategy keeps additional fill-in at a minimum. Such sparse quasi-Newton approaches for unconstrained and constrained optimization are considered repeatedly in the literature, e.g. in [91, 68, 58, 31, 57, 24]. General aspects of derivative approximations in the context of

optimization are given, for example, by Polak [67]. Griewank and Toint establish *partitioned quasi-Newton methods* for specific functions they call *partially separable* [37, 38, 39, 40]. The special case of unconstrained optimization problems arising from discretized time-continuous models is considered, for example, by Malmedy and Toint [62].

Griewank and Toint observe that many functions in finite-dimensional optimization problems resulting from discretizing an infinite-dimensional counterpart are stated as sums [37],

$$\zeta(y) = \sum_{i=1}^{M} \zeta_i(y) \quad \text{with} \quad \zeta_i : \mathbb{R}^N \to \mathbb{R}, \quad i = 1, \dots, M. \tag{4.127}$$

Each *contribution* ζ_i corresponds to an element of the discretization grid, e.g. a decomposition of a time interval, a mesh approximating a geometric domain, or—as it is in this work—a tree representing the stochastic process. The contributions ζ_i only depend on a small number of the optimization variables y, i.e. ζ is a *partially separable function* of the form (4.17).

Two key ingredients lead to computationally efficient Hessian update strategies for (4.127) preserving the specific structure of the Hessian $\nabla^2\zeta$. First, with ζ being a sum of contributions, its derivatives $\nabla\zeta$ and $\nabla^2\zeta$ feature the same characteristic, i.e. those read

$$\nabla\zeta(y) = \sum_{i=1}^{M} \nabla\zeta_i(y) \quad \text{and} \quad \nabla^2\zeta(y) = \sum_{i=1}^{M} \nabla^2\zeta_i(y). \tag{4.128}$$

Rather than approximating the overall Hessian $\nabla^2\zeta$ directly, the idea is to approximate each contribution $\nabla^2\zeta_i$ separately [37]. Monitoring these approximations and accumulating them to the overall Hessian approximation leads to the update strategy

$$\mathfrak{B}^{(k)} = \sum_{i=1}^{M} \mathfrak{B}_i^{(k)} \quad \text{with} \quad \mathfrak{B}^{(k)} \approx \nabla^2\zeta(y^{(k)}) \quad \text{and} \quad \mathfrak{B}_i^{(k)} \approx \nabla^2\zeta_i(y^{(k)}). \tag{4.129}$$

Thereby, the update strategy (4.129) preserves the specific structure of the Hessian of the function ζ resulting from its presentation as a sum (4.127). The matrices $\mathfrak{B}_i^{(k)}$ are obtained by applying one of the formulae (2.32), and the same one to all contributions, using

$$\mathfrak{s}_i^{(k)} = \mathfrak{s}^{(k)}, \quad \mathfrak{g}_i^{(k)} = \nabla\zeta_i(y^{(k+1)}) - \nabla\zeta_i(y^{(k)}) \quad \text{and} \quad \mathfrak{r}_i^{(k)} = \mathfrak{s}_i^{(k)} - \mathfrak{B}_i^{(k)}\mathfrak{g}_i^{(k)}. \tag{4.130}$$

The second ingredient for approximating the Hessian $\nabla^2\zeta$ allows to compute its contributions efficiently: evaluate $\mathfrak{B}_i^{(k)}$ only on the domain of the contribution ζ_i instead of doing the same on the entire domain of the overall function ζ. More precisely, evaluate only the nontrivial entries of $\mathfrak{B}_i^{(k)}$ and contribute these to the overall approximation $\mathfrak{B}^{(k)}$.

In the following, functions $\mathcal{P}_i$ are introduced that first map the derivatives $\nabla\zeta_i$ and $\nabla^2\zeta_i$ onto the domain of the respective contribution ζ_i and then reverse the respective processes. These mappings are used to establish Hessian update strategies operating only on the domains of the contributions.

Gradients of Partially Separable Functions

Recall from Sect. 4.1.3 the definition of a partially separable function,

$$\zeta(y) = \sum_{i=1}^{M} \zeta_i\left((y_j)_{j\in\mathcal{J}_i}\right) \quad \text{with} \quad \mathcal{J}_i \subseteq \{1,\dots,N\} \quad \text{for} \quad i = 1,\dots,M, \tag{4.131}$$

and let $\mathcal{P}_i^1$ be the mapping of the vector y onto the domain of the contribution ζ_i, i.e.

$$\mathcal{P}_i^1 : \mathbb{R}^N \to \mathbb{R}^{|\mathcal{J}_i|} \quad \text{with} \quad \mathcal{P}_i^1(y) = (y_j)_{j\in\mathcal{J}_i}. \tag{4.132}$$

The reverse mapping $\mathcal{P}_i^{-1}$ is expressed by means of the characteristic function χ and reads

$$\mathcal{P}_i^{-1} : \mathbb{R}^{|\mathcal{J}_i|} \to \mathbb{R}^N \quad \text{with} \quad \mathcal{P}_i^{-1}(\mathcal{P}_i^1(y)) = \left(\chi(j\in\mathcal{J}_i)y_j\right)_{j=1,\dots,N}. \tag{4.133}$$

Note that $\mathcal{P}_i^{-1}$ is not the inverse of $\mathcal{P}_i^1$ since mapping a vector y back and forth leads to $y_j = 0$ for $j \notin \mathcal{J}_i$. Now, the gradient of a partially separable function (4.131) by means of these mappings reads

$$\nabla\zeta(y) = \sum_{i=1}^{M} \mathcal{P}_i^{-1}\left(\nabla_{\mathcal{P}_i^1(y)}\zeta_i\left(\mathcal{P}_i^1(y)\right)\right). \tag{4.134}$$

Hessian Updates for Partially Separable Functions

The Hessian $\nabla^2\zeta$ is stated the same way as the gradient $\nabla\zeta$ (4.134). First, a contribution $\nabla^2\zeta_i$ is mapped based on

$$\mathcal{P}_i^2 : \mathbb{R}^{n\times n} \to \mathbb{R}^{|\mathcal{J}_i|\times|\mathcal{J}_i|} \quad \text{with} \quad \mathcal{P}_i^2(Y) = \left[Y_{kl}\right]_{k,l\in\mathcal{J}_i}. \tag{4.135}$$

Afterwards, the mapping $\mathcal{P}_i^2$ is reversed by using its counterpart

$$\mathcal{P}_i^{-2} : \mathbb{R}^{|\mathcal{J}_i|\times|\mathcal{J}_i|} \to \mathbb{R}^{N\times N} \quad \text{with} \quad \mathcal{P}_i^{-2}(\mathcal{P}_i^2(Y)) = \left[\chi(k\in\mathcal{J}_i)\chi(l\in\mathcal{J}_i)Y_{kl}\right]_{k,l=1,\dots,N}. \tag{4.136}$$

The mapping $\mathcal{P}_i^2$ is not used explicitly but expressed by means of the mapping $\mathcal{P}_i^1$ (4.132). Therefore, the Hessian of a partially separable functions reads

$$\nabla^2\zeta(y) = \sum_{i=1}^{M} \mathcal{P}_i^{-2}\left(\nabla^2_{\mathcal{P}_i^1(y)\mathcal{P}_i^1(y)}\zeta_i\left(\mathcal{P}_i^1(y)\right)\right). \tag{4.137}$$

Applying the presentation (4.137) to the update strategy (4.129) leads to

$$\mathfrak{B}^{(k)} = \sum_{i=1}^{M} \mathcal{P}_i^{-2}\left(\hat{\mathfrak{B}}_i^{(k)}\right) \quad \text{with} \quad \hat{\mathfrak{B}}_i^{(k)} \approx \nabla^2_{\mathcal{P}_i^1(y)\mathcal{P}_i^1(y)}\zeta_i(\mathcal{P}_i^1(y)). \tag{4.138}$$

The local approximation $\hat{\mathfrak{B}}_i^{(k)}$ is composed of the vectors (4.130) on the domain of the respective contribution, i.e.

$$\hat{\mathfrak{s}}_i^{(k)} = \mathcal{P}_i^1(\mathfrak{s}^{(k)}), \quad \hat{\mathfrak{g}}_i^{(k)} = \mathcal{P}_i^1(\mathfrak{g}_i^{(k)}) \quad \text{and} \quad \hat{\mathfrak{r}}_i^{(k)} = \hat{\mathfrak{s}}_i^{(k)} - \hat{\mathfrak{B}}_i^{(k)}\hat{\mathfrak{g}}_i^{(k)}. \tag{4.139}$$

Completely Separable Functions

Specific forms of partially separable functions (4.131) comprise contributions ζ_i without overlapping derivatives, i.e. the mappings of $\nabla\zeta_i$ and $\nabla^2\zeta_i$ coincide with partial derivatives of ζ with respect to the corresponding mapping of y.

Two variables y_{j_1} and y_{j_2} are said to be *joint variables* if there exists an index set $\mathcal{J}_i$ containing both indices j_1 and j_2, i.e. it is $j_1, j_2 \in \mathcal{J}_i$ for at least one $i \in \{1, \dots, M\}$. *Completely separable functions* are partially separable functions (4.127) without joint variables, i.e. the partial separability is described by pairwise disjoint index sets $\mathcal{J}_{i_1} \cap \mathcal{J}_{i_2} = \emptyset$ for $i_1 \neq i_2$. For completely separable functions, a partial derivative with respect to joint variables coincide with the respective derivative of one of its contributions, i.e. a partial gradient reads

$$\nabla_{\mathcal{P}_i^1(y)}\zeta(y) = \nabla_{\mathcal{P}_i^1(y)}\zeta_i(\mathcal{P}_i^1(y)), \tag{4.140}$$

and a partial Hessian satisfies

$$\nabla^2_{\mathcal{P}_i^1(y)\mathcal{P}_i^1(y)}\zeta(y) = \nabla^2_{\mathcal{P}_i^1(y)\mathcal{P}_i^1(y)}\zeta_i(\mathcal{P}_i^1(y)). \tag{4.141}$$

Applying $\mathcal{P}_i^2$ to $\nabla^2\zeta$ (4.137) and using (4.141) leads to the relation

$$\mathcal{P}_i^2(\nabla^2\zeta(y)) = \nabla^2_{\mathcal{P}_i^1(y)\mathcal{P}_i^1(y)}\zeta_i\left(\mathcal{P}_i^1(y)\right). \tag{4.142}$$

4.5.2. Tree-Sparse Hessian Update Strategies – Outgoing Control

In the following, the Hessian update strategy for partially separable functions discussed in Sect. 4.5.1 is applied to approximate the Hessians of the Lagrangian for the TSP in outgoing control form (4.3). For this, recall the formulation of the Lagrangian (4.40) corresponding to the outgoing TSP (4.3) reading

$$\mathcal{L}(y,\eta) = \sum_{j\in V} \mathcal{L}_j(x_j,u_j,\eta) - \xi^T s \tag{4.143}$$

with the Lagrangian node functions $\mathcal{L}_j$ (4.41) as well as the vector η (4.25) subsuming all variables but the primal variables y. The Lagrangian (4.143) is completely separable with respect to the node variables $y_j = (x_j, u_j)$. Therefore, let $\mathcal{P}_j^1$ map the primal vector y onto the node variables y_j, i.e.

$$\mathcal{P}_j^1 : \mathbb{R}^{n^v} \to \mathbb{R}^{n_j^v} \quad \text{with} \quad \mathcal{P}_j^1(y) = y_j = \begin{pmatrix} x_j \\ u_j \end{pmatrix}. \tag{4.144}$$

The remaining mappings

$$\mathcal{P}_j^{-1} : \mathbb{R}^{n_j^v} \to \mathbb{R}^{n^v}, \quad \mathcal{P}_j^2 : \mathbb{R}^{n^v\times n^v} \to \mathbb{R}^{n_j^v\times n_j^v} \quad \text{and} \quad \mathcal{P}_j^{-2} : \mathbb{R}^{n_j^v\times n_j^v} \to \mathbb{R}^{n^v\times n^v} \tag{4.145}$$

are defined appropriately following the lines in Sect. 4.5.1. With the Lagrangian (4.143) being completely separable, it holds that

$$\nabla^2_{\mathcal{P}_j^1(y)\mathcal{P}_j^1(y)}\mathcal{L}(y,\eta) = \nabla^2_{\mathcal{P}_j^1(y)\mathcal{P}_j^1(y)}\mathcal{L}_j(\mathcal{P}_j^1(y),\eta) = \nabla^2_{y_jy_j}\mathcal{L}_j(y_j,\eta). \tag{4.146}$$

Now, each Hessian (4.146) is approximated by a node subblock $\hat{\mathfrak{B}}_j$ using the same update formula for each node $j \in V$ with

$$\mathfrak{s}_j^{(k)} = y_j^{(k+1)} - y_j^{(k)}, \quad \mathfrak{g}_j^{(k)} = \nabla_{y_j}\mathcal{L}_j^+(y_j^{(k+1)}) - \nabla_{y_j}\mathcal{L}_j^-(y_j^{(k)}), \quad \mathfrak{r}_j^{(k)} = \mathfrak{s}_j^{(k)} - \hat{\mathfrak{B}}_j^{(k)}\mathfrak{g}_j^{(k)}, \tag{4.147}$$

where the functions $\mathcal{L}_j^+$ and $\mathcal{L}_j^-$ are defined by

$$\mathcal{L}_j^+(y_j^{(k+1)}) := \mathcal{L}_j(y_j^{(k+1)},\eta^{(k)}) \quad \text{and} \quad \mathcal{L}_j^-(y_j^{(k)}) := \mathcal{L}_j(y_j^{(k)},\eta^{(k)}). \tag{4.148}$$

The approximation $\hat{\mathfrak{B}}_j^{(k)}$ of the Hessian of the node function $\mathcal{L}_j$ in (4.143) reads

$$\hat{\mathfrak{B}}_j^{(k)} \approx \nabla^2_{y_j y_j} \mathcal{L}_j(y_j^{(k)}, \eta^{(k)}) = \begin{bmatrix} H_j^{(k)} & (J_j^{(k)})^T \\ J_j^{(k)} & K_j^{(k)} \end{bmatrix}, \tag{4.149}$$

where the node subblocks H_j, K_j and J_j are given by (4.45). Finally, the mapping $\mathcal{P}_j^{-2}$ places the block $\hat{\mathfrak{B}}_j^{(k)}$ onto the diagonal block of $\nabla^2_{yy}\mathcal{L}$ corresponding to node j. The overall approximation of the Hessian in the outgoing control case reads

$$\nabla^2_{yy}\mathcal{L}(y^{(k)}, \eta^{(k)}) \approx \sum_{j \in V} \mathcal{P}_j^{-2}(\hat{\mathfrak{B}}_j^{(k)}). \tag{4.150}$$

4.5.3. Tree-Sparse Hessian Update Strategies – Incoming Control

The Lagrangian corresponding to the TSP in the incoming control form (4.4) is composed of the two types of node functions $\mathcal{L}_{ij}$ (4.54) and $\mathcal{L}_j$ (4.55) reading

$$\mathcal{L}(y, \eta) = \sum_{j \in V} \mathcal{L}_{ij}(x_i, u_j, \eta) + \sum_{j \in V} \mathcal{L}_j(x_j, \eta) - \xi^T s, \tag{4.151}$$

where η is again the vector of all variables but the primal variables y. Note that in contrast to the outgoing control case discussed in Sect. 4.5.2, the Lagrangian (4.151) is not completely separable.

Now, the overall Hessian $\nabla^2_{yy}\mathcal{L}$ of the Lagrangian (4.151) is approximated by means of approximations $\mathfrak{B}_{ij} \approx \nabla^2_{yy}\mathcal{L}_{ij}$ and $\mathfrak{B}_j \approx \nabla^2_{yy}\mathcal{L}_j$ for the respective node functions. For this, the two mappings

$$\mathcal{P}_{ij}^1 : \mathbb{R}^n \to \mathbb{R}^{n_j^v}, \quad \mathcal{P}_{ij}^1(y) = \begin{pmatrix} x_i \\ u_j \end{pmatrix} \quad \text{and} \quad \mathcal{P}_j^1 : \mathbb{R}^n \to \mathbb{R}^{n_j^x}, \quad \mathcal{P}_j^1(y) = x_j \tag{4.152}$$

are introduced and the corresponding mappings $\mathcal{P}_{ij}^{-1}$, $\mathcal{P}_{ij}^2$, $\mathcal{P}_{ij}^{-2}$ and $\mathcal{P}_j^{-1}$, $\mathcal{P}_j^2$, $\mathcal{P}_j^{-2}$ are defined following the lines in Sect. 4.5.1. Then, the projected approximations of the Hessians of the node functions $\mathcal{L}_{ij}$ read

$$\hat{\mathfrak{B}}_{ij}^{(k)} \approx \nabla^2_{\mathcal{P}_{ij}^1(y)\mathcal{P}_{ij}^1(y)} \mathcal{L}_{ij}(x_i^{(k)}, u_j^{(k)}, \eta^{(k)}) = \begin{bmatrix} H_{ij}^{(k)} & (J_j^{(k)})^T \\ J_j^{(k)} & K_j^{(k)} \end{bmatrix} \tag{4.153}$$

with H_{ij}, K_j and J_j as defined in (4.61a) to (4.61c). The projected approximations of the

Hessians of the node functions $\mathcal{L}_j$ are

$$\hat{\mathfrak{B}}_j^{(k)} \approx \nabla^2_{x_j x_j} \mathcal{L}_j(x_j^{(k)}, \eta^{(k)}) = \bar{H}_j^{(k)}, \tag{4.154}$$

where $\bar{H}_j$ is given by (4.61d). Using the definitions

$$\mathcal{L}_{ij}^+(x_i^{(k+1)}, u_j^{(k+1)}) := \mathcal{L}_{ij}(x_i^{(k+1)}, u_j^{(k+1)}, \eta^{(k)}), \quad \mathcal{L}_j^+(x_j^{(k+1)}) := \mathcal{L}_j(x_j^{(k+1)}, \eta^{(k)}) \tag{4.155a}$$

$$\mathcal{L}_{ij}^-(x_i^{(k)}, u_j^{(k)}) := \mathcal{L}_{ij}(x_i^{(k)}, u_j^{(k)}, \eta^{(k)}), \quad \mathcal{L}_j^-(x_j^{(k)}) := \mathcal{L}_j(x_j^{(k)}, \eta^{(k)}), \tag{4.155b}$$

the approximations $\hat{\mathfrak{B}}_{ij}$ and $\hat{\mathfrak{B}}_j$ are updated by applying one of the formulae (2.32), where the respective differences of the iterates are given by

$$\mathfrak{s}_{ij}^{(k)} = \begin{pmatrix} x_i^{(k+1)} - x_i^{(k)} \\ u_j^{(k+1)} - u_j^{(k)} \end{pmatrix} \quad \text{and} \quad \mathfrak{s}_j^{(k)} = x_j^{(k+1)} - x_j^{(k)}, \tag{4.156}$$

the corresponding differences of the gradients of the Lagrangian read

$$\mathfrak{g}_{ij}^{(k)} = \begin{pmatrix} \nabla_{x_i} \mathcal{L}_{ij}^+(x_i^{(k+1)}, u_j^{(k+1)}) - \nabla_{x_i} \mathcal{L}_{ij}^-(x_i^{(k)}, u_j^{(k)}) \\ \nabla_{u_j} \mathcal{L}_{ij}^+(x_i^{(k+1)}, u_j^{(k+1)}) - \nabla_{u_j} \mathcal{L}_{ij}^-(x_i^{(k)}, u_j^{(k)}) \end{pmatrix}, \tag{4.157a}$$

$$\mathfrak{g}_j^{(k)} = \nabla_{x_j} \mathcal{L}_j^+(x_j^{(k+1)}) - \nabla_{x_j} \mathcal{L}_j^-(x_j^{(k)}). \tag{4.157b}$$

and, finally, it is $\mathfrak{r}_{ij}^{(k)} = \mathfrak{g}_{ij}^{(k)} - \hat{\mathfrak{B}}_{ij}^{(k)} \mathfrak{s}_{ij}^{(k+1)}$ and $\mathfrak{r}_j^{(k)} = \mathfrak{g}_j^{(k)} - \hat{\mathfrak{B}}_j^{(k)} \mathfrak{s}_j^{(k)}$, respectively.

Considering a simple tree comprising the root 0 and the successors $S(0) = \{1, 2\}$, the Hessian of the Lagrangian (4.151) has the form

$$\nabla^2_{yy} \mathcal{L}(y^{(k)}, \eta^{(k)}) = \begin{bmatrix} K_0^{(k)} & & & & & \\ & H_0^{(k)} & (J_1^{(k)})^T & & (J_2^{(k)})^T & \\ & J_1^{(k)} & K_1^{(k)} & & & \\ & & & H_1^{(k)} & & \\ & J_2^{(k)} & & & K_2^{(k)} & \\ & & & & & H_2^{(k)} \end{bmatrix} \tag{4.158}$$

with

$$H_0^{(k)} = \bar{H}_0^{(k)} + H_{01}^{(k)} + H_{02}^{(k)}, \quad H_1^{(k)} = \bar{H}_1^{(k)} \quad \text{and} \quad H_2^{(k)} = \bar{H}_2^{(k)}. \tag{4.159}$$

The overall approximation of the Hessian of the Lagrangian (4.151) reads

$$\nabla^2_{yy}\mathcal{L}(y^{(k)},\eta^{(k)}) \approx \sum_{j\in V} \mathcal{P}_{ij}^{-2}(\hat{\mathfrak{B}}_{ij}^{(k)}) + \sum_{j\in V} \mathcal{P}_{j}^{-2}(\hat{\mathfrak{B}}_{j}^{(k)}). \tag{4.160}$$

Hence, the mappings $\mathcal{P}_{ij}^{-2}$ and $\mathcal{P}_{j}^{-2}$ place the node subblocks K_j and $\bar{H}_j$ onto the diagonal corresponding to node j. Additionally, $\mathcal{P}_{ij}^{-2}$ places the node subblock H_{ij} onto the diagonal as well as J_j and J_j^T onto secondary diagonals corresponding to the predecessor i.

4.6. Numerical Issues

Subsequently, some numerical issues are outlined concerning the treatment of the TSPs. First of all, Section 4.6.1 presents further tree-sparse algorithms in the context of tree-sparse optimization. All tree-sparse algorithms except for the outward substitutions are considered as so-called *inward algorithms*, meaning the respective node operations are performed in an inward sweep over the tree nodes (cf. Chap. 5). The reason for this is the specific concern for accumulating a large amount of data in a numerically stable way, which is discussed in Sect. 4.6.2. Finally, Section 4.6.3 provides some remarks on problem scaling for the TSPs.

4.6.1. Other Tree-Sparse Algorithms

In the following, two additional tree-sparse algorithms are presented that are, besides the tree-sparse KKT algorithms, most important in the context of tree-sparse optimization. First, the matrix-vector product (MVP) with parts of the KKT matrix Ω (4.67) takes center stage in both evaluating the gradient of the Lagrangian $\nabla_y\mathcal{L}$ (4.26) and evaluating the constraints $c_\mathcal{E}$ and $c_\mathcal{I}$ for tree-sparse QPs. Second, evaluating the problem data of TSPs are also realized by tree-sparse algorithms.

Tree-Sparse Matrix-Vector Products

Consider the following MVP with the KKT matrix,

$$\begin{pmatrix} \nu \\ \alpha \\ \rho \end{pmatrix} \mathrel{+}= \begin{bmatrix} H & A & B \\ A^T & & \\ B^T & & \end{bmatrix} \begin{pmatrix} y \\ z \\ v \end{pmatrix}, \tag{4.161}$$

Table 4.8.: Tree-sparse MVP algorithm – Outgoing control case

	H-Block	A-Block	A^T-Block	B-Block	B^T-Block
1:	$f_j \mathrel{+}= H_j x_j$	$h_j \mathrel{+}= G_j x_i$	$f_j \mathrel{-}= \lambda_j$	$r_j \mathrel{+}= F_j^r x_j$	$f_j \mathrel{+}= {F_j^r}^T v_j$
2:	$f_j \mathrel{+}= J_j^T u_j$	$h_j \mathrel{+}= E_j u_i$	$f_i \mathrel{+}= G_j^T \lambda_j$	$r_j \mathrel{+}= D_j^r u_j$	$d_j \mathrel{+}= {D_j^r}^T v_j$
3:	$d_j \mathrel{+}= K_j u_j$	$h_j \mathrel{-}= x_j$	$d_i \mathrel{+}= E_j^T \lambda_j$		
4:	$d_j \mathrel{+}= J_j x_j$	$e \mathrel{+}= F_j x_j$	$f_j \mathrel{+}= F_j^T \mu$		
5:		$e \mathrel{+}= D_j u_j$	$d_j \mathrel{+}= D_j^T \mu$		

Table 4.9.: Tree-sparse MVP algorithm – Incoming control case

	H-Block	A-Block	A^T-Block	B-Block	B^T-Block
1:	$d_j \mathrel{+}= K_j u_j$	$h_j \mathrel{+}= E_j u_j$	$d_j \mathrel{+}= E_j^T \lambda_j$	$r_j^u \mathrel{+}= F_{ij}^r x_i$	$f_i \mathrel{+}= {F_{ij}^r}^T v_j^u$
2:	$f_j \mathrel{+}= H_j x_j$	$h_j \mathrel{-}= x_j$	$d_j \mathrel{+}= D_j^T \mu$	$r_j^u \mathrel{+}= D_j^r u_j$	$d_j \mathrel{+}= {D_j^r}^T v_j^u$
3:	$f_i \mathrel{+}= J_j^T u_j$	$e \mathrel{+}= D_j u_j$	$f_j \mathrel{-}= \lambda_j$	$r_j^x \mathrel{+}= F_j^r x_j$	$f_j \mathrel{+}= {F_j^r}^T v_j^x$
4:	$d_j \mathrel{+}= J_j x_i$	$e \mathrel{+}= F_j x_j$	$f_j \mathrel{+}= F_j^T \mu$		
5:		$h_j \mathrel{+}= G_j x_i$	$f_i \mathrel{+}= G_j^T \lambda_j$		

and recall the tree-sparse node subblock notation from the previous sections, i.e. the matrix node subblocks (4.42) and (4.45), the right-hand side notation

$$\nu = \left((f_j, d_j)\right)_{j\in V}, \quad \alpha = \left(\left(h_j\right)_{j\in V}, e_V\right), \quad \rho = \left(r_j\right)_{j\in V}, \tag{4.162}$$

and the argument vector node subblocks

$$y = \left((x_j, u_j)\right)_{j\in V}, \quad z = \left(\left(\lambda_j\right)_{j\in V}, \mu\right) \quad \text{and} \quad v = \left(v_j\right)_{j\in V}. \tag{4.163}$$

Table 4.8 lists the node operations for performing the MVP (4.161) in the outgoing control case. The reading of this table is the same as those of the tree-sparse KKT algorithms. All operations in one column are performed from top to bottom[1].

The incoming control version of the tree-sparse MVP (4.161) listed in Table 4.9 is based on the notation (4.56) to (4.58) for the matrix node subblocks as well as the right-hand side notation

$$\nu := \left((d_j, f_j)\right)_{j\in V}, \quad \alpha := \left(\left(h_j\right)_{j\in V}, e_V\right), \quad \rho := \left(\left(r_j^u, r_j^x\right)\right)_{j\in V}, \tag{4.164}$$

[1]The order of performing the node operations does not affect the outcome of the overall operations.

Table 4.10.: Evaluation of zero-order and first-order problem data – Outgoing control case

	Objective	Constraints	Gradient	Derivatives
1:	$\nu \mathrel{+}= \phi_j(x_j, u_j)$	$h_j \leftarrow g_j(x_i, u_i)$	$f_j \leftarrow \nabla_{x_j} \phi_j(x_j, u_j)$	$G_j \leftarrow \nabla_{x_i} g_j(x_i, u_i)$
2:		$h_j \mathrel{-}= x_j$	$d_j \leftarrow \nabla_{u_j} \phi_j(x_j, u_j)$	$E_j \leftarrow \nabla_{u_i} g_j(x_i, u_i)$
3:		$r_j \leftarrow r_j(x_j, u_j)$		$F_j^r \leftarrow \nabla_{x_j} r_j(x_j, u_j)$
4:		$e \mathrel{+}= f_j(x_j, u_j)$		$D_j^r \leftarrow \nabla_{u_j} r_j(x_j, u_j)$
5:				$F_j \leftarrow \nabla_{x_j} f_j(x_j, u_j)$
6:				$D_j \leftarrow \nabla_{u_j} f_j(x_j, u_j)$

and the argument vector subblocks

$$y = ((u_j, x_j))_{j \in V}\,, \quad z = \Big((\lambda_j)_{j \in V}\,, \mu\Big) \quad \text{and} \quad v = \big((v_j^u, v_j^x)\big)_{j \in V}\,. \tag{4.165}$$

Evaluation of Tree-Sparse Problem Data

Evaluating an NLP (4.1) includes the evaluation of the problem functions $f, c_{\mathcal{E}}$ and $c_{\mathcal{R}}$ as well as their respective first-order and second-order derivatives. For smooth NLPs, evaluating the problem data can be divided into the following five tasks:

1. the *objective* value $f(y)$,
2. the values $c_{\mathcal{E}}(y)$ and $c_{\mathcal{I}}(y)$ of the *constraint* functions,
3. the *gradient* of the objective,
4. the Jacobians or first-order *derivatives* of the constraints and
5. the *Hessian* of the Lagrangian (4.24).

The operations for evaluating the outgoing TSP (4.3) are listed in tables 4.10 and 4.11 whereas tables 4.12 and 4.13 list the respective operations for evaluating the incoming TSP (4.4).

4.6.2. Stable Accumulation

In tree-sparse optimization, one has to deal with very large vectors and needs to address the effects of rounding errors and cancellation that might occur while creating accumulated information [45]. In the presented tree-sparse algorithms, i.e. the KKT algorithms in Sect. 4.3 as well as the algorithms for evaluating MVPs or the problem data of the TSPs in Sect. 4.6.1, such accumulations arise in context with the objective function and the global equality constraints.

Table 4.11.: Evaluation of the Hessian of the Lagrangian – Outgoing control case

	H-Block	K-Block	J-Block
1:	$H_j \mathrel{+}= \nabla^2_{x_j x_j} \phi_j(x_j, u_j)$	$K_j \mathrel{+}= \nabla^2_{u_j u_j} \phi_j(x_j, u_j)$	$J_j \mathrel{+}= \nabla^2_{u_j x_j} \phi_j(x_j, u_j)$
	for $\tau : m$ **do**	**for** $\tau : m$ **do**	**for** $\tau : m$ **do**
2:	$H_j \mathrel{-}= \mu_\tau \nabla^2_{x_j x_j} f_{j\tau}(x_j, u_j)$	$K_j \mathrel{-}= \nabla^2_{u_j u_j} f_{j\tau}(x_j, u_j)$	$J_j \mathrel{-}= \nabla^2_{u_j x_j} f_{j\tau}(x_j, u_j)$
	for $\tau : l^r_j$ **do**	**for** $\tau : l^r_j$ **do**	**for** $\tau : l^r_j$ **do**
3:	$H_j \mathrel{-}= v_{j\tau} \nabla^2_{x_j x_j} r_{j\tau}(x_j, u_j)$	$K_j \mathrel{-}= v_{j\tau} \nabla^2_{u_j u_j} r_{j\tau}(x_j, u_j)$	$J_j \mathrel{-}= v_{j\tau} \nabla^2_{u_j x_j} r_{j\tau}(x_j, u_j)$
	for $\tau : n^x_j$ **do**	**for** $\tau : n^x_j$ **do**	**for** $\tau : n^x_j$ **do**
4:	$H_i \mathrel{-}= \lambda_{j\tau} \nabla^2_{x_i x_i} g_{j\tau}(x_i, u_i)$	$K_i \mathrel{-}= \lambda_{j\tau} \nabla^2_{u_i u_i} g_{j\tau}(x_i, u_i)$	$J_i \mathrel{-}= \lambda_{j\tau} \nabla^2_{u_i x_i} g_{j\tau}(x_i, u_i)$

Table 4.12.: Evaluation of zero-order and first-order problem data – Incoming control case

	Objective	Constraints	Gradient	Derivatives
1:	$\nu \mathrel{+}= \phi_{ij}(x_i, u_j)$	$h_j \leftarrow g_j(x_i, u_j)$	$d_j \leftarrow \nabla_{u_j} \phi_{ij}(x_i, u_j)$	$G_j \leftarrow \nabla_{x_i} g_j(x_i, u_i)$
2:	$\nu \mathrel{+}= \phi_j(x_j)$	$h_j \mathrel{-}= x_j$	$f_j \mathrel{+}= \nabla_{x_j} \phi_j(x_j)$	$E_j \leftarrow \nabla_{u_i} g_j(x_i, u_i)$
3:		$r^u_j \leftarrow r_{ij}(x_i, u_j)$		$F^r_{ij} \leftarrow \nabla_{x_i} r_{ij}(x_i, u_j)$
4:		$r^x_j \leftarrow r_j(x_j)$		$D^r_j \leftarrow \nabla_{u_j} r_{ij}(x_i, u_j)$
5:		$e \mathrel{+}= f_{ij}(x_i, u_j)$		$F^r_j \leftarrow \nabla_{x_j} r_j(x_j)$
6:		$e \mathrel{+}= f_j(x_j)$		$F_j \mathrel{+}= \nabla_{x_j} f_j(x_j)$
7:				$D_j \leftarrow \nabla_{u_j} f_{ij}(x_i, u_j)$
8:			$f_i \mathrel{+}= \nabla_{x_i} \phi_{ij}(x_i, u_j)$	$F_i \mathrel{+}= \nabla_{x_i} f_{ij}(x_i, u_j)$

Table 4.13.: Evaluation of the Hessian of the Lagrangian – Incoming control case

	H-Block	K-Block	J-Block
1:	$H_i \mathrel{+}= \nabla^2_{x_i x_i} \phi_{ij}(x_i, u_j)$	$K_j \mathrel{+}= \nabla^2_{u_j u_j} \phi_{ij}(x_i, u_j)$	$J_i \mathrel{+}= \nabla^2_{u_j x_i} \phi_{ij}(x_i, u_j)$
2:	$H_j \mathrel{+}= \nabla^2_{x_j x_j} \phi_j(x_j)$		
	for $\tau : m$ **do**	**for** $\tau : m$ **do**	**for** $\tau : m$ **do**
3:	$H_i \mathrel{-}= \mu_\tau \nabla^2_{x_i x_i} f_{ij\tau}(x_i, u_j)$	$K_j \mathrel{-}= \mu_\tau \nabla^2_{u_j u_j} f_{ij\tau}(x_i, u_j)$	$J_i \mathrel{-}= \mu_\tau \nabla^2_{u_j x_i} f_{ij\tau}(x_i, u_j)$
4:	$H_j \mathrel{-}= \mu_\tau \nabla^2_{x_j x_j} f_{j\tau}(x_j)$		
	for $\tau : n^x_j$ **do**	**for** $\tau : n^x_j$ **do**	**for** $\tau : n^x_j$ **do**
5:	$H_i \mathrel{-}= \lambda_{j\tau} \nabla^2_{x_i x_i} g_{j\tau}(x_i, u_j)$	$K_j \mathrel{-}= \lambda_{j\tau} \nabla^2_{u_j u_j} g_{j\tau}(x_i, u_j)$	$J_i \mathrel{-}= \lambda_{j\tau} \nabla^2_{u_j x_i} g_{j\tau}(x_i, u_j)$
	for $\tau : l^{ru}_j$ **do**	**for** $\tau : l^{ru}_j$ **do**	**for** $\tau : l^{ru}_j$ **do**
6:	$H_i \mathrel{-}= v^u_{j\tau} \nabla^2_{x_i x_i} r_{ij\tau}(x_i, u_j)$	$K_j \mathrel{-}= v^u_{j\tau} \nabla^2_{u_j u_j} r_{ij\tau}(x_i, u_j)$	$J_i \mathrel{-}= v^u_{j\tau} \nabla^2_{u_j x_i} r_{ij\tau}(x_i, u_j)$
	for $\tau : l^{rx}_j$ **do**	**for** $\tau : l^{rx}_j$ **do**	**for** $\tau : l^{rx}_j$ **do**
7:	$H_j \mathrel{-}= v^x_{j\tau} \nabla^2_{x_j x_j} r_{j\tau}(x_j)$	–	–

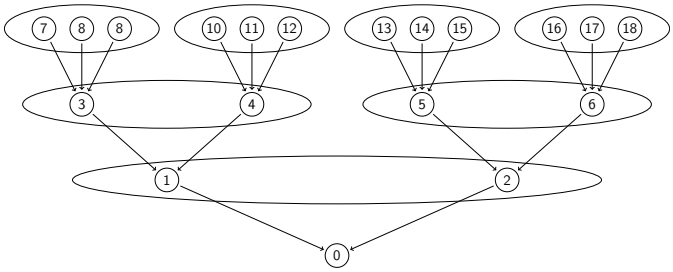


Figure 4.2.: Grouping of tree nodes for stable accumulation

More precisely, accumulated information are generated when

- evaluating the fill-in block X as well as modifying the right-hand side e in the basic recursions (cf. tables 4.5 and 4.7),
- evaluating the right-hand side e in the tree-sparse MVP (cf. tables 4.8 and 4.9)
- evaluating the values ν and e of the objective ϕ and the global constraints f of a TSP, respectively (cf. tables 4.10 and 4.12).

The approach presented next corresponds to an insertion method where the data are ordered with respect to the node probabilities. The way this is done is illustrated by means of the vector sum s of a tree-sparse primal variable vector $y = (y_j)_{j \in V}$,

$$s = \sum_{j \in V} s_j \quad \text{with} \quad s_j := \sum_{l=1}^{n_j^x} x_{jl} + \sum_{l=1}^{n_j^u} u_{jl}. \tag{4.166}$$

Recall from the stochastic background that each node $j \in V$ represents an event with an associated probability p_j and that the objective and global constraint node functions consume these probabilities (cf. Sect. 4.1.3). For each tree level, the probabilities add up to one (cf. (4.20)), leading to smaller probabilities with increasing tree levels. The tree-sparse problem data usually are scaled with the node probabilities and inherit their magnitude.

Now, evaluating the vector sum (4.166) is done as follows. In a first step, the node sums s_j are evaluated for each node $j \in V$. Afterwards, the successors $k \in S(j)$ of node j are grouped as shown in Fig. 4.2. Second, the intermediate sums s_k are added to the value s_j in an inward sweep over the tree, that is $s_j \mathrel{+}= \sum_{k \in S(j)} s_k$.

4.6.3. Problem Scaling

IPMs are significantly affected by scaling of the optimization problem. Some IPM implementations such as Clean::IPM [72] and Ipopt [105] provide for automatic problem scaling that

can be plugged in to improve the efficiency and robustness of the optimization algorithm. However, these general approaches are not tailored to specific optimization models such as the TSPs. Moreover, in the tree-sparse case, the specific approach of automatic row-scaling is even incompatible with the KKT solution algorithms (see discussion below).

Fortunately, tree-sparse problems are already well-scaled with the node probabilities p_j (cf. Sect. 4.1.3). The tree-sparse objective ϕ (4.21) and the global equality constraint function f (4.22), for example, are expected values, and the same applies to the Lagrangian $\mathcal{L}$ of TSPs as well. Therefore, the Hessian H of the Lagrangian and the global constraint block F of the tree-sparse Jacobian $\nabla c_{\mathcal{E}}$ experience a probability-based column-scaling. The same column-scaling can be achieved for the Jacobian $\nabla c_{\mathcal{R}}$ of the range constraints by multiplying the respective node-wise presentation of $c_{\mathcal{R}}$ with the node probabilities. For unscaled node functions, i.e. the dynamics as well the simple bounds, the node probabilities are consumed by the corresponding Lagrange multipliers.

Automatic Row-Scaling and Tree-Sparse KKT Solution

In Clean::IPM, the problem functions f, $c_{\mathcal{E}}$ and $c_{\mathcal{I}}$ are scaled such that all components of scaled gradients are less than or equal to a constant $\kappa_g > 0$ [72]. The scaled functions read

$$f \leftarrow \sigma_f f, \quad c_{\mathcal{E}} \leftarrow \Sigma_{\mathcal{E}} c_{\mathcal{E}} \quad \text{and} \quad c_{\mathcal{I}} \leftarrow \Sigma_{\mathcal{I}} c_{\mathcal{I}} \tag{4.167}$$

with diagonal matrices $\Sigma_{\mathcal{E}} \in \mathbb{R}^{m \times m}$ and $\Sigma_{\mathcal{I}} \in \mathbb{R}^{k \times k}$. The scaling factors are

$$\sigma_f = \min\left\{1, \frac{\kappa_g}{\|\nabla f\left(x^{(0)}\right)\|_\infty}\right\} \quad \text{and} \quad \sigma_i = \min\left\{1, \frac{\kappa_g}{\|\nabla c_i\left(x^{(0)}\right)\|_\infty}\right\} \quad \text{for} \quad i \in \mathcal{E} \cup \mathcal{I}. \tag{4.168}$$

Now, consider a scaled outgoing problem (4.3) where all scaling factors are consumed by the node functions. With the scaling matrices $\Sigma_j \in \mathbb{R}^{n_j^x \times n_j^x}$, the scaling

$$\{g_j(x_i, u_i) - x_j\} \leftarrow \{\Sigma_j g_j(x_i, u_i) - \Sigma_j x_j\} \tag{4.169}$$

of the dynamics (4.3b) affect their linearized versions (4.75c) in the KKT system (4.75) the following way:

$$\Sigma_j x_j = G_j x_i + E_j u_i + h_j, \quad j \in V. \tag{4.170}$$

The basic KKT recursion (cf. Table 4.5) is not designed for the scaled states $\Sigma_j x_j$ in (4.170).

However, altering the basic recursion, the scaling factors can be accounted for by replacing (4.88a) to (4.88c) with

$$\tilde{H}_j := H_j + \sum_{k \in S(j)} G_k^T \bar{H}_k \Sigma_k^{-1} G_k, \qquad \tilde{f}_j := f_j + \sum_{k \in S(j)} G_k^T \left(\bar{f}_k + \bar{H}_k \Sigma_k^{-1} h_k \right), \tag{4.171a}$$

$$\tilde{K}_j := K_j + \sum_{k \in S(j)} E_k^T \bar{H}_k \Sigma_k^{-1} E_k, \qquad \tilde{d}_j := d_j + \sum_{k \in S(j)} E_k^T \left(\bar{f}_k + \bar{H}_k \Sigma_k^{-1} h_k \right), \tag{4.171b}$$

$$\tilde{J}_j := J_j + \sum_{k \in S(j)} E_k^T \bar{H}_k \Sigma_k^{-1} G_k. \tag{4.171c}$$

Hence, the scaling effect of Σ_j is reversed, which arises doubts about scaling the dynamics in the first place.

Chapter 5

Distributed Tree-Sparse Optimization

This chapter deals with the distribution of the tree-sparse problems and algorithms. It provides the theoretical foundation for running computer programs using tree-sparse optimization in parallel computational environments and presents the distribution of the complete interior-point algorithm used for solving the TSPs. Parts of the presentations in this chapter are also about to be published in [48].

The approach of distribution exploits the node-wise formulations of problems and algorithms and is based on a static distribution of the tree nodes. Blomvall proposes to distribute these nodes in a natural order by applying a depth-first search (DFS) [10]. Thereby and by using reasonable computation orders for processing the distributed tree nodes, good parallel performance can be achieved. Following Blomvall's idea, this chapter formalizes the concept of *depth-first distributed trees* and presents theoretical results that are used to develop fitting iteration rules for them. Some of those results are already postulated by the author [50]. Now, the missing proofs are supplemented using the framework established around the distributed trees. Additionally, *distributed DFS-based tree algorithms* are introduced serving as models for distributing the tree-sparse algorithms.

This chapter is organized as follows. Section 5.1 motivates the approach of distribution and states the resulting distributed programming model. The concept of depth-first distributed trees and their theoretical results are presented in Sect. 5.2. Afterwards, Section 5.3 establishes the distributed DFS-based tree algorithms and concretizes the general discussions for these models to the tree-sparse KKT algorithms. Finally, Section 5.4 discusses the distribution of the complete IPM algorithm used for solving the TSPs. The terminology of graph theory and the respective notation used in this chapter is provided on demand, i.e. it is stated when it is used for the first time. For a comprehensive background on graph theory the reader is referred to standard textbooks such as [22].

5.1. Distributed Programming Model

The optimization problems studied in this work are stated with respect to the nodes of an underlying scenario tree. The dynamic constraints (4.3b) and (4.4b) of the TSPs (4.3) and (4.4), for example, are defined for each node $j \in V$, where each node constraint comprises a nonlinear *node function* (g_j) that depends on *node variables* (x_i, u_j, x_j). The same applies to the range constraints (4.3c)–(4.4c), (4.4d) and to the simple bounds (4.3d)–(4.3e), (4.4e)–(4.4f). Moreover, in the tree-sparse objectives (4.3a) and (4.4a) as well as in the global equality constraints (4.3f) and (4.4g), each node contributes nonlinear terms ($\phi_{ij}, \phi_j, f_{ij}, f_j$) to the respective accumulated values.

The tree-sparse algorithms are formulated in the same node-related way as the TSPs. Each algorithm is described by a set of so-called *node operations* that are applied in a certain order during a sweep of the tree nodes. For instance, evaluating the problem data of a TSP (see Sect. 4.6.1 and especially tables 4.10 to 4.13) includes evaluating the function values as well as respective first-order and second-order derivatives of all problem-defining node functions. Furthermore, Section 4.3 discusses the node operations for the tree-sparse KKT algorithms and arranges those in the dynamic algorithm tables 4.4 to 4.7.

In the following, let DA(j) denote the *set of data* that is associated with a node $j \in V$, and let OP(j) denote an algorithm-specific *set of node operations*. The items in a set DA(j) are the *node data* for node j. Usually, those node data are labeled with a respective node subscript (e.g. x_j, u_j) and exist for each node in the tree. Some data exists only once, e.g. the value ν of a tree-sparse objective and the fill-in matrix subblock X_V in the tree-sparse factorization. These *global data* are attributed to the tree root $0 \in V$ and supplemented to the set DA(0). In the dynamic algorithm tables, global data are always listed without any subscript (e.g. X instead of X_V). Then, node operations are usually performed for each node j in the tree, i.e. one type of node operation is listed in each set OP(j). So-called *global operations* form the exceptions. They involve only global data, are performed only once during a tree-sparse algorithm and are supplemented to the set OP(0) of the root. For instance, the Cholesky factorization $X_V = L_\emptyset L_\emptyset^T$ ($X \leftarrow LL^T$ in tables 4.5 and 4.7) is a global operation.

Now, the distributed programming model for tree-sparse optimization is as follows. In the parallel run of a computer program, the nodes of the tree are assigned to the participating *working units* (e.g. processes). Each working unit holds the data DA(j) of its assigned nodes and is responsible for performing the operations in the sets OP(j). All working units cooperate in a so-called *single-program-multiple-data* (SPMD) framework, i.e. they work asynchronously

in the same program on different parts of the program data, and they communicate with each other whenever data transmissions are necessary (see, e.g., [70]).

The distributed programming model is completely static. First, a given tree is split only one time and the nodes of the tree are distributed statically and uniquely among the working units. This way, the respective shares of the overall computational workload are fixed and the applied *node distribution* defines a partitioning of the program data. To save additional memory overhead and communication, dynamic rescheduling of data and workload are excluded, i.e. the nodes of a split tree are not reassigned to other working units. Thus, a working unit idles whenever it has completed its share of the workload and other working units are still working on theirs. Second, for each working unit, the order of processing the assigned nodes is also static, i.e. the order of performing the respective operations $\mathrm{OP}(j)$ is fixed. Therefore, a working unit also idles when it is working on $\mathrm{OP}(j_1)$ and waits for transmission of data from $\mathrm{DA}(j_2)$ of a node j_2 that is assigned to another working unit.

In this static programming model, the choice of the node distribution and the computation orders for the working units are mandatory for the parallel performance of the program. In this thesis, these are based on the depth-first strategy as proposed by Blomvall [10]. Section 5.2 discusses the node distribution, and Sect. 5.3 develops fitting computation orders. Moreover, a balanced distribution of the workload avoids the working units to idle due to lack in occupation. The tree-sparse models considered in this thesis feature corresponding tree-sparse algorithms that are well-balanced, i.e. for each algorithm, the amounts of work in the respective sets $\mathrm{OP}(j)$ are all about the same. Thus, in these cases, a uniform node distribution leads to a well-balanced distribution of the workload.

5.2. Depth-First Distributed Trees

Distributed trees result from splitting trees in several parts and distributing these parts among the working units. Basically, the parts of a split tree are glued together to form the distributed tree, which then comprises the respective tree parts and the information on how these fit together. In the following, Section 5.2.1 first clarifies basic graph terminology and states the used graph notation. It then introduces terminology for specific nodes and edges in distributed trees and concludes with the definition of depth-first distributed trees. Subsequently, Section 5.2.2 states and proves the theoretical results for this kind of trees. The concept of depth-first distributed trees are—to the best of the author's knowledge—not considered in the literature before[1].

[1] The concept of depth-first distributed trees are also going to be published in [48].

5.2.1. Distributed Trees

In graph theory, a graph comprises a set V of *nodes* or *vertices* and a set E of *edges* that contains pairs of then to be said *adjacent* nodes. Each of two adjacent nodes $a, b \in V$ is *incident* on the edge $(a, b) \in E$, and the edge (a, b) is also said to be incident on both nodes a and b. The edges (a, b) and (b, a) coincide in *undirected* graphs and they do not in *directed* graphs. *Paths* of *length* l are sequences $(a_1, \ldots, a_l)$ of adjacent nodes $(a_{k-1}, a_k) \in E$ for $k = 2, \ldots, l$, which connect the comprising nodes a_k. In a *connected* graph, all nodes in the set V are connected with each other. The connected parts of a graph are called *components*. Hence, a connected graph comprises a single component. *Cycles* are specific paths $(a_1, \ldots, a_l)$ that have coinciding endpoints $a_1 = a_l$, and a graph without cycles is called *acyclic*. Now, a *forest* is an acyclic undirected graph and a *tree* is a connected forest. For each two nodes a and b in a tree, there is a unique path $(a, \ldots, b)$ connecting those nodes. Its length defines the *distance* between its two endpoints a and b.

The trees $T = (V, E)$ that are about to be split have numbered sets of nodes $V = \{0, \ldots, n\}$ and are so-called rooted out-trees with root 0. In a *rooted tree* one node is dedicated as the root, which then induces a direction on the tree edges. For a node $j \in V$ of a rooted tree T, the *level* $t(j)$ refers to the distance between this node and the root, i.e. the level is the length of the path $\Pi(j) = (j, ..., 0)$ from node j to root 0. The *ancestors* of node j are those nodes on the path $\Pi(j)$ that feature a lower tree level. The immediate ancestor $\pi(j)$ is called the *predecessor*, i.e. the predecessor is the node $\pi(j) \in \Pi(j)$ with $t(\pi(j)) = t(j) - 1$. Only the root has no predecessor, meaning it is the only node with $\pi(0) = \emptyset$. Being an ancestor for one node means having the same node as *descendant*. Immediate descendants of a node j are called *successors* and are gathered in its *set of successors* $S(j)$. Nodes $l \in L$ have no successors and form the *leaves* of the tree T, i.e. $L = \{l \in V \colon S(l) = \emptyset\}$.

The direction of the tree edges induced by the root is here chosen to point from the root to the leaves. Hence, the set E of such an *out-tree* T comprises only those edges (a, b) with $t(a) = t(b) - 1$, i.e. node a is the predecessor $\pi(b)$ of the adjacent endpoint b. The edge (a, b) is an *inedge* for node b and an *outedge* for its predecessor a. Thus, for a node $j \in V$, the set of edges E contains the inedge $(\pi(j), j)$ incident on its predecessor $\pi(j)$, and E contains the outedges (j, k) incident on its successors $k \in S(j)$.

The set of nodes $V(T) = \{0, \ldots, n\}$ of tree T is numbered so that T is said to be post-ordered. For this, let T_v be the *subtree* of T rooted at $v \in V(T)$, i.e. T_v is the subgraph that is induced by node v and all its descendants in $V(T)$. More generally, a *subgraph* $\bar{G}$ of a graph $G = (V, E)$ *induced* by the set of nodes $\bar{V} \subseteq V$ contains all edges $(a, b) \in E$ of adjacent nodes a, b in the

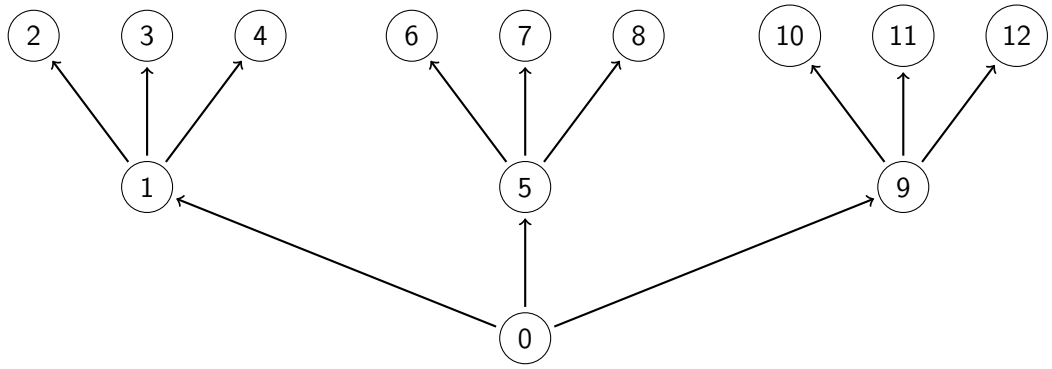


Figure 5.1.: A post-ordered out-tree

subset $\bar{V}$, i.e. $(a, b) \in \bar{E}$. Now, the tree T is *post-ordered* if for any node $v \in V(T)$ the set of nodes of the subtree T_v is $V(T_v) = \{v, v+1, \ldots, v+n_v-1\}$, where $n_v = |V(T_v)|$ is the number of nodes of the subtree.

In the following, a post-ordered out-tree T with numbered node set $V(T) = \{0, \ldots, n\}$ and root 0 (see Fig. 5.1) is split and distributed among q working units. For this, let the set of nodes $V(T)$ be partitioned into q parts, i.e. $V(T) = \bigcup_{p=1}^{q} V_p$ with nonempty and pairwise disjoint subsets $V_p \subseteq V(T)$. The subgraph induced by the set of nodes V_p is a forest and will be denoted by $\mathcal{F}_p = (V_p, E_p)$. The components of a forest are again trees. To distinguish the split tree T from the components of a forest, the latter will henceforth be denoted using the calligraphic letter $\mathcal{T}$.

For a node $v \in V_p$, the path $\Pi(v)$ remains the path to the root 0 of the tree T and $t(v)$ still refers to its level in T. The subtrees $\mathcal{T}_r^p$ in the forest $\mathcal{F}_p$ are rooted in the canonical way induced by the tree T, meaning that the root r is the unique node with lowest tree level $t(r)$ in $\mathcal{T}_r^p$. The roots in the forest $\mathcal{F}_p$ are gathered in the *set of roots* $\mathcal{R}_p$ and the union $\mathcal{R} = \bigcup_{p=1}^{q} \mathcal{R}_p$ denotes the *set of all roots*.

If the tree T is distributed among more than one working unit, i.e. if $q > 1$, the union of all forest edges E_p is a strict subset of the set of tree edges E. *Loose edges* are those edges in E that do not belong to any set E_p. They are denoted by $\mathcal{E}$, i.e. $\mathcal{E} = E \setminus \bigcup_{p=1}^{q} E_p$. For a forest $\mathcal{F}_p$, the subset $\mathcal{E}_p \subseteq \mathcal{E}$ comprises those loose edges that are incident on a node in V_p. Any loose edge $(s, r) \in \mathcal{E}$ is an inedge for a root $r \in \mathcal{R} \setminus \{0\}$. The node s adjacent to root r is called a *sender*, i.e. a sender is incident on at least one loose outedge $(s, \cdot) \in \mathcal{E}$. For a forest $\mathcal{F}_p$, the set $\mathcal{S}_p$ denotes the respective *set of senders* and the union $\mathcal{S} = \bigcup_{p=1}^{q} \mathcal{S}_p$ is the *set of all senders*.

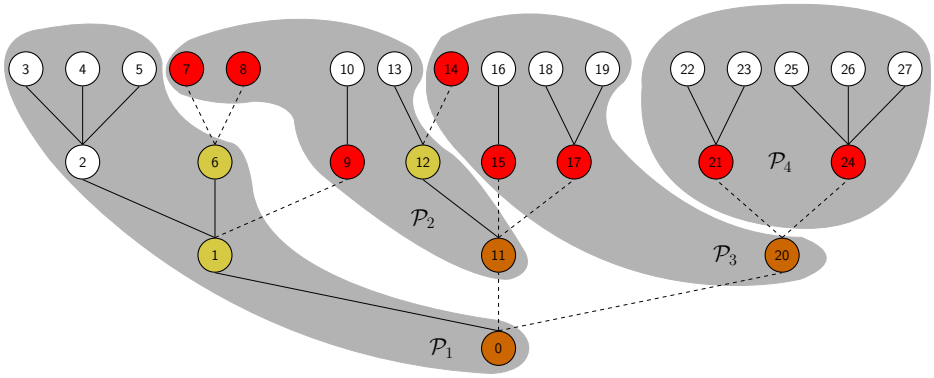


Figure 5.2.: A depth-first distributed tree consisting of four equally sized parts $\mathcal{P}_1$ to $\mathcal{P}_4$ with the respective roots (red nodes), the senders (yellow nodes) as well as those nodes that are both (orange nodes)

The definition of a distributed tree combines the established terminology.

Definition 9 (Distributed Tree)**.** *Let $T = (V, E)$ be an out-tree with the numbered node set $V = \{0, \dots, n\}$. Furthermore, let $V = \bigcup_{p=1}^{q} V_p$ be a node partitioning of the tree nodes. A* distributed tree *is then the collection*

$$\mathcal{D} = \{T, \mathcal{P}_1, \dots, \mathcal{P}_q, \mathcal{E}, \mathcal{R}, \mathcal{S}\} \quad \textit{with} \quad \mathcal{P}_p = \{\mathcal{F}_p, \mathcal{E}_p, \mathcal{R}_p, \mathcal{S}_p\} \quad \textit{for} \quad p = 1, \dots, q, \tag{5.1}$$

where each part *$\mathcal{P}_p$ consists of the node-induced subgraph $\mathcal{F}_p$ and the corresponding loose edges $\mathcal{E}_p$, roots $\mathcal{R}_p$ and senders $\mathcal{S}_p$. The collection $\mathcal{D}$ is said to be a* distribution *of the tree T.*

Finally, the following definition introduces the specific distributed tree type that is studied in this thesis. The name of this kind of distributed tree is motivated by the depth-first search, which, when properly used for numbering the nodes $V(T)$, induces a post-order on the tree T (cf. Sect. 5.3.1).

Definition 10 (Depth-First Distributed Tree)**.** *Let $T = (V, E)$ be a post-ordered out-tree with numbered node set $V = \{0, \dots, n\}$. A* depth-first distributed tree *is a distribution of T resulting from an* ascending node partitioning *with respect to the node numbering, i.e.*

$$V = \bigcup_{p=1}^{q} V_p \quad \textit{with} \quad V_p = \left\{ \sum_{k=1}^{p-1} |V_p|, \dots \sum_{k=1}^{p} |V_p| - 1 \right\}. \tag{5.2}$$

Figure 5.2 shows a depth-first distributed tree consisting of four equally sized parts, and Table 5.1 lists the respective sets of roots, senders and loose edges. Note that for a part $\mathcal{P}_p$, the sets of roots and senders are not necessarily disjoint, i.e. a node $v \in V_p$ can be incident

Table 5.1.: Data of the depth-first distributed tree in Fig. 5.2

p	$\mathcal{R}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$
1	0	0,1,6	$(0,11),(0,20),(1,9),(6,7),(6,8)$
2	7,8,9,11	11,12	$(6,7),(6,8),(1,9),(0,11),(11,15),(11,17),(12,14)$
3	14,15,17,20	20	$(0,20),(11,15),(11,17),(12,14),(20,21),(20,24)$
4	21,24	—	$(20,21),(20,24)$

on both a loose inedge $(\cdot, v) \in \mathcal{E}_p$ and a loose outedge $(v, \cdot) \in \mathcal{E}_p$. In Fig. 5.2, for example, node $11 \in V_2$ is both a root and a sender.

5.2.2. Properties of Depth-First Distributed Trees

After distributing a tree T, the loose edges in its distribution $\mathcal{D}$ connect the distributed tree parts $\mathcal{P}_p$ beyond the scopes of the respective working units. In terms of distributed programming, communication is taking place along the loose edges $\mathcal{E}$. The roots $\mathcal{R} \setminus \{0\}$ as well as the senders $\mathcal{S}$ are those nodes that possibly cause idle times. Studying the loose edges and their incident nodes in depth-first distributed trees is the basis for developing iteration rules for the respective tree parts (cf. Sect. 5.3.2). In the following, four theorems make conclusions about the locations of senders and roots in depth-first distributed tree parts and about how the senders of one part and the roots of a second part are related to each other. These theorems require some preparations, starting with properties that hold for any post-ordered tree.

The first proposition results directly from the notation introduced in Sect. 5.2.1. It is stated here for later reference.

Proposition 1. *For any two nodes v_1, v_2 in a tree T with $v_1 \neq v_2$ it holds:*

(a) $v_1 \in \Pi(v_2)$ if and only if $v_2 \in V(T_{v_1})$.

(b) $t(v_1) = t(v_2)$ implies $v_1 \notin V(T_{v_2})$ and $v_1 \notin \Pi(v_2)$.

The subsequent propositions and the first lemma state general properties of post-ordered trees with the first one of them resulting directly from their definition. These properties are used frequently to prove the theoretical results of depth-first distributed trees.

Proposition 2. *For each node v in a post-ordered tree T it holds:*

(a) $v < j$ for all nodes j in the node set $V(T_v) \setminus \{v\}$.

(b) $j < v$ for all nodes j on the path $\Pi(v) \setminus \{v\}$.

Proof. The first claim follows with $j \in V(T_v) = \{v+1, \ldots, v+n_v-1\}$ and from the definition of the post-order. The second claim follows from (a) by applying Prop. 1(a), that is $v \in V(T_j)$. □

Proposition 3. *Let v_1, v_2 be two nodes in a post-ordered tree T with $v_1 < v_2$ and $v_2 \notin V(T_{v_1})$. Then $j < v_2$ holds for all nodes $j \in V(T_{v_1})$.*

Proof. Since T is post-ordered, the node set of T_{v_1} is given by

$$V(T_{v_1}) = \{v_1, \dots, v_1 + n_{v_1} - 1\}.$$

With $v_2 > v_1$ and $v_2 \notin V(T_{v_1})$ it follows that $v_2 \geq v_1 + n_{v_1} > j$ for all $j \in V(T_{v_1})$. □

Lemma 1. *Let v_1, v_2 be two nodes in a post-ordered tree T with the predecessors $\pi_1 = \pi(v_1)$ and $\pi_2 = \pi(v_2)$. Also, let the nodes have the order $v_1 < v_2$. If $\pi_1 \neq \pi_2$ holds, exactly one of the following two statements is true:*

(a) $\pi_2 \in \Pi(\pi_1)$ or

(b) the predecessors have the same order $\pi_1 < \pi_2$.

Proof. Suppose (a) holds. Then Prop. 2(b) implies the order $\pi_2 < \pi_1$ and, hence, (b) cannot be true. Thus, it is sufficient to verify that $\pi_2 \notin \Pi(\pi_1)$ implies (b). This can be shown by contradiction. Assuming $\pi_2 \notin \Pi(\pi_1)$ and $\pi_2 < \pi_1$, Proposition 2(a) provides $\pi_1 \notin V(T_{\pi_2})$ and from Prop. 3 it can be concluded that $j < \pi_1 < v_1$ for all nodes $j \in V(T_{\pi_2})$. With $v_2 \in V(T_{\pi_2})$ this implies the order of $v_2 < v_1$ which contradicts the assumption $v_1 < v_2$. Hence, $\pi_2 \notin \Pi(\pi_1)$ implies $\pi_1 < \pi_2$. □

In the following, the results of depth-first distributed trees are established. For this, the function $P(v) = p$ is used to map a node $v \in V$ on its distributed tree part $\mathcal{P}_p$. The first proposition results directly from the definitions of roots and senders.

Proposition 4. *For a depth-first distributed tree $\mathcal{D}$ the following three statements are true:*

(a) $P(v_1) \leq P(v_2)$ for any two nodes $v_1, v_2 \in V$ with $v_1 < v_2$.

(b) $P(\pi(r)) < P(r)$ for any root $r \in \mathcal{R} \setminus \{0\}$.

(c) There is a node $j \in S(s)$ with $P(s) < P(j)$ for any sender $s \in \mathcal{S}$.

Proof. Claim (a) is true by definition. For a root $r \in \mathcal{R} \setminus \{0\}$ it is that $\pi(r) < r$ due to the post-order. Hence, $P(\pi(r)) \leq P(r)$ follows from (a). Moreover, it is $P(\pi(r)) \neq P(r)$ or otherwise r would be as descendant of $\pi(j)$ in $V(\mathcal{T}_{\pi(r)}^{P(r)})$, which contradicts to r being a root. Hence, (b) is true. For any sender $s \in \mathcal{S}$ there is by definition at least one root $r \in \mathcal{R} \setminus \{0\}$ such that $\pi(r) = s$. Thus, $r \in S(s)$. Therefore, (c) follows from (b). □

The following two lemmata concentrate on the mapping $P(\cdot)$ of senders and their descendants in the distributed tree.

Lemma 2. *Let $\mathcal{D}$ be a depth-first distributed tree and $s \in \mathcal{S}_p$. For any node $v \notin V(T_s)$ with $v > s$ it holds $P(v) > P(s)$.*

Proof. According to Prop. 4(c) there is a node $j_1 \in S(s) \subseteq V(T_s)$ with $P(j_1) > P(s)$. With $v \notin V(T_s)$ Prop. 3 provides $v > j$ for all nodes j in the subtree T_s. With Prop. 4(a) it is $P(v) \geq P(j_1) > P(s)$. □

Lemma 3. *Let $\mathcal{D}$ be a depth-first distributed tree. Let the node v be on the part $\mathcal{P}_p$ and $s \in \mathcal{S}_p$ with $s \neq v$ and $t(v) = t(s)$. Then the nodes v and s suffice the order $v < s$ and it is $v \notin \mathcal{S}_p$.*

Proof. The level assumption $t(v) = t(s)$ yields $v \notin V(T_s)$ with Prop. 1(b). Applying Lemma 2, the order $v > s$ implies $P(v) > P(s)$, which contradicts the assumption $P(v) = P(s)$. Hence, $v < s$ must hold. The same argumentation leads to the contradiction $P(v) < P(s)$ if $v \in \mathcal{S}_p$. □

Now, the first theorem states that all senders of the same part of a distributed tree lie on the same path to the root 0 of the tree T.

Theorem 4 (Sender-Path)**.** *Let $\mathcal{D}$ be a depth-first distributed tree and $s_1, s_2 \in \mathcal{S}_p$ with $s_1 < s_2$. Then s_1 lies on the path from s_2 to the root, i.e. $s_1 \in \Pi(s_2)$.*

Proof. Let $s_1, s_2 \in \mathcal{S}_p$ with $s_1 < s_2$. Assume that $s_1 \notin \Pi(s_2)$ holds, which is equivalent to $s_2 \notin V(T_{s_1})$ according to Prop. 1(a). Lemma 2 states $P(s_2) > P(s_1)$, contradicting $P(s_2) = P(s_1)$. Hence, $s_1 \in \Pi(s_2)$ holds. □

This theorem leads to the following definitions.

Definition 11 (Sender-Path, Sender-Subtree and Sender-Root)**.** *Let $\mathcal{D} = (T, \mathcal{P}_1, \ldots, \mathcal{P}_q, \mathcal{R}, \mathcal{S})$ be a depth-first distributed tree. The* sender-path *of part $\mathcal{P}_p = (\mathcal{F}_p, \mathcal{E}_p, \mathcal{R}_p, \mathcal{S}_p)$ is given by*

$$\Pi^p_{\mathcal{S}} := \{v \in V_p \colon \textit{it exists } s \in \mathcal{S}_p \textit{ with } v \in \Pi(s)\}\,. \tag{5.3}$$

The sender-root $r^p_{\mathcal{S}}$ *is the start of the sender-path, i.e. the sender-root is characterized by the property*

$$t(r^p_{\mathcal{S}}) = \min\{t(v) \colon v \in \Pi^p_{\mathcal{S}}\}\,. \tag{5.4}$$

The sender-subtree $\mathcal{T}^p_{\mathcal{S}}$ *is the subtree in the forest $\mathcal{F}_p$ that roots at the sender-root.*

Clearly, the sender-root is indeed a root, i.e. $r^p_{\mathcal{S}} \in \mathcal{R}_p$, or otherwise its predecessor $\pi(r^p_{\mathcal{S}})$ would be on the same part and take over the pole position in the sender-path $\Pi^p_{\mathcal{S}}$.

The second theorem considers two roots on the same part of the distributed tree that are also on the same tree level. It states that they share a common predecessor.

Theorem 5 (Root's Predecessors)**.** *Let $\mathcal{D}$ be a depth-first distributed tree. Any two roots $r_1, r_2 \in \mathcal{R}_p$ on the same tree level $t(r_1) = t(r_2)$ have a common predecessor $\pi(r_1) = \pi(r_2)$.*

Proof. For $r_1 = r_2$ nothing is to show. Thus, consider the case $r_1 \neq r_2$ and set $\pi_1 = \pi(r_1)$ and $\pi_2 = \pi(r_2)$. Assume that $\pi_1 \neq \pi_2$ and without loss of generality let $r_1 < r_2$. The relation $t(r_1) = t(r_2)$ implies $t(\pi_1) = t(\pi_2)$ and with Prop. 1(b) it can be concluded that π_2 neither lies on the path $\Pi(\pi_1)$ nor is in the subtree T_{π_1}. Applying Lemma 1 gives $\pi_1 < \pi_2$ and with Prop. 3 it holds $j < \pi_2$ for all $j \in V(T_{\pi_1})$. Since $r_1 \in V(T_{\pi_1})$, Prop. 4 leads to the order $P(\pi_1) < P(r_1) \leq P(\pi_2) < P(r_2)$, contradicting the assumption of $P(r_1) = P(r_2)$. Hence, $\pi_1 = \pi_2$ holds. □

The next theorems state relations between the root levels and node numbers on the distributed tree parts and highlight the special case of the sender-root.

Theorem 6 (Descending Root Levels)**.** *Let $\mathcal{D}$ be a depth-first distributed tree. For any two roots $r_1, r_2 \in \mathcal{R}_p$ with $r_1 \leq r_2$ it is $t(r_1) \geq t(r_2)$.*

Proof. The claim is true for $r_1 = r_2$. Therefore, consider the case $r_1 < r_2$ and assume $t(r_1) < t(r_2)$. With $\pi_1 = \pi(r_1)$ and $\pi_2 = \pi(r_2)$ the order $t(\pi_1) < t(\pi_2)$ holds and, thus, $\pi_2 \notin \Pi(\pi_1)$. Analogously to the proof of Thm. 5, Lemma 1 can be used to obtain $\pi_1 < \pi_2$, and Prop. 3 together with Prop. 4 leads to the order $P(\pi_1) < P(r_1) \leq P(\pi_2) < P(r_2)$ that again contradicts $P(r_1) = P(r_2)$. Hence, $t(r_1) \geq t(r_2)$ holds. □

Theorem 7 (Sender-Root Level)**.** *Let $\mathcal{D}$ be a depth-first distributed tree. The sender-root has (if defined) the lowest tree level on $\mathcal{P}_p$, that is*

$$t(r_{\mathcal{S}}^p) = \min\{t(v) \colon v \in V_p\} \tag{5.5}$$

Proof. It is sufficient to show that $r_{\mathcal{S}}^p$ has the minimum tree level among the roots $\mathcal{R}_p$ since for $v \notin \mathcal{R}_p$ it is $\pi(v) \in V_p$ and $t(\pi(v)) < t(v)$. For $|\mathcal{R}_p| = 1$ there is nothing to show. For $|\mathcal{R}_p| > 1$ consider a root $r \in \mathcal{R}_p$ with $r \neq r_{\mathcal{S}}^p$ and assume that $t(r) < t(r_{\mathcal{S}}^p)$. From Thm. 6 it can be concluded that $r > r_{\mathcal{S}}^p$. Since r is a root it is $r \notin V(\mathcal{T}_{\mathcal{S}}^p) \subseteq V(T_{r_{\mathcal{S}}^p})$. With $V(T_s) \subseteq V(T_{r_{\mathcal{S}}^p})$ for any sender $s \in \mathcal{S}_p$, Lemma 2 can be applied to obtain the contradiction $P(r) > P(r_{\mathcal{S}}^p)$. Hence, $t(r) \geq t(r_{\mathcal{S}}^p)$ is true. □

Corollary 1. *Let $\mathcal{D}$ be a depth-first distributed tree, assume that the sender-root $r_{\mathcal{S}}^p$ exists and suppose that there is another root $r \in \mathcal{R}_p$ with $r \neq r_{\mathcal{S}}^p$ and $t(r) = t(r_{\mathcal{S}}^p)$. Then $r < r_{\mathcal{S}}^p$ holds.*

Proof. The argument for this proof is already presented in the proof of Thm. 7. Since $r \in \mathcal{R}_p$ is a root it cannot be in the subtree $T_{r_{\mathcal{S}}^p}$. With the help of Prop. 3 and Prop. 4, the assumption of $r > r_{\mathcal{S}}^p$ leads to the contradiction $P(r) > P(r_{\mathcal{S}}^p)$. Hence, $r < r_{\mathcal{S}}^p$ holds. □

5.3. Distributed DFS-Based Tree Algorithms

In this section, algorithms are developed that iterate over each part of a depth-first distributed tree such that the idle times are minimized. First, the tree-sparse algorithms are abstracted to the so-called *DFS-based tree algorithms* (Sect. 5.3.1). Afterwards, the properties of depth-first distributed trees are used to conclude the iteration rules for their distributed counterparts (Sect. 5.3.2). These properties are further exploited to save communication overhead in the distributed performance of a DFS-based tree algorithm by performing the *post-distribution communication reduction* (PDCR) (Sect. 5.3.3). Finally, the previous general discussions for the *distributed DFS-based tree algorithms* are concretized by returning to the tree-sparse algorithms and presenting distributed versions of the tree-sparse KKT algorithms (Sect. 5.3.4). The models of the distributed DFS-based tree algorithms, the PDCR as well as the distributed tree-sparse algorithms are—to the best of the author's knowledge—not considered in the literature[2].

5.3.1. DFS-Based Tree Algorithms

In the following, two algorithms based on the depth-first strategy are considered serving as models for the tree-sparse algorithms. The first one of these *DFS-based tree algorithms* processes the nodes of the tree in an inward sweep from treetop to the root. In opposition to this *inward algorithm*, the second so-called *outward algorithm* processes the nodes in an outward sweep by starting at the root and then going up in the tree. These two types of tree algorithms are motivated by the KKT algorithms (cf. Sect. 4.3), which take center stage in generalizing the tree-sparse algorithms. In contrast to the other tree-sparse algorithms (cf. Sect. 4.6.1), the KKT algorithms require certain orders of node processing and performing the operations for each node in the tree. For stability reasons, those requirements are also applied to the other tree-sparse algorithms. Moreover, every algorithm but the outward substitution is treated as an inward algorithm (cf. Sect. 4.6.2).

Subsequently, the characteristics of the tree-sparse KKT algorithms are highlighted by classifying the operations as well as the involved data, and by reviewing the prescribed computation orders. Terminology in the context of processing the nodes of a tree is given next. Finally, this section concludes with the definition of the DFS-based tree algorithms.

[2]Parts of these concepts and algorithms are going to be published in [48].

Classification of Data and Operations

In the context of distributed tree-sparse optimization, a second useful classification of operations and data—besides *common* and *global* (cf. Sect. 5.1)—is necessary. For a node $j_1 \in V$, the node data in the corresponding set $\mathrm{DA}(j_1)$ is referred to as *local data*. *Nonlocal data* subsume the data in the sets $\mathrm{DA}(j_2)$ of other nodes $j_2 \neq j_1$. Global data are local for operations in the set $\mathrm{OP}(0)$ and nonlocal for the operations in any other set $\mathrm{OP}(j)$ with $j \in V \setminus \{0\}$. Accordingly, a node operation is said to be *local* if it involves only local data in its performance. *Nonlocal operations* are then those in a set $\mathrm{OP}(j_1)$ that also involve nonlocal data from a set $\mathrm{DA}(j_2)$ of at least one other node $j_2 \neq j_1$. Global operations count as local operations in the set $\mathrm{OP}(0)$.

By taking a closer a look at the nonlocal operations, e.g. by examining item 9 of the incoming control version of the basic recursion in Table 4.5,

$$H_i \mathrel{-}= J_j^T J_j, \quad f_i \mathrel{-}= J_j^T d_j \quad \text{and} \quad u_j \mathrel{+}= J_j x_i, \tag{5.6}$$

the classification of nonlocal operations is refined as follows. Operations that use local data to modify nonlocal data ($H_i \mathrel{-}= J_j^T J_j$, $f_i \mathrel{-}= J_j^T d_j$) are called *nonlocal write operations*. *Nonlocal read operations* refer to the operations that modify local data and involve nonlocal data in the performance ($u_j \mathrel{+}= J_j x_i$). Note that due to the way the dynamic algorithm tables are designed, the nonlocal operations in a set $\mathrm{OP}(j)$ involve aside from local data only nonlocal data from the set $\mathrm{DA}(0)$ of the tree root 0 or nonlocal data from the set $\mathrm{DA}(i)$ of its predecessor $i = \pi(j)$.

Computation Orders for Nodes and Operations

Each tree-sparse KKT algorithm prescribes a certain computation order on the node operations to ensure a correct overall performance of the respective algorithm. First and independent of the specific KKT algorithm, each set of operations is completed at once, i.e. the node operations in one set $\mathrm{OP}(j_1)$ are executed one after another without applying operations of another set $\mathrm{OP}(j_2)$ in between. Second, the factorization as well as the inward substitution require the nodes to be processed in an inward sweep over the tree, i.e. these algorithms start at the leaves L and end with processing the root 0. This means that the operations in one set $\mathrm{OP}(j)$ for a node $j \in V$ are performed only after the sets $\mathrm{OP}(k)$ of all its successors $k \in S(j)$ are completed. The outward substitution processes the nodes in an outward sweep over the tree, meaning node j is processed only after the set $\mathrm{OP}(\pi(j))$ of its predecessor $\pi(j)$ is completed. Finally, the order of the node operations in one set $\mathrm{OP}(j)$ is also prescribed by the KKT algorithms. In an inward sweep over the tree, the operations in one set $\mathrm{OP}(j)$ are performed from top to bottom ($\downarrow$) as

listed in the respective tables whereas, in an outward sweep, they are performed from bottom to top (↑).

Traversals and Event Points

A *node iteration* over a tree $T = (V, E)$ defines a finite sequence $\{v_l\}_{l=1,\dots,K}$ of nodes within which the nodes are *visited*. An iteration over T is said to be *complete* if each node $v \in V$ is visited at least once. *Traversals* are methods for traversing the nodes of a tree. They induce complete node iterations for which any two consecutive nodes are adjacent. By convention, traversals of rooted out-trees always start and end at the root $0 \in V$. During such a tree traversal, a node $v \in V \setminus \{0\}$ is said to be *discovered* when it is visited through its inedge for the first time. The node is *finished* after all its successors are discovered and v is then left for the first time through its inedge again. Now, the root 0 is discovered at the start of the tree traversal and the root is finished when it is visited the next time after all successors $S(0)$ are discovered. Finally, the tree T is said to be finished when all its nodes are finished and the root is visited for the last time.

The *depth-first search* is a specific tree traversal. It discovers all nodes on a path $\Pi(l)$ from the root 0 to a leaf $l \in L$ first, and then finishes these nodes on its way back to the root. After discovery, a node $v \in V$ is finished only after finishing all its successors $S(v)$. Hence, the depth-first search completes one branch of the tree before it it turns towards the next. These DFS-traversals are also used to constitute an order on a tree. For example, a post-order is established by assigning numbers to the tree nodes when finishing them.

For distributions $\mathcal{D}$ of the tree T, the events of discovery and finishing are extended the following way. A node $v \in V_{p_1}$ is still discovered when it is visited for the first time. The node v is finished after discovering all its successors $k \in S(v)$ including those in other sets of nodes V_{p_2} and v is then left through its inedge again. A subtree $\mathcal{T}_r$ is discovered with the discovery of the root r, and it is finished when all nodes $v \in V(\mathcal{T}_r)$ are finished and r is visited for the last time. The forest $\mathcal{F}_p$ is discovered with the discovery of the first root in $\mathcal{R}_p$ and finished with finishing the last subtree $\mathcal{T}_r^p$. Finally, the distribution $\mathcal{D}$ is said to be finished when the same applies for the tree T.

DFS-Based Tree Algorithms

Now returning to the tree-sparse algorithms, the requirements regarding the order of processing the tree nodes are realized using depth-first traversals over rooted out-trees as follows. For processing the nodes in an outward sweep, the operations in the set $\mathrm{OP}(j)$ are performed upon discovery of the node $j \in V$. Hence, the node operations are applied in direction of the tree

edges. In realizing an inward sweep, on the other hand, the operations in $\mathrm{OP}(j)$ are performed when finishing node j, which means that node operations are applied in opposite direction of the tree. This leads to the following definition of models for the tree-sparse algorithms.

Definition 12 (DFS-Based Tree Algorithms)**.** *Let $T = (V, E)$ be post-ordered out-tree. A* DFS-based tree algorithm *for the tree T consists of a set $\mathrm{OP}(j)$ of local and nonlocal operations for each node $j \in V$ and a depth-first traversal over T. In an* outward algorithm*, the operations in $\mathrm{OP}(j)$ are applied upon discovery of node j. In an* inward algorithm*, the operations in $\mathrm{OP}(j)$ are applied when finishing the node j.*

5.3.2. Distributed Tree Algorithms

Distributed tree algorithms, in general, refer to algorithms that iterate over the parts of a distributed tree (cf. Sect. 5.2.1). *Distributed DFS-based tree algorithms*, in specific, iterate over the parts of a depth-first distributed tree $\mathcal{D}$ keeping the occurring idle times at a minimum. They avoid idle times by priorizing those roots and senders in $\mathcal{D}$ that generate data for other working units while postponing the processing of those nodes in $\mathcal{R} \cup \mathcal{S}$ to the latest possible time that require data from other working units. The priorities of roots and senders in depth-first distributed trees are identified based on the theoretical results presented in Sect. 5.2.2.

Basically, a distributed DFS-based tree algorithm is a distributed extension of a sequential counterpart. Each working unit p iterates over the subtrees in the forest of its part $\mathcal{P}_p$ and applies the underlying outward or inward algorithm to each subtree $\mathcal{T}_r^p$ in $\mathcal{F}_p$, i.e. each subtree is traversed in a depth-first manner and the operations in $\mathrm{OP}(j)$ are executed either upon discovery or when finishing the node j, respectively. In doing so, the overall node computation order requirements are guaranteed. Clearly, communication routines are invoked whenever data need to be transmitted via loose edges.

Now, there are two flexibilities in the local computation order of a working unit p. First, the iteration order over the subtrees in the forest $\mathcal{F}_p$ does not affect the proper performance of the tree algorithm. Second, there are several equivalent depth-first traversals leading to different node orders. For example, each subtree $\mathcal{T}_r$ inherits a post-order from the depth-first traversal used to number the nodes and, hence, to distribute the tree T. This order will be called the *induced order*. Furthermore, any order of the tree T induces an order $l_1 < \ldots l_{|L|}$ on its leaves $l_k \in L$. The unique order that leads to the reverse leaf order $l_{|L|} < \cdots < l_1$ will be referred to as the *reverse order*. Note that reverse orders of post-orders are again post-orders. The choice of the depth-first traversal for each subtree is also free.

Both flexibilities are used to reduce the idle times in the distributed performance of a DFS-

based tree algorithm. First, for the inward algorithm, the basic rule is to process the senders $s \in \mathcal{S}$ at the latest possible time since they require nonlocal data from roots assigned to other working units for their execution. Therefore, the sender-subtree $\mathcal{T}_{\mathcal{S}}^{p}$ is processed at last from the respective working unit p. Lemma 3 provides that nodes with a lower number have a higher priority. This is realized by applying the induced computation order to the sender-subtree. For the remaining subtrees, the computation order does not affect the idle times. Hence, there is no harm in using the induced order as well. Applying the node operations in opposite direction to the rooted out-tree T implies that for two senders $s_1, s_2 \in \mathcal{S}_p$ with $t(s_1) < t(s_2)$ the node s_2 is processed first and, therefore, the nonlocal data in the sets $\mathrm{DA}(r)$ from the adjacent roots $r \in S(s_2) \cap \mathcal{R}$ are required earlier. Theorems 6 and 7 together with Prop. 1 allow the conclusion that the subtrees $\mathcal{T}_r$ with lower root numbers have higher priorities.

For the outward algorithm, the computation order criteria are exactly opposite, i.e. the senders are processed at the earliest possible time and the subtrees with lower tree level are priorized since the nonlocal data they require is sooner available. This leads to the following models for the distributed tree-sparse algorithms.

Definition 13 (Distributed DFS-Based Tree Algorithms)**.** *Let $\mathcal{D}$ be a depth-first distributed tree. A* distributed DFS-based tree algorithm *consists of an iteration rule over the subtrees $\mathcal{T}_r^p$ in $\mathcal{F}_p$ for each part $\mathcal{P}_p$ and a DFS-based tree algorithm that is applied to each subtree $\mathcal{T}_r$. An* inward algorithm *iterates over the subtrees in ascending order with respect to the root number and the nodes of each subtree are processed in the order induced by the tree distribution. An* outward algorithm *iterates over the subtrees in descending order with respect to the root number and the nodes of each subtree are processed in the reverse of the induced order.*

5.3.3. Post-Distribution Communication Reduction

Consider the distribution $\mathcal{D}$ of a rooted out-tree T. After node distribution, the sets of roots $\mathcal{R}$ and senders $\mathcal{S}$ are fixed, i.e. the node distribution determines the set of loose edges $\mathcal{E}$ and, in doing so, it also determines the information that is transmitted between the working units during the run of a distributed tree algorithm. However, while the information to be exchanged is fixed, the number of invoked communication routines to realize the data exchange is not. Each invocation of a communication routine causes overhead in the computational costs of the program.

In the following, the *post-distribution communication reduction* (PDCR) aims for saving communication calls during the distributed run of a DFS-based tree algorithm without producing additional idle time in the performance. The PDCR exploits common properties shared by

the nonlocal operations in a set $\mathrm{OP}(j)$. Therefore, recall from Sect. 5.3.1 the classification of operations and the involved data into local and nonlocal ones as well as the refinement of nonlocal operations into nonlocal read and nonlocal write operations. Considering item 4 of the outward substitution of the basic recursion in Table 4.7, for example, this item has as nonlocal read operation the form

$$x_{j_1} \mathrel{+}= G_{j_1} x_i,\ x_{j_2} \mathrel{+}= G_{j_2} x_i \quad \text{for} \quad j_1, j_2 \in V \quad \text{with} \quad \pi(j_1) = \pi(j_2) = i. \tag{5.7}$$

Two nodes j_1 and j_2 with the same predecessor i need the same nonlocal data in $\mathrm{DA}(i)$ for their operations. On the other hand, nonlocal write operations of the same two nodes modify the same nonlocal data in $\mathrm{DA}(i)$ additively. Item 4 of the factorization in Table 4.7, for example, has the form

$$H_i \mathrel{+}= G_{j_1}^T H_{j_1} G_{j_1},\ H_i \mathrel{+}= G_{j_2}^T H_{j_2} G_{j_2} \quad \text{for} \quad j_1, j_2 \in V \quad \text{with} \quad \pi(j_1) = \pi(j_2) = i, \tag{5.8}$$

and item 9 can be written as

$$H_i \mathrel{+}= -\left(J_{j_1}^T J_{j_1}\right),\ H_i \mathrel{+}= -\left(J_{j_2}^T J_{j_2}\right) \quad \text{for} \quad j_1, j_2 \in V \quad \text{with} \quad \pi(j_1) = \pi(j_2) = i. \tag{5.9}$$

These properties of nonlocal operations are exploited by the PDCR the following way. Consider two roots $r_1, r_2 \in \mathcal{R}_{p_1}$ of a working unit p_1 with the same predecessor $\pi(r_1) = \pi(r_2) = s \in \mathcal{S}_{p_2}$. For the nonlocal read operation (5.7), the data x_s in the set $\mathrm{DA}(s)$ is transmitted from working unit p_1 to unit p_2. Since the nonlocal operations in $\mathrm{OP}(r_1)$ and $\mathrm{OP}(r_2)$ need the same data, it is sufficient to send x_s only once to p_1. For the nonlocal write operations (5.8), the working unit p_1 first accumulates the data to

$$\bar{H}_s = G_{r_1}^T H_{r_1} G_{r_1} + G_{r_2}^T H_{r_2} G_{r_2}, \tag{5.10}$$

and then sends the accumulation $\bar{H}_s$ to the unit p_2. This way, the PDCR reduces the communication calls to one. Informally speaking, the loose edges $(s, r_1), (s, r_2) \in \mathcal{E}$ are merged to one. The reduction of the PDCR is expandable to $k > 2$ roots in $\mathcal{R}_p$ with the same predecessor.

The PDCR causes no additional idle time by reducing the number of communication calls. In the nonlocal write operation (5.8), for example, the sender s needs the data from the roots r_1 and r_2 at the same time. Hence, waiting for the data of one or both of the roots or waiting for the accumulation $\bar{H}_s$ is all the same for node s. For the nonlocal read operation (5.7), the

PDCR even reduces the idle time. When the working unit p_1 processes the second of the both roots r_1 and r_2, the required data x_s is already available. Thus, p_1 does not idle while waiting for completion of a communication routine.

Now, considering distributed DFS-based tree algorithms and depth-first distributed trees again, the properties of the latter (cf. Sect. 5.2.2) even abet the PDCR. Theorem 5 provides that two roots $r_1, r_2 \in \mathcal{R}_p$ have the same predecessor *whenever* they are on the same tree level. Together with the root level order provided by Thm. 6, the communication between two working units during a distributed DFS-based tree algorithm is limited to at most one communication call per tree level.

5.3.4. Distributed Tree-Sparse KKT Algorithms

This section returns to the tree-sparse algorithms and applies the models of the distributed DFS-based tree algorithms (cf. Sect. 5.3.2) to the tree-sparse KKT algorithms (cf. Sect. 4.3). For both control cases, distributed versions of the complete KKT algorithms are presented. Additionally, a distributed version of the matrix-vector product (MVP) with the tree-sparse KKT matrix in outgoing control form is discussed (cf. Sect. 4.6.1). Distributing a tree-sparse MVP features an additional aspect of communication that does not arise in the distribution of the tree-sparse KKT algorithms.

Distributed Versions of the Tree-Sparse KKT Algorithms

For the distributed versions of the tree-sparse KKT algorithms, the first phase of eliminating the range duals is now placed on top of the respective basic recursion. The complete distributed tree-sparse KKT algorithm in the outgoing control form is listed in the single Table 5.2. This algorithm table subsumes tables 4.4 and 4.5 and includes new operations for the incurrent communication. It maintains the prior numbering of the items and labels those of the first KKT phase with a prime (items $1'$ to $4'$). The new operations of communication are numbered with small Roman numerals (items i to iv).

The following description of the communication in Table 5.2 takes the PDCR (cf. Sect. 5.3.3) already into account. For this, recall the events of discovering and finishing nodes, subtrees and forests in a distribution $\mathcal{D}$ of an out-tree T (cf. Sect. 5.3.1). In each stage of the KKT solution, the communication of items i and ii are invoked for each sender in $\mathcal{S}$ whereas the items iii and iv are only performed for some of the roots in $\mathcal{R} \setminus \{0\}$. During the inward algorithms of the KKT solution procedure, i.e. the factorization and the inward substitution, each sender $j \in \mathcal{S}$ first receives data (e.g. H_j^p, f_j^p) from one or several other working units ($\curlyvee$) when finishing j.

Table 5.2.: Distributed tree-sparse KKT algorithm – Outgoing control case

	Factorization ↓	Inward Subst. ↓	Outward Subst. ↑
i:	$\curlyvee\ H_j^p, K_j^p, J_j^p$		
ii:	$\curlyvee\ F_j^p, D_j^p, X^p$	$\curlyvee\ f_j^p, d_j^p, e^p$	$\uparrow\uparrow\ x_j, u_j, \mu$
1′:			$-v_j \leftarrow \Psi_j(-v_j)$
2′:	$H_j \mathrel{+}= F_j^{rT}\Psi_j F_j^r + \Phi_j^x$	$f_j \mathrel{+}= F_j^{rT}\Psi_j r_j$	$-v_j \mathrel{+}= F_j^r x_j$
3′:	$K_j \mathrel{+}= D_j^{rT}\Psi_j D_j^r + \Phi_j^u$	$d_j \mathrel{+}= D_j^{rT}\Psi_j r_j$	$-v_j \mathrel{+}= D_j^r u_j$
4′:	$J_j \mathrel{+}= D_j^{rT}\Psi_j F_j^r$		
1:	$K_j \leftarrow L_j L_j^T$		
2:	$D_j \leftarrow L_j^{-1} D_j$	$d_j \leftarrow L_j^{-1} d_j$	$u_j \leftarrow -L_j^{-T} u_j$
3:	$J_j \leftarrow L_j^{-1} J_j$		$u_j \mathrel{+}= J_j x_j$
4:	$H_j \mathrel{-}= J_j^T J_j$	$f_j \mathrel{+}= J_j^T d_j$	$-\lambda_j \mathrel{+}= H_j x_j$
5:	$F_j \mathrel{-}= D_j J_j$		
6:	$X \mathrel{+}= D_j D_j^T$	$e \mathrel{-}= D_j d_j$	
7:	$H_i \mathrel{+}= G_j^T H_j G_j$	$f_i \mathrel{+}= G_j^T(f_j + H_j h_j)$	$x_j \mathrel{+}= G_j x_i$
8:	$K_i \mathrel{+}= E_j^T H_j E_j$	$d_i \mathrel{+}= E_j^T(f_j + H_j h_j)$	$x_j \mathrel{+}= E_j u_i$
9:	$J_i \mathrel{+}= E_j^T H_j G_j$		
10:	$F_i \mathrel{+}= F_j G_j$	$e \mathrel{+}= F_j h_j$	$-\lambda_j \mathrel{+}= F_j^T(-\mu)$
11:	$D_i \mathrel{+}= F_j E_j$	$h_j \leftrightarrow f_j$	$u_j \mathrel{+}= D_j^T(-\mu)$
iii:	$\downarrow\ H_i, K_i, J_i$	$\downarrow\ f_i, d_i, e$	$\uparrow\ x_i, u_i, -\mu$
iv:	$\downarrow\ F_i, D_i, X$		
12:	$X \leftarrow LL^T$	$e \leftarrow L^{-1} e$	$-\mu \leftarrow L^{-T}(-\mu)$

The received data is accumulated to the respective local data in DA(j) (e.g. $H_j \mathrel{+}= \sum_p H_j^p$, $f_j \mathrel{+}= \sum_p f_j^p$) before continuing with the node operations in OP(j). Second, after finishing all roots $r \in \mathcal{R}_p \setminus \{0\}$ with $i = \pi(r)$, the nonlocal modifications (e.g. H_i) are ready for transmission (items *iii* and *iv*). When finishing the last subtree $\mathcal{T}_r^p$ with $i = \pi(r)$, these data are sent in inward direction (↓) to the predecessor i. Hence, communication invoked during an inward KKT algorithm is directed opposite to the direction of the tree edges. On the other hand, communication invoked during an outward KKT algorithm is carried out in direction of the out-tree T. Upon discovery of a subtree $\mathcal{T}_r^p$, the working unit p checks whether the required nonlocal data (x_i, u_i, μ) are already available. If not, these data are received (↑) from the predecessor $i = \pi(r)$ (item *iii*). After completing the operations OP(j) for a sender $j \in \mathcal{S}_p$, the working unit p initiates the transmission of data to the other working units (item *ii*). The respective data (x_j, u_j, μ) are sent upon discovery of j in outward direction (↑↑) to each working unit requiring them. To accumulate the global data (X, e, μ) in a numerically stable way as described in Sect. 4.6.2, they are communicated the same way as common node data. That is

Table 5.3.: Distributed tree-sparse KKT algorithm – Incoming control case

	Factorization $\downarrow$	Inward Subst. $\downarrow$	Outward Subst. $\uparrow$
i:	$\curlyvee\ H_j^p, F_j^p, X^p$	$\curlyvee\ f_j^p, e^p$	$\Uparrow\ x_j, \mu$
$1'$:			$-v_j^u \leftarrow \Psi_j^u(-v_j^u)$
$2'$:			$-v_j^x \leftarrow \Psi_j^x(-v_j^x)$
$3'$:	$K_j \mathrel{+}= D_j^{r\,T}\Psi_j^u D_j^r + \Phi_j^u$	$d_j \mathrel{+}= D_j^{r\,T}\Psi_j^u r_j^u$	$-v_j^u \mathrel{+}= D_j^r u_j$
$4'$:	$J_j \mathrel{+}= D_j^{r\,T}\Psi_j^u F_{ij}^r$		
$5'$:	$H_i \mathrel{+}= F_{ij}^{r\,T}\Psi_j^u F_{ij}^r$	$f_i \mathrel{+}= F_{ij}^{r\,T}\Psi_j^u r_j^u$	$-v_j^u \mathrel{+}= F_{ij}^r x_i$
$6'$:	$H_j \mathrel{+}= F_j^{r\,T}\Psi_j^x F_j^r + \Phi_j^x$	$f_j \mathrel{+}= F_j^{r\,T}\Psi_j^x r_j^x$	$-v_j^x \mathrel{+}= F_j^r x_j$
1:	$K_j \mathrel{+}= E_j^T H_j E_j$	$d_j \mathrel{+}= E_j^T(H_j h_j + f_j)$	$-\lambda_j \mathrel{+}= H_j x_j$
2:	$K_j \leftarrow L_j L_j^T$		
3:	$J_j \mathrel{+}= E_j^T H_j G_j$		$x_j \mathrel{+}= E_j u_j$
4:	$H_i \mathrel{+}= G_j^T H_j G_j$	$f_i \mathrel{+}= G_j^T(H_j h_j + f_j)$	$x_j \mathrel{+}= G_j x_i$
5:	$F_i \mathrel{+}= F_j G_j$	$e \mathrel{+}= F_j h_j$	$-\lambda_j \mathrel{+}= F_j^T(-\mu)$
6:	$D_j \mathrel{+}= F_j E_j$		
7:	$D_j \leftarrow D_j L_j^{-T}$	$h_j \leftrightarrow f_j$	
8:	$J_j \leftarrow L_j^{-1} J_j$	$d_j \leftarrow L_j^{-1} d_j$	$u_j \leftarrow -L_j^{-T} u_j$
9:	$H_i \mathrel{-}= J_j^T J_j$	$f_i \mathrel{-}= J_j^T d_j$	$u_j \mathrel{+}= J_j x_i$
10:	$F_i \mathrel{-}= D_j J_j$		
11:	$X \mathrel{+}= D_j D_j^T$	$e \mathrel{-}= D_j d_j$	$u_j \mathrel{+}= D_j^T(-\mu)$
ii:	$\downarrow\ H_i, F_i, X$	$\downarrow\ f_i, e$	$\uparrow\ x_i, -\mu$
12:	$X \leftarrow LL^T$	$e \leftarrow L^{-1}e$	$-\mu \leftarrow L^{-T}(-\mu)$

although eventually destined for the tree root 0, the data X^p and e^p (item ii) run through the entire communication process of the inward algorithm.

Distributing the tree-sparse KKT algorithm in the incoming control form proceeds completely analogously to the outgoing control form (cf. Sect. 5.3.4). The distributed version of the incoming control case is listed in Table 5.3 and subsumes the range elimination phase in Table 4.6 as well as the basic recursion in Table 4.7.

Distributed Version of the MVP Algorithm in Outgoing Control Form

Unlike the factorization and the substitution of the tree-sparse KKT algorithms, computing the MVP with a tree-sparse KKT matrix as well as evaluating the problem data for a TSP are realized by inward algorithms that also include nonlocal read operations (cf. Sect. 4.6.1). The distributed version of the tree-sparse MVP in the outgoing control case, listed in Table 5.4, exemplifies the distribution of these inward algorithms. For a node $j \in V$, items 1 and 2 of the equality constraint block A are nonlocal read operations requiring the state variables x_i

Table 5.4.: Distributed tree-sparse MVP algorithm – Outgoing control case

	H-Block	A-Block	A^T-Block	B-Block	B^T-Block
i:		$\Uparrow x_j, u_j$	$\Uparrow \mu$		
iii:		$\curlyvee e^p$	$\curlyvee f_j^p, d_j^p$		
1:	$f_j \mathrel{+}= H_j x_j$	$h_j \mathrel{+}= G_j x_i$	$f_j \mathrel{-}= \lambda_j$	$r_j \mathrel{+}= F_j^r x_j$	$f_j \mathrel{+}= {F_j^r}^T v_j$
2:	$f_j \mathrel{+}= J_j^T u_j$	$h_j \mathrel{+}= E_j u_i$	$f_i \mathrel{+}= G_j^T \lambda_j$	$r_j \mathrel{+}= D_j^r u_j$	$d_j \mathrel{+}= {D_j^r}^T v_j$
3:	$d_j \mathrel{+}= K_j u_j$	$h_j \mathrel{-}= x_j$	$d_i \mathrel{+}= E_j^T \lambda_j$		
4:	$d_j \mathrel{+}= J_j x_j$	$e \mathrel{+}= F_j x_j$	$f_j \mathrel{+}= F_j^T \mu$		
5:		$e \mathrel{+}= D_j u_j$	$d_j \mathrel{+}= D_j^T \mu$		
iii:		$\downarrow e$	$\downarrow f_i, d_i$		

as well as the control variables u_i of the predecessor $i = \pi(j)$ to modify the local data h_j. Furthermore, items 4 and 5 of the transpose A^T modify local data using the global multiplier μ. Therefore, the distributed performance of this inward algorithm requires communication that is in direction of the tree and directed opposite to the algorithm.

For communicating in outward direction during an inward algorithm ($\Uparrow$), it would be ill-advised to initiate the respective communication calls (item i in Table 5.4) at the events of nodes, i.e. upon discovery of a sender $s \in \mathcal{S}$ or when finishing s. This communication strategy would lead to unnecessary idle times since the inward algorithm dictates processing the senders at the latest possible time. Instead, to provide the required nonlocal data (x_i, u_i, μ) as soon as possible, each working unit p initiates the data transmission upon discovery of the forest $\mathcal{F}_p$. With appropriate designed loops over the roots $\mathcal{R}_p$ and over the senders $\mathcal{S}_p$, the working units complete the outward communication before starting with the respective inward algorithm, which then proceeds analogously to the incoming KKT algorithms.

5.4. Distributed Solver for Tree-Sparse Problems

This section presents the distribution of the complete interior-point algorithm used for solving the TSPs. For this, recall from Sect. 2.2 the discussion of solution algorithms for NLPs in general and the one of IPMs in specific. The primal-dual filter line-search interior-point algorithm is composed of the following four types of operations.

Problem-specific operations: These operations depend on the problem-specific structures of the optimization problem. Computing (2.21) of the primal-dual search direction $\Delta(x, s, z, v)^{(k)}$, i.e. evaluating the solution of a KKT system, is the most obvious operation of this type. Further examples are matrix-vector products with parts of the KKT matrix as well

as evaluations of the optimization model at hand. Each of these three operations is problem-specific since their performance depends on the structure of the KKT matrix.

Vector-valued vector operations: These operations use input arguments such as vectors or scalars to generate the resulting vector in an element-wise manner. The addition of two vectors, for example, is a vector-valued vector operation of the form $\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$. Others such as the scalar multiplication feature the form $\mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$. In the IPM algorithm, for example, the updates (2.24) of the iterates (x, s) and (z, v) are composed of the previous two vector-valued vector operations.

Collective operations: These operations generate a single information from one or several vectors. First, logical-valued collective operations answer questions that are either true or false, e.g. a check of the element-wise relation $v \geq 0$ for a vector v is answered this way. Scalar-valued vector operations generate a single scalar from one or several vectors. Examples for scalar-valued vector operations are norms and the scalar product. The evaluation of the KKT error (2.20) is based on vector norms.

Scalar operations: These operations generate a single scalar from one or several other scalars. For example, the update heuristic for the barrier parameter μ^{bp} in the IPM algorithm is a scalar operation. However, updates rules for μ^{bp} such as the rule of Mehrotra's predictor-corrector heuristic [65] may be based on more complex operations.

Additionally to those four types of operations, the IPM algorithm contains loops and junction points based on logical decisions (if-clauses). In an SPMD programming model, all working units cooperate in one single program, which means that there is an instance of the same program for each working unit. The SPMD model dictates that each working unit runs through the same loops in its respective instance and takes the same branching at each junction point. Clearly, this is achieved if the decisions in all program instances are based on the same values.

Now, the IPM algorithm is distributed by distributing each of the four operation types. For this, problem-specific operations are performed by tree-sparse algorithms realized as distributed DFS-based tree algorithms (cf. Sect. 5.3). Second, scalar operations are simply duplicated in each instance of the program. Furthermore, unit-specific shares of vector-valued vector operations are independent of each other and, therefore, performed asynchronously by each working unit. Finally, the collective operations require a collaboration of the working units for the single information they generate. In the distributed case, each working unit typically creates a scalar or a logical value of its assigned vector part on its own and all these values are finally combined into a single information. Collective operations usually involve a so-called *reduce-and-scatter* communication routine. The values of the working units are first brought

together on one dedicated instance of the program and the merged result is then sent back to the other working units. In tree-sparse optimization, this dedicated program instance corresponds to the working unit that is responsible for the tree root 0. However, for accumulating scalar-valued collective operations such as the scalar product, this reduce-and-scatter procedure does not apply. For stability reasons these accumulations are also realized by distributed DFS-based tree algorithms (cf. Sect. 4.6.2).

When dealing with NLPs, the distribution of the IPM extends the following way. First, the filter line-search globalization (cf. Sect. 2.2.2) is a scalar operation that is based on several collective operations. The filter (Def. 8) comprising pairs of scalars to keep track of the progress of the IPM algorithm is duplicated for each program instance. The SPMD programming model ensures that each instance establishes the same filter without additional synchronization beyond the one that is carried out by the collective operations. Second, the tree-sparse inertia correction is already distributed since it is incorporated into the tree-sparse KKT solution algorithms (cf. Sect. 4.4). Finally, the tree-sparse Hessian update strategies in the proposed quasi-Newton approach (cf. Sect. 4.5) are highly problem-specific and, therefore, realized by tree-sparse algorithms.

Chapter 6

Software Design

This chapter presents the design of the software that is used to solve the tree-sparse problems. Sophisticated data structures are designed and plugged in the interior-point solver Clean::IPM developed by Schmidt [72]. The interior-point framework Clean::IPM is part of Clean (short for *A C++ Library for Efficient Algorithms in Numerics*) that is developed in the working group *Algorithmic Optimization* of Steinbach at the Leibniz Universität Hannover. The software framework Clean, which is intended to become public domain when it is considered to be sufficiently mature [72], is a generic C++ library that contains flexible algorithms primarily for optimization and the solution of linear systems. The provided algorithms feature a generic design based on C++ templates that makes them independent of the used data objects. Those data objects implement operations on the behalf of the algorithm and, in doing so, mask the specific implementation of the problem data they hold. This allows the instantiation of a problem-tailored setup of an algorithm achieving runtime efficiency by compile-time decisions based on the used data objects.

Taking advantage of the flexible algorithm design in Clean, sophisticated data objects tailored to the tree-sparse problems are developed and then used in Clean::IPM. These tree-sparse objects are banded together in a C++ library that will be referred to as the *Tree-Sparse Library* (TSL). The tree-sparse objects and tree-sparse operations are based on the concepts of distributed trees and distributed DFS-based tree algorithms (cf. Chap. 5) that are implemented in the *Distributed Tree Environment*[1]. The latter C++ library is intended to become part of Clean and will be referred to as Clean::DTE.

Fundamental software concepts applied in Clean set an software framework that is of vital importance for the TSL. Outlines of the most important concepts, i.e. those that are crucial for design choices made by the author, are given in the first two sections of this chapter. The

[1]The name of the library is inspired by the software framework *Parallel Tree Environment* developed by Hofmann [46].

author wants to emphasize that the concepts in Clean go back to Steinbach and the outlines given here are neither complete nor final, i.e. they may vary upon publication of Clean.

Now, this chapter is organized as follows. Section 6.1 sets the software framework for Clean::DTE and the TSL. The subsequent section focuses on the distribution of data objects and algorithms in Clean in general and in Clean::IPM in specific. The design of Clean::DTE is introduced in Sect. 6.3 and Sect. 6.4 discusses the design of the TSL.

6.1. Software Framework

This section sets the software framework for the designs of Clean::DTE (cf. Sect. 6.3) and the TSL (cf. Sect. 6.4). The framework includes the programming language and the use of external libraries (Sect. 6.1.1), the most used generic programming techniques (Sect. 6.1.2), basic ideas of Clean (Sect. 6.1.3) and details on the IPM solver Clean::IPM (Sect. 6.1.4).

6.1.1. C++ and External Libraries

The software is written in C++ satisfying mostly the C++11 standard but also employing some features of the current C++14 standard [95]. The software implementation requires the C++ standard library and the Boost C++ libraries (version 1.56.0 or higher) [94]. Mathematical operations are based on BLAS [55] and LAPACK [3]. Communication is based on the Message Passing Interface (MPI) [96]. The communication routines are called through the Boost.MPI C++ interface and carried out by the MPI implementation Open MPI (version 1.65.0) [97].

6.1.2. Generic Programming Techniques

The three subsequently described generic C++ techniques using templates are used extensively in the design of Clean::DTE and the TSL.

Policies and Traits

Many classes are implemented following a *policy-based class design pattern* and use *type traits* to define the set of relevant types for the class.

The policy-based class design pattern is introduced by Alexandrescu [2] and can be seen as a compile-time variant of the *Strategy Pattern* [28]. A policy defines the template interface for a class and the selection of a specific policy determines its behavior. Type traits are programming techniques that allow compile-time decisions based on types instead of making runtime decisions

Listing 6.1: Example for the policy-traits design

```
1 template<class Widget_Policy>
2 struct Widget_Traits
3 {
4   using A = typename Widget_Policy::A;
5   using B = std::ostream;
6   using C = typename A::Traits::C;
7   using D = E<Widget_Policy>;
8 };
9
10 template<class Widget_Policy>
11 class Widget
12 {
13   using Traits = Widget_Traits<Widget_Policy>;
14   using A      = typename Traits::A;
15   using B      = typename Traits::B;
16   using C      = typename Traits::C;
17   using D      = typename Traits::D;
18 };
```

based on values. Type traits are used extensively in some generic libraries, e.g. Boost [94] and the C++ *Standard Template Library* (STL) [90, 99].

In the developed software, the policy-traits design is defined as follows. A policy class encapsulates the free type choices for a policy-based class. The policy-based class uses a traits class to obtain all types required in its scope. The traits class used by a policy-based class shares the same template interface, i.e. it is instantiated by the same policy. An example of the policy-traits design is shown in Lst. 6.1.

Types defined in policies and traits classes are complete, i.e. they are fully instantiated. Incomplete types may be passed as additional template parameters to policy-based classes and, thus, form extensions to the template interfaces of the latter. An incomplete template parameter usually requires the full instantiation of the parameterized policy-based class or its traits class.

Template Base Classes and CRTP

To avoid code duplication, common groups of functionalities are encapsulated in their own classes labeled with the suffix `_Base`. Inheriting from a class `X_Base` provides the derived class with the functionalities of `X_Base`. The `X_Base` class is not parameterized by a policy but by the traits of the inheriting class. Hence, the traits class of the derived class must also define all types that are required by the `X_Base` class. To allow more flexibility in the class designs, deriving classes may also be parameterized by the `X_Base` class through a template parameter.

Some of the base classes employ the *Curiously Recurring Template Pattern* (CRTP) [18], which is similar to the *Barton-Nackmann-Trick* [6]. The CRTP design combines templates and inheritance in such a way that the template hierarchy is directed opposite to the inheritance

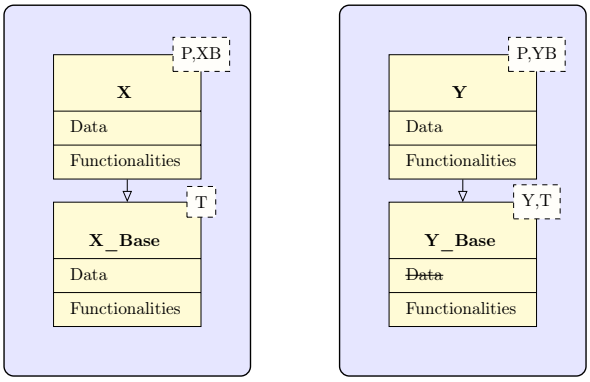


Figure 6.1.: Inheritance diagrams for non-CRTP class `X_Base` (left) and CRTP class `Y_Base` (right) with the template parameters `Policy` (`P`), `Traits` (`T`), `X_Base` (`XB`), `Y_Base` (`YB`) and the class `Y`

Listing 6.2: Tag dispatching in function overloading

```
1  struct Valuable {};
2  struct Trash    {};
3  struct New_Car      { using Tag = Valuable; };
4  struct Accident_Car { using Tag = Trash; };
5
6  template<class X>
7  void give_away(X& x) { give_away(x, typename X::Tag()); }
8
9  template<class X>
10 void give_away(X& x, Valuable tag) { sell(x); }
11
12 template<class X>
13 void give_away(X& x, Trash tag) { dump(x); }
```

hierarchy. A class `Y` inherits the CRTP class `Y_Base` and passes itself in complete instantiation as template parameter to `Y_Base`. In this design, the `Y_Base` is not allowed to have its own data members. Again, the `Y_Base` class may also be a template parameter for the derived class. Figure 6.1 shows *Unified Modeling Language* (UML) [100] diagrams for deriving classes that inherit from CRTP and non-CRTP base classes.

Tag Dispatching

Tag dispatching is a generic programming technique used for function overloading to dispatch based on properties of a type [15]. A *tag* is an empty structure that is attached as type definition to a class `X` and used by a function to delegate the instance `x` of the class `X` to the proper function overload. An example of tag dispatching is outlined in Lst. 6.2.

6.1.3. Basic Ideas of Clean

The guiding idea of Clean is to provide numerical algorithms that are independent of specific data types. The user of such a numerical algorithm (e.g. the author) may replace standard data types by problem-tailored ones without affecting the logic of the numerical scheme and, therefore, without the need to reimplement the algorithm. This independency of data types requires a strict separation of responsibilities, which is in Clean as follows.

An *algorithm* implements the logic of a numerical scheme and employs *data objects* to carry out mathematical operations. A data object represents a mathematical object such as a vector or a matrix, and it provides a set of mathematical operations. The interaction between algorithms and data objects is defined by fixed *interfaces*. Algorithms use these interfaces to instruct the data objects with the required operations. The latter are then responsible for performing the operations.

Clearly, an algorithm runs only then correctly, i.e. it performs the logic of the implemented numerical scheme as intended, if the used data objects harmonize with each other. For example, operations requiring two or more different kinds of objects are defined by only one of these. The operation-defining object and the others must have common agreements to interact with each other, e.g. member methods, memory management and interfaces. Definitions of data objects that are required to harmonize with each other are grouped in *servers*. Basically, each algorithm has one server that defines the data objects in use. Furthermore, the server—and not necessarily the objects—provides the required interfaces for its corresponding algorithm.

An example for an algorithm in Clean is the interior-point framework Clean::IPM, which is discussed in Sect. 6.1.4.

Vectors

Vectors are specific data objects in Clean and take center stage in the design of distributed algorithms (cf. Sect. 6.2). In the vector design, the data of a vector, i.e. its content, is strictly separated from its structural information such as the vector length. The data elements of a vector are numerical values, e.g. real numbers, and the data is stored in a single data array, i.e. a contiguous memory block. The vector interface provides typical vector operations like the vector addition and the scalar product as well as direct element access through the subscript operator (`[]`).

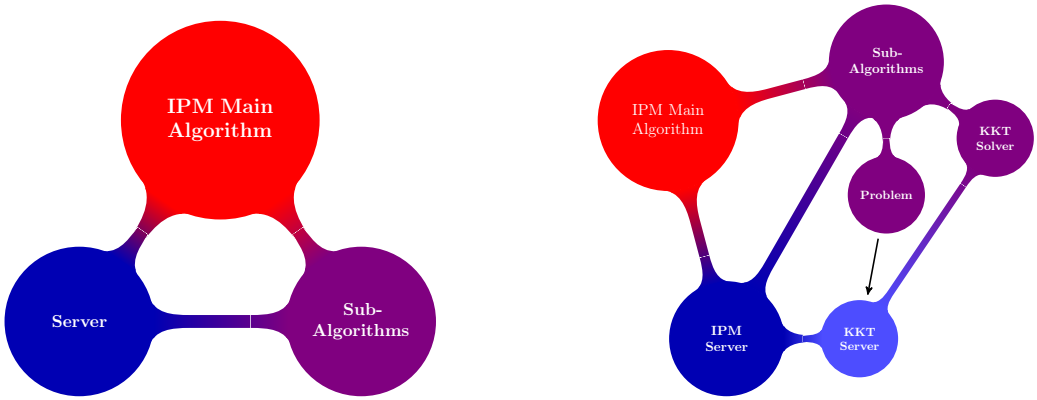


Figure 6.2.: Basic design of Clean::IPM (left) and refined design showing the incorporation of the KKT server (right)

6.1.4. The Interior-Point Solver Clean::IPM

Clean::IPM is a flexible interior-point framework for different classes of problems which flexibility is realized by two key features. First, in a modular design using building blocks, the user has free choices of some sublogics of the IPM algorithm. Second, as an algorithm in Clean, the IPM framework separates strictly between algorithm logics and data objects, and it dictates the interaction between those through fixed interfaces (cf. Sect. 6.1.3).

The basic design of Clean::IPM is shown on the left-hand side in Fig. 6.2. It comprises the main algorithm, the subalgorithms and the server. The main algorithm implements the overall algorithmic logic, which is an infeasible primal-dual interior-point method. A subalgorithm implements a sublogic of the IPM such as the update rule for the barrier parameter or the solution of the KKT system. The server provides the interfaces to the data objects used in the main algorithm and the subalgorithms. For a more detailed depiction of the software design of Clean::IPM the reader is referred to [72].

Clean::IPM is developed to solve different classes of problems. Its modular design allows an user to assemble an algorithm that works best for her problem at hand. In this thesis, Clean::IPM is used to solve smooth nonlinear nonconvex optimization problems. In [73], for example, Schmidt employs his IPM framework to solve optimization problems with locatable and separable nonsmooth aspects.

To solve the nonlinear tree-sparse problems efficiently, highly sophisticated data objects exploiting the specific structures of the tree-sparse problems are plugged in Clean::IPM (cf. Chap. 4 and Sect. 3.2). The designs of these problem-tailored tree-sparse objects are presented in Sect. 6.4. For knowing the requirements on them and the way how to plug them in, one needs a closer look at the design of Clean::IPM and the occurring interfaces therein.

Specific Servers and Interfaces

Generally, the IPM main algorithm and its subalgorithms have the same server, i.e. the IPM server. Two subalgorithms form exceptions to this design. First, the KKT solver handling the solution of the KKT system has its own server. This KKT server is a member of the IPM server and provides the KKT solver interface and a KKT matrix interface for operations with the KKT matrix such as a matrix-vector product. The second exception is the implementation of the optimization problem, which is responsible for evaluating the required optimization problem data including first-order and second-order derivatives. It provides the problem interface for evaluation requests and does not have an own server but, instead, it gains access to the KKT server. Evaluated problem data are stored directly into the KKT server without using a fixed interface. For this, the KKT server is passed to the problem through its problem interface. This direct data access couples the problem implementation to the KKT server in use. In a refined design, the diagram on the right-hand side in Fig. 6.2 shows the incorporation of the KKT server.

Besides the three specific interfaces for the KKT solver, the KKT matrix as well as the problem, two additional types of interfaces are used in Clean::IPM. The first type is formed by the interfaces of the IPM server for all subalgorithms except the ones for the KKT solver and the problem. Details about these interfaces are not relevant for the design of the tree-sparse objects. The second type are vector interfaces (cf. Sect. 6.1.3) that are provided by each vector. Clean::IPM accesses any vector consistently through its interface, which is a key feature for distributing the algorithm (cf. Sect. 6.2.3).

KKT Objects and IPM Objects

A server is responsible for defining the used data objects that are required to harmonize with each other (cf. Sect. 6.1.3). In Clean::IPM, there are basically two kinds of data objects defined by the respective servers, i.e. the KKT objects and the IPM objects. Examples for KKT objects are the KKT matrix, a factorization of the KKT matrix called KKT inverse and the KKT vector, which implements the variable vector and the right-hand side of the KKT system (4.67). While a KKT matrix or its inverse are optional types the programmer can decide on whether and how to implement them, the KKT vector is needed by the IPM server and its design must meet two requirements. First, the KKT vector comprises three subvectors, i.e. a vector of the primal variables (`Var_Vector`), a vector of Lagrange multipliers corresponding to the equality constraints (`Equ_Vector`) and a vector of duals corresponding to the range constraints (`Rng_Vector`). Second, the KKT vector and all its subvectors are Clean vectors as described

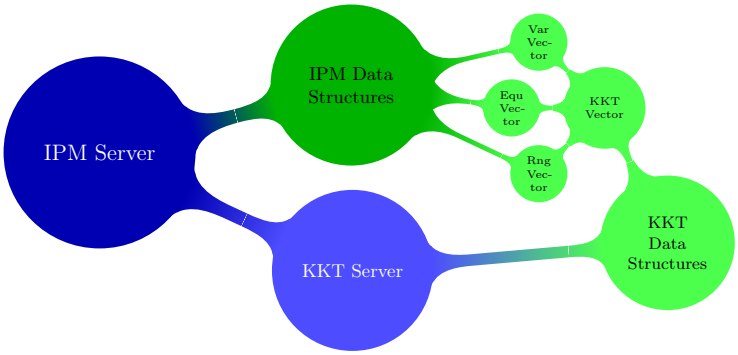


Figure 6.3.: Composition of IPM objects in Clean::IPM

in Sect. 6.1.3. The KKT server determines the vector data management of the KKT vector and its subvectors. This means, the KKT server is responsible for the way the vector data are stored in the memory block, i.e. the order of the data elements in the data array.

IPM objects are those objects that are introduced specifically for the implementation of the IPM. They are either vectors or diagonal matrices. Each object defined by the IPM server is composed of one or several KKT subvectors. In doing so, the IPM server adopts the vector data management of the KKT server for its objects and, hence, there is no need for a data mapping between the two servers. This way, IPM objects harmonize generically with KKT objects. Thus, all objects of the IPM server harmonize with each other as required. The design of the IPM objects is shown in Fig. 6.3.

Using Tree-Sparse Objects in Clean::IPM

Taking all previous described aspects into account (all interfaces, the responsibilities xof the KKT server as well as the coupling between this server and the problem), problem-tailored data objects are used in Clean::IPM the following way:

1. Implementing a KKT server that provides the KKT solver and the KKT matrix interfaces and which defines the KKT vector and its subvectors.
2. Providing the vector interface for each vector type.
3. Implementing a problem subalgorithm fitting to the KKT server that provides the problem interface.

6.2. Distributed Objects and Algorithms

The distribution of algorithms in Clean is motivated by the topic of this thesis. The design of distribution and its application in Clean::IPM is a joint work of Marc Steinbach, Martin Schmidt

as developer of Clean::IPM [72] and the author as developer of problem-tailored data objects for distributed problems (cf. Sect. 6.4). In Clean, the distribution of an algorithm is based on the distribution of the used data objects. More precisely, Clean algorithms are completely independent of the used computational environment that is kept hidden within the objects. This way, Clean algorithms are distributed algorithms by design.

The key ingredient for distributing an algorithm is the fundamental concept of the communicator, which is outlined in Sect. 6.2.1. The communicator hides the computational environment in use and provides a set of prescribed communication routines. This concept is motivated by the MPI standard [96]. A second important ingredient is a specific design of the data objects that are used by an algorithm. This design benefits from a distributed vector (Sect. 6.2.2), which extends a Clean vector (cf. Sect. 6.1.3) by including its distribution in its defining structure. The objects in Clean::IPM feature this specific design that is explained in Sect. 6.2.3.

6.2.1. Communicators

The entire structure of the network topology of the used computational environment—no matter if it is a parallel system (e.g. a shared-memory system or a distributed-memory system) or a sequential system—is encapsulated within a communicator. The communicator masks the specific environment in use and provides a fixed interface that consists of three types of routines, namely getter methods for some structure information, point-to-point (P2P) communication routines and collective communication routines. P2P methods allow direct data transmissions between two computational units (e.g. processes or threads) and collective communications involve all computational units in the computational environment. The routines of the communicator interface are listed in Table 6.1.

6.2.2. Distributed Vectors

A distributed vector is a vector split into distributed vector parts, which are distributed among the working units of the computational environment. This includes a single-split vector on a sequential system. The design of a distributed vector extends the design of an undistributed vector (cf. Sect. 6.1.3) as follows. The distribution of a vector becomes an aspect of its defining structure. The structural information of a distributed vector contains its total length, the length of the local vector part and the communicator. Each part of the distributed vector is stored in a single data array, i.e. a contiguous memory block, and the contents of the vector parts form a partition of the content of the distributed vector. Finally, the vector interface remains the same.

Table 6.1.: Communicator interface in Clean

Name	Type	Functionality
`process_id`	structure	Returns ID (rank) of the calling process.
`n_processes`	structure	Returns number of processes in communicator.
`send`	P2P	Sends some data to specified process. Blocks process until endpoint received data.
`send_asynch`	P2P	Sends some data to specified process. Process is not blocked.
`recv`	P2P	Receive some data from specified process. Blocks process until data is received.
`broadcast`	collective	Broadcasts some data.
`all_reduce`	collective	Reduces some data by a specified operation and scatters the result to all processes.
`synchronize`	collective	Synchronizes all processes.
`all_add_and_scatter`	collective	`all_reduce` with addition
`all_mult_and_scatter`	collective	`all_reduce` with multiplication
`all_min_and_scatter`	collective	`all_reduce` with `min`
`all_max_and_scatter`	collective	`all_reduce` with `max`
`all_and_and_scatter`	collective	`all_reduce` with logical `and`
`all_or_and_scatter`	collective	`all_reduce` with logical `or`

A distributed vector is responsible for performing a vector operation appropriately in the computational environment. The maximum norm of a distributed vector, for example, may be performed the following way. First, on each computational unit, the maximum norm of the assigned vector part is evaluated locally. Afterwards, the maximum over all vector parts is evaluated using the `all_max_and_scatter` routine of the communicator.

6.2.3. Distributed Data Objects in Clean::IPM

Algorithms in Clean are by now completely distributed:

1. An algorithm is independent of the used data objects by parameterization.
2. An algorithm delegates its distribution to these objects.
3. Assembling an algorithm with distributed data objects leads to a distributed setup of the algorithm.

An algorithm not only features distribution by design, the distribution can additionally be abetted by the used data objects. Employing data objects that are in a way generic with respect to the distribution allows their reuse in combination with problem-tailored objects. The way this is done in Clean::IPM is outlined next.

The second item in the listing above essentially means that an algorithm delegates its distribution to its server that in turn defines data objects harmonizing with each other (cf. Sect. 6.1.3).

The distribution becomes a key aspect of harmonization, which the programmer of the server has to keep in mind. In Clean::IPM, the server of the algorithm, i.e. the IPM server, delegates this responsibility further to the KKT server. The IPM objects are composed of KKT subvectors (cf. Sect. 6.1.4), which are possibly distributed vectors (cf. Sect. 6.2.2). Hence, the IPM server adopts the distribution of the KKT server for the IPM objects. In doing so, the distributed IPM objects harmonize generically with the KKT objects and with each other.

6.3. The Distributed Tree Environment

The *Distributed Tree Environment* (Clean::DTE) is a C++ library that provides distributed trees and an infrastructure for distributed tree algorithms in the TSL (see Chap. 5). Clean::DTE is developed to support the implementation of tree-sparse problems and their problem-tailored data objects (cf. Sect. 6.4). The design of Clean::DTE is influenced by the *Boost Graph Library* (BGL) [75] as well as by previous software that has been developed under supervision of Steinbach in the context of tree-sparse problems. The name of Clean::DTE is inspired by Hofmann's *Parallel Tree Environment* (PTE) [46].

Instead of writing a new library, two existing ones could have been employed to realize the features of Clean::DTE. With the BGL, Boost provides a generic library for describing and iterating over graphs, and there is even a distributed version called the *Parallel Boost Graph Library* (PBGL) [98]. But trees are very specific graphs and the BGL has been considered as too general by the author to fit the desired functionalities into its framework. The PTE library, on the other hand, is a generic library for distributed trees. Unfortunately, the PTE dictates a specific design for distributing objects on trees to fit into its framework. The prescribed object design has been considered as unfitting and too restrictive for the objects in the TSL.

Clean::DTE is not only developed to provide distributed trees but also to support the distribution of trees, i.e. the way from the tree to the distributed tree. Two considerations for the distribution of trees affect the design of the library. First, the DFS-based distribution rule (cf. Sect. 5.2.1) is one way to split a tree. Although it is currently the only distribution rule, other rules are not excluded by design. Second, the distribution of a tree among the working units may involve transmission of some data between these units. Clean::DTE supports data transmission by the serialization of data types using the Boost.Serialization library [94].

6.3.1. Main Design

The Clean::DTE library consists of *containers* that contain *elements* and provide specific functionalities. Distributed tree algorithms are realized by the interaction of *traversals* and

visitors. The concept of containers and elements in Clean::DTE is geared to the concept of generic containers[2] in the STL [90, 99]. A generic container in the STL is a template container class that can be instantiated to contain any type of object. Two restrictions to this arbitrary choice of objects make the containers in Clean::DTE less generic. First, the objects stored in the container must meet some requirements on its design that are described in Sect. 6.3.3. Objects that meet these requirements are called elements. Second, a container may use its elements to define its structure, e.g. to describe the topology of a tree.

There are currently three types of containers in Clean::DTE, i.e. two graph data structures implementing a *tree* and a *distributed tree* as well as an object type called *splitter* that supports the distribution of a tree. Details about these three containers are given in Sect. 6.3.2.

The interoperability between graph data structures and graph algorithms is similar to the one in the BGL [75]. A traversal defines the algorithm pattern outside of the containers and uses an interface masking the details of the graph implementation. A graph algorithm is extended by visitors providing certain *event points* that are invoked during the traversal [28]. The traversals and visitor types in Clean::DTE are presented in Sect. 6.3.4.

There are two major differences between the graph data structures and graph algorithms in the BGL and the ones in Clean::DTE. First, the graphs in Clean::DTE are distributed trees and undistributed trees as special cases of distributed trees. The traversals and their interfaces are tailored to these specific graphs and do not support arbitrary graphs like the BGL. Second, the graph data structures in Clean::DTE, i.e. the tree and distributed tree containers, comprise only nodes and no graph edges. Graph algorithms are always traversals of all nodes, and the event points of visitors are only invoked at nodes and not on edges.

6.3.2. Containers

The containers in Clean::DTE and their relations to each other are shown in Fig. 6.4. A *distributed tree* container type implements the topology of a distributed tree (cf. Sect. 5.2.1). It contains elements representing the nodes, levels, roots and senders of a distributed tree part and it provides an interface for the traversals. A *tree* container type implements the topology of an undistributed tree. It contains elements representing the nodes and levels of the tree and provides an interface for a splitter as well as the same traversal interfaces as the distributed tree container type. A *splitter* container type implements a distribution rule. Given an undistributed tree, the number of participating working units (n_p) as well as relevant system information ($w_1, \ldots, w_{n_p}$) as input, the splitter produces construction arguments for

[2]The generic containers of the STL are included in the C++ language and are part of the C++ standard library.

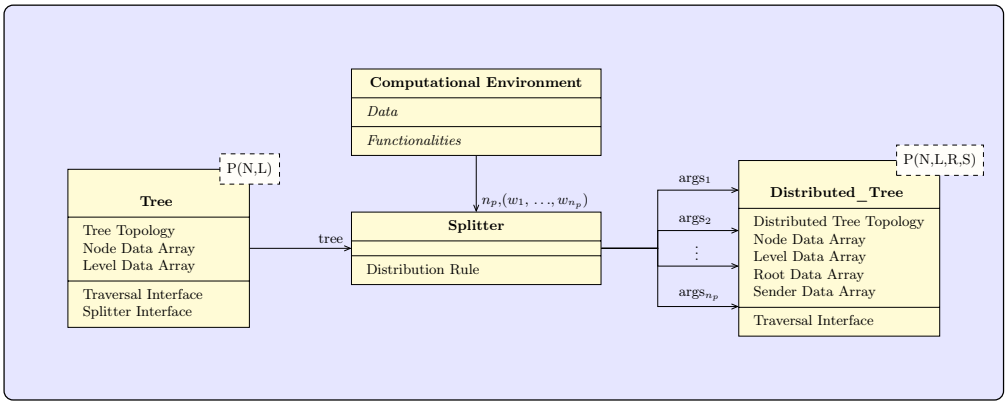


Figure 6.4.: Container types in Clean::DTE with template policies (`P(...)`) containing types for nodes (`N`), levels (`L`), roots (`R`) and senders (`S`)

the resulting distributed tree parts including the respective shapes of the parts as well as their incorporation in the distributed tree

Currently, Clean::DTE provides the following implementations of container types. The `Tree` container implements an arbitrary tree, although the considered trees are specific ones: they are symmetric and all leaves are on the same level. The `DFS_Splitter` defines the DFS-based distribution rule and produces construction arguments for the `DFS_Distributed_Tree`, which implements the distributed tree type arising from that specific distribution rule.

6.3.3. Elements

Elements are considered as basic types that must meet three requirements. First, any element must provide an empty constructor, i.e. the element can be built without passing any arguments upon its construction. Second, any element must be transmittable in the used computational environment, i.e. the element must be serializable to a data stream that can be transferred using the communication routines of a communicator (cf. Sect. 6.2.1). The third requirement allows the assembling of new elements from existing ones: except for the `Empty_Element`, any element `X` must be combinable with any other element `Y` but itself. The concentration of two elements is realized by inheritance from a template parameter. The `Empty_Element` is the endpoint in the inheritance hierarchy, i.e. it is the supreme base class. In Fig. 6.5, for example, the element `X` is concentrated with the element `Y` by passing `Y` as template parameter to `X` that in turn derives itself from the template `Y`. The element `Y` is then derived from the `Empty_Element`.

The elements provided and used by Clean::DTE are shown in Fig. 6.6. As basic types, they are not realized by the policy-traits design. There are nodes, levels, roots and senders for the (distributed) trees as well as the `Empty_Element` and the `Data_Element` that allows to attach

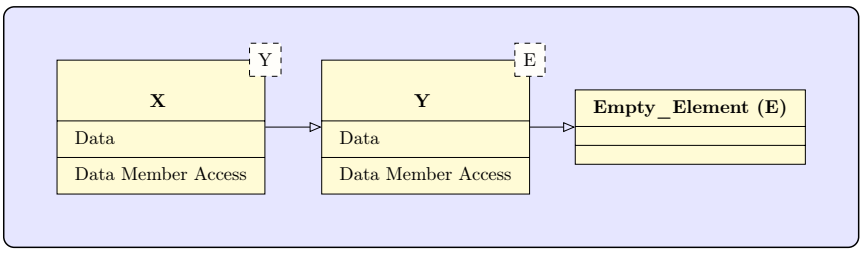


Figure 6.5.: Concentration of two elements in Clean::DTE

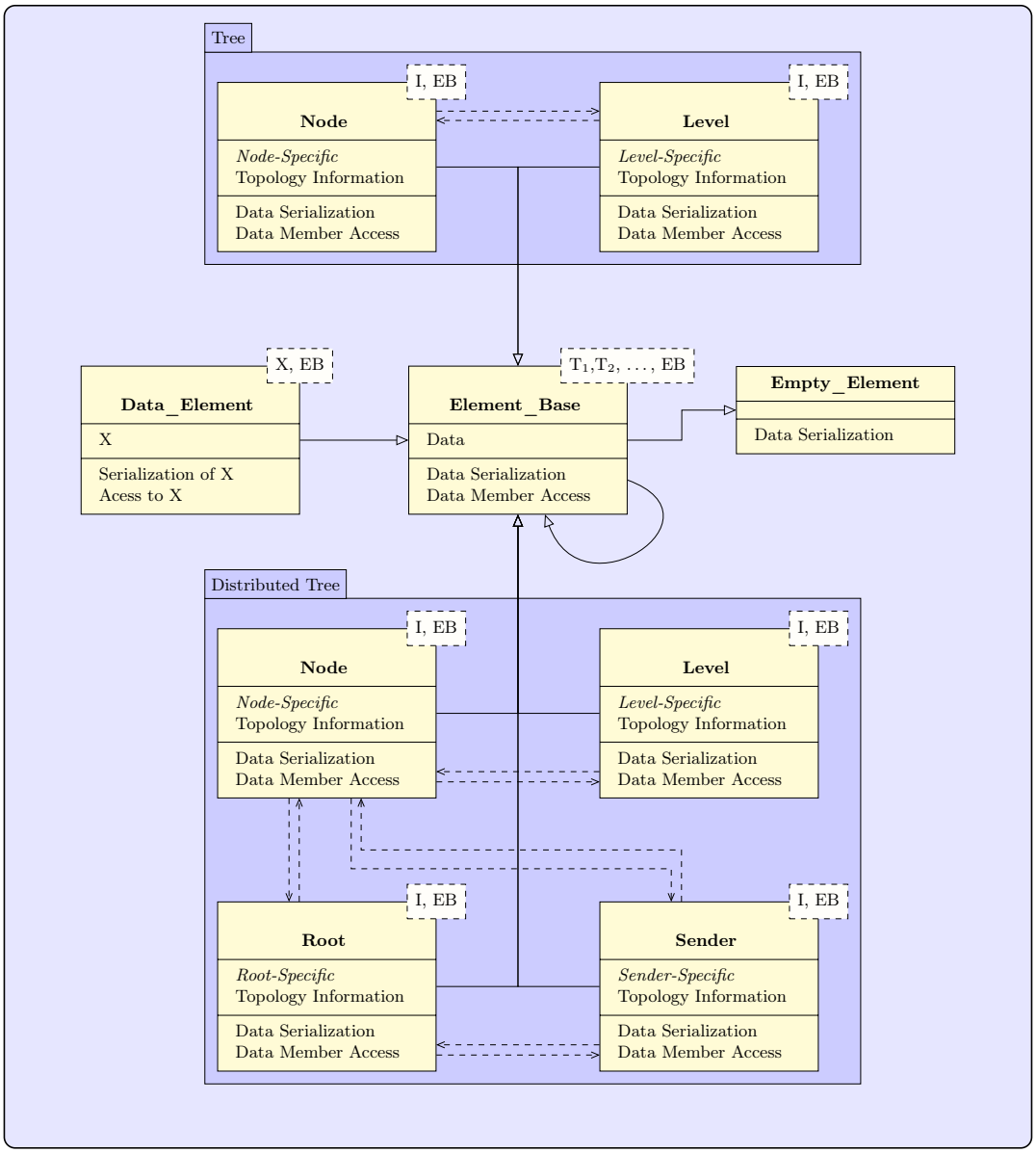


Figure 6.6.: Elements in Clean::DTE with template parameters `Integral Type` (`I`), `Element_Base` (`EB`) and an arbitrary serializable structure `X`

Table 6.2.: Traversals in Clean::DTE

Name	Traversal
`linear_forward`	Linear iteration over node array from first to last element.
`linear_backward`	Linear iteration over node array from last to first element.
`depth_first`	Traversal of all nodes in depth-first order.
`depth_first_reverse`	Traversal of all nodes in reverse depth-first order.

Table 6.3.: Event points in Clean::DTE

Name	Invocation
Event Points for Linear Traversals	
`start`	At the begin of the iteration before the first node is visited.
`start_node`	At the visit of each node in the node array.
`end`	At the end of the iteration after the last node has been visited.
Event Points for Depth-First Traversals	
`start_forest`	At the begin of the traversal before the first node is discovered.
`start_tree`	At the discovery of each root in the distributed tree part.
`start_node`	At the discovery of each node in the distributed tree part.
`finish_node`	At finishing of each node in the distributed tree part.
`finish_tree`	At finishing of each root in the distributed tree part.
`finish_forest`	At the end of the traversal after the last node has been finished.

arbitrary data to an assembled element. In Fig. 6.6, the `Element_Base` represents any element but the `Empty_Element` and the inheritance loop arrow illustrates the assembling of arbitrary many elements stopping with the `Empty_Element`. Elements subsumed in boxes in Fig. 6.6 are used by the `Tree` and `DFS_Distributed_Tree` containers to define their respective graph structures.

6.3.4. Traversals and Visitors

Currently, there are two types of traversals listed in Table 6.2. First, *linear traversals* iterate over the nodes of a distributed tree part from begin to end of the node array or vice versa. Second, *depth-first traversal* traverse the nodes of a distributed tree part in depth-first or reversed depth-first order (cf. Sect. 5.3.2). Each traversal type defines its own event points. Table 6.3 lists the event points of linear and depth-first traversals and, furthermore, specifies the events of their respective invocations.

6.4. The Tree-Sparse Library

The Tree-Sparse Library subsumes the implementation of the TSPs and the tree-sparse algorithms (see Chap. 4). This library is developed by the author to solve the tree-sparse problems using the interior-point framework Clean::IPM, and it requires Clean as well as the Clean::DTE library (see Sect. 6.3). The design of the TSL is greatly influenced by the designs of Clean (cf. Sect. 6.1.3) and Clean::IPM. Furthermore, some aspects of its design are inspired by Hutanu [49], by previous work of the author [50] as well as by further software development in the context of tree-sparse optimization under supervision of Steinbach.

6.4.1. Main Design

The TSL contains *tree-sparse objects* such as problem-tailored KKT objects (cf. Sect. 6.1.4) and implementations of the tree-sparse optimization problems. Furthermore, the library facilitates the implementation of tree-sparse algorithms such as evaluating the problem data of a TSP and performing the three stages of the KKT solution. In this context, these algorithms are referred to as *tree-sparse operations*. The TSL is developed simultaneously with the Clean::DTE library and the latter is designed to support the first one. Hence, the design of the TSL is based on Clean::DTE and strongly coupled to its design. The tree-sparse objects use distributed tree containers (cf. Sect. 6.3.2) to manage their data and to execute the tree-sparse operations. Each tree-sparse operation involves a traversal over the distributed tree and the performance of the operation is managed by an *algorithm visitor* (cf. Sect. 6.3.4). Managing an operation includes invoking its corresponding *node operations* (cf. Sect. 5.1).

The guiding idea for the design of the TSL is an easy support for the exploitation of local sparsities in the hierarchical sparsity of tree-sparse problems (cf. Sect. 3.2.3). This is realized by a specific modular design of the tree-sparse objects that reflects the tree-sparse sparsity hierarchy as well as by a close cooperation between those objects and the algorithm visitors. The latter feature a design that is independent of the local sparsity layer in the hierarchy. This way, the management of a tree-sparse operation is decoupled from the implementation of its problem-tailored node operations. Details on the algorithm visitors are given in Sect. 6.4.2.

Tree-sparse objects are composed of several classes, each of which is associated with a specific layer in the sparsity hierarchy. The classes corresponding to the local sparsity layer provide the implementation of the node operations. Assembling a tree-sparse object requires the selection of a representative for each layer. This way, the implementation of problem-tailored objects is narrowed down to implementing problem-tailored representatives for the local sparsity layer. The design of the tree-sparse objects is discussed in detail in Sect. 6.4.3. That section

Table 6.4.: Algorithm visitors in the TSL

Algorithm Visitor	Tree-Sparse Object	Contr.-Spec.	Type
`NLP_Eval`	Problem	yes	inward
`Vector_Norm_1`	KKT Vector and Sub-Vectors	no	inward
`Vector_Scalar_Product`	KKT Vector and Sub-Vectors	no	inward
`Vector_Sum`	KKT Vector and Sub-Vectors	no	inward
`Linear_Map`	KKT Matrix	yes	inward
`Hessian_Quadratic_Form`	KKT Matrix	yes	inward
`Factorize`	KKT Inverse	yes	inward
`Substitute_Inward`	KKT Inverse	yes	inward
`Substitute_Outward`	KKT Inverse	yes	outward

also describes the close cooperation of the objects and the algorithm visitors in performing a tree-sparse operation.

The TSL provides default implementations for the node operations of KKT matrices and their inverses. These default operations are based on BLAS and do not exploit local sparsities. Section 6.4.4 gives some details on this feature and, additionally, describes an infrastructure that allows to solve tree-sparse KKT systems using sparse routines of state-of-the-art sparse libraries such as the HSL Mathematical Software Library [47].

6.4.2. Algorithm Visitors for Tree-Sparse Operations

Tree-sparse operations are realized by distributed DFS-based tree algorithms (cf. Sect. 5.3.2). An operation is carried out by a depth-first traversal over the distributed tree as well as an operation-specific algorithm visitor that manages the operations using the event points that are invoked by the traversal (cf. Sect. 6.3.4). Usually, there are differences in the management between the incoming and the outgoing control version of the operation, leading to control-specific implementations of the algorithm visitors. Table 6.4 lists the algorithm visitors in the TSL and specifies the objects that define the corresponding node operations. For stability reasons, most tree-sparse operations are performed as inward algorithms (cf. Sect. 4.6.2).

Managing a tree-sparse operation includes three responsibilities that are in the following illustrated by means of the factorization of the tree-sparse KKT matrix. For this, Lst. 6.3 presents a simplified version of the algorithm visitor that manages the incoming control version of the distributed factorization (see Sect. 5.3.4). The first job of an algorithm visitor is to invoke the algorithm-specific operations $\mathrm{OP}(j)$ on the operation-defining object. Furthermore, the visitor has the responsibility to provide access to data that are required for the node operation but are inaccessible within the scope of the operation-defining object, i.e. data of other tree-sparse objects and data of the same object that are in other sets $\mathrm{DA}(j_2)$ of other nodes

Listing 6.3: Simplified factorize algorithm

```
1 template<...>
2 class Factorize_Algorithm
3   : public Inward_Communication_Skeleton<...>
4 {
5 public:
6   using IW_Base = Inward_Communication_Skeleton<...>;
7
8   void
9   start_forest()
10  {
11    // communication
12    IW_Base::clear_buffer();
13  }
14
15  void
16  finish_node()
17  {
18    // communication
19    IW_Base::receive_and_add_pred_write_data_from_outer_nodes();
20    M_add_received_data();
21
22    // node operation
23    M_inverse.factorize_node(Hi(), Fi(), ...);
24  }
25
26  void
27  finish_tree()
28  {
29    // communication
30    M_fill_send_data();
31    IW_Base::send_additive_pred_write_data_inward();
32  }
33
34  void
35  finish_forest()
36  {
37    // node operation
38    M_inverse.factorize_forest(...);
39  }
40
41 }; // class Factorize_Algorithm<...>
```

Table 6.5.: Skeletons in the TSL

Skeleton Visitor	Algorithm Type	Purpose
`Stable_Cumulation`	inward	Stable data accumulation.
`Collective_Sum`	inward	Stable accumulation of a scalar.
`Inward_Communication`	inward	Communication buffer management and communication routines.
`Outward_Communication`	outward	Communication buffer management and communication routines.

$j_2 \neq j$. In Lst. 6.3, for example, the tree-sparse factorization is formed by the node operations `factorize_node` (line 23) implementing items 1 to 11 in Table 5.3 as well as `factorize_forest` (line 41) realizing item 12 in the same table. Both node operations are defined by the KKT inverse object. The data `Hi` and `Fi` in the set DA(i) of the predecessor $i = \pi(j)$ are passed through this node operation to the KKT inverse. As the third of its responsibilities, an algorithm visitor manages the communication that is involved in the performance of the distributed tree-sparse operation. During the tree-sparse factorization in Table 5.3, for example, the factorization visitor initiates the receiving ($\curlyvee$) before the node operation `factorize_node` is invoked for a sender as well as the sending ($\downarrow$) after `factorize_node` is invoked for a root.

For each inward algorithm, the kinds of communication are the same and only the lengths of the transmitted data differ from each other. This common communication feature of inward algorithms is implemented by the `Inward_Communication_Skeleton`. Other so-called *skeleton visitors* implement, for example, the communication during an outward algorithm or the stable data accumulation (cf. Sect. 4.6.2). Table 6.5 lists the skeleton visitors in the TSL and their respective purposes.

The factorization visitor in Lst. 6.3 employs the `Inward_Communication_Skeleton` to manage the communications in Table 5.3. It first initiates the receiving of data from successors on other distributed tree parts (line 19) and then accumulates the received data (line 20). Before sending data from a root to its predecessor on another distributed tree part (line 31), the algorithm visitor first fills the respective data into the communication buffer of the skeleton (line 30).

6.4.3. Tree-Sparse Objects

The tree-sparse objects (illustrated in Fig. 6.7) include

1. all vector types (i.e. the limits vector of bounds, the limits vector of ranges and the KKT vector types),

2. all KKT objects (i.e. the matrix, the inverse, the KKT vector and its subvectors),

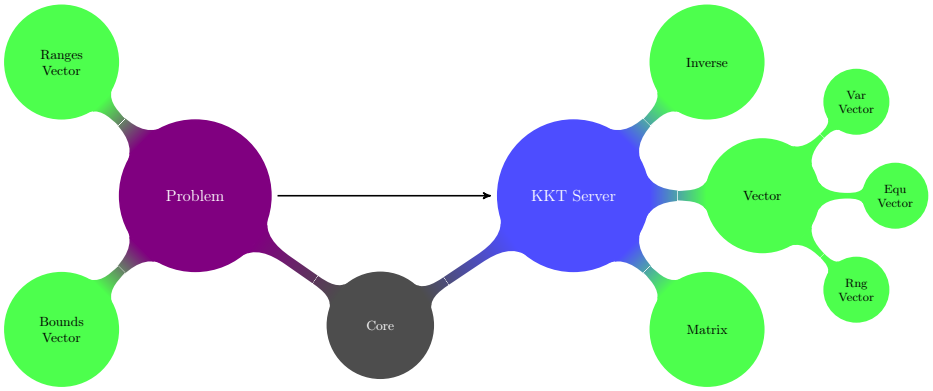


Figure 6.7.: Objects in the TSL

3. implementations of tree-sparse problems and

4. the KKT server.

All tree-sparse objects share common data members that are encapsulated into the tree-sparse core. These common data members include the problem dimensions, the distributed tree, a buffer for communication routines and a workspace for node operations. Coming along with the common data members are common features (i.e. getter methods, interfaces and other routines) that are gathered in an `Object_Base`-class. Each tree-sparse object inherits from the `Object_Base`. Additionally, there are object-specific base classes that summarize object-specific features for the KKT matrix, the KKT inverse and the optimization problem, respectively. Figure 6.8 depicts the inheritance hierarchy and the common and object-specific features for the tree-sparse objects `Matrix`, `Inverse` and `Problem`.

Figure 6.8 shows the coarse design of base classes in the TSL. Any tree-sparse object is actually derived from three `Object_Base`-classes, each of which represents a layer in the sparsity hierarchy of tree-sparse problems, i.e. the common tree-sparse layer, the control-specific layer and the problem-specific layer (cf. Sect. 6.4.1). The same design applies to the object-specific base classes and the assembling of a tree-sparse core. Figure 6.9 illustrates the detailed inheritance hierarchies for the `Matrix`, any tree-sparse `Vector`, the KKT `Server` and the tree-sparse `Core`.

The interfaces required by Clean::IPM are implemented in master classes (e.g. `Matrix`, `Inverse`, `Problem` or `Server`) that inherit from the several base classes (see Fig. 6.8 and Fig. 6.9). While being the most basic base classes in an standard object-oriented class design, these master classes form the endpoint in the inheritance hierarchy of the tree-sparse objects, i.e. they are the most specialized classes.

Performing a tree-sparse operation is a close cooperation between a tree-sparse object and an algorithm visitor (cf. Sect. 6.4.1 and 6.4.2). Using tag dispatching, a master class delegates

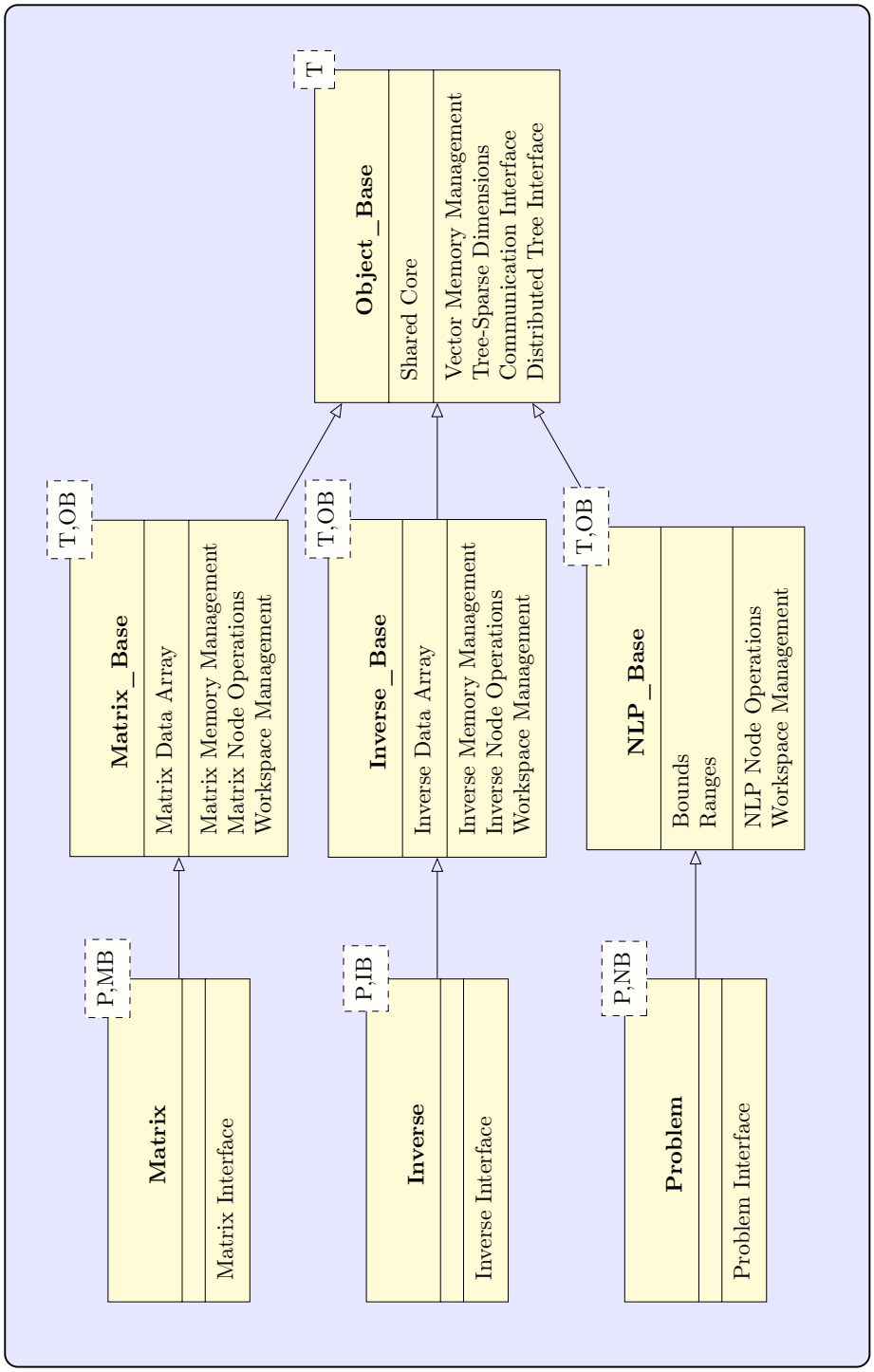


Figure 6.8.: Inheritance and template hierarchy for the tree-sparse objects `Matrix`, `Inverse` and `Problem` with template parameters `Policy` (`P`), `Traits` (`T`), `Matrix_Base` (`MB`), `Inverse_Base` (`IB`), `NLP_Base` (`NB`) and `Object_Base` (`OB`)

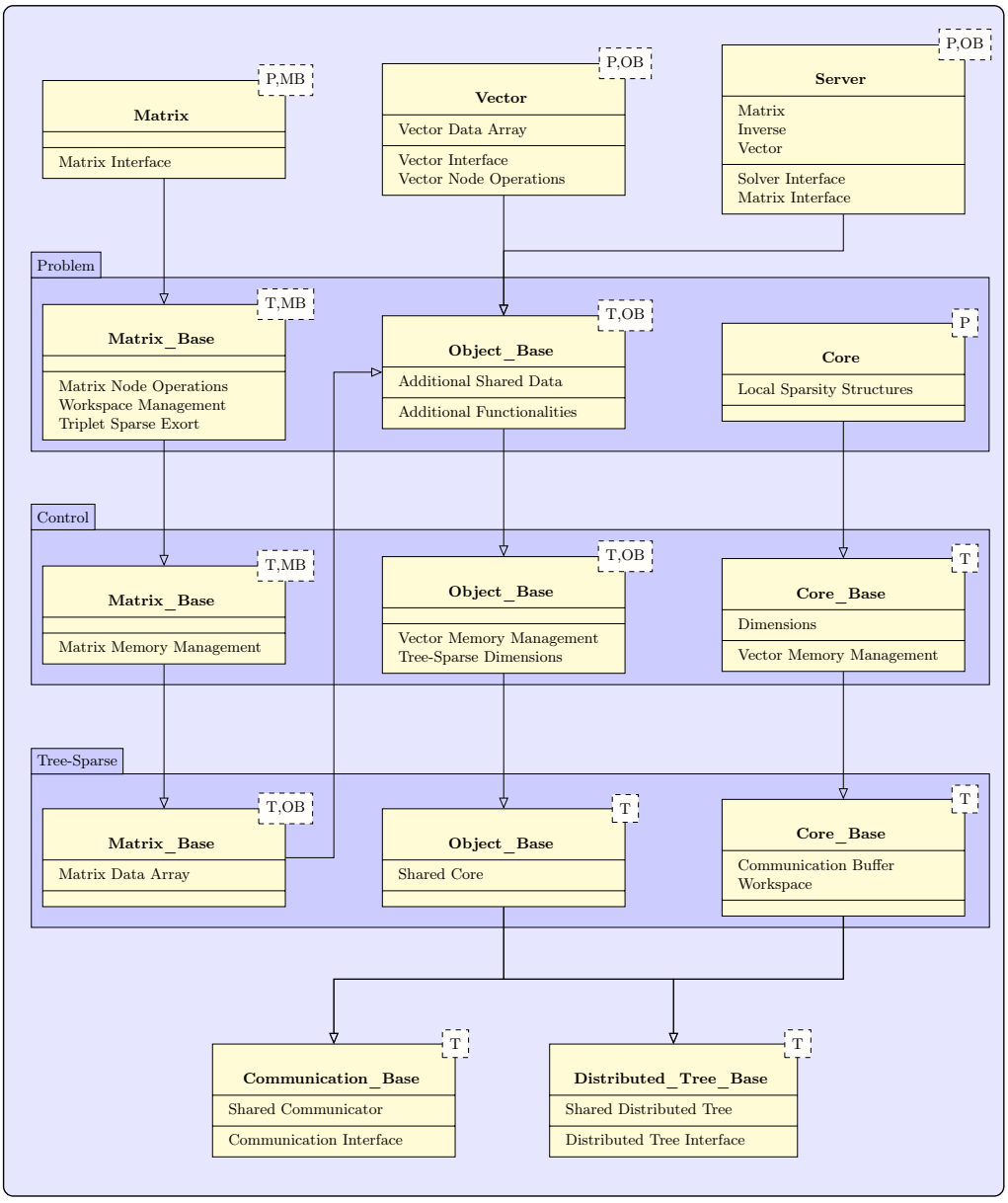


Figure 6.9.: Detailed inheritance and template hierarchy for the tree-sparse `Matrix`, a tree-sparse `Vector` and the tree-sparse `Core` with template parameters `Policy` (`P`), `Traits` (`T`), `Matrix_Base` (`MB`) and `Object_Base` (`OB`)

Table 6.6.: Elements in the TSL extending Clean::DTE

Name	Purpose
`Probability_Node`	Attaches a probability to an element.
`Offset_Node`	Memory management of data of tree-sparse objects.
`TS_Offset_Node`	Triplet-sparse export for the tree-sparse KKT matrix.

the management of a tree-sparse operation to the respective control-specific algorithm visitor. Furthermore, the algorithm visitor gets granted access to the otherwise private node operations of the tree-sparse object.[3] The control flow for the factorization of the tree-sparse KKT matrix, for example, is as follows:

1. A client calls the routine `factorize` on the master class `Inverse`.
2. The `Inverse` uses tag dispatching to delegate the task to its control-specific overload of the `factorize` routine.
3. The `Inverse` creates an instance of the control-specific `Factorize_Algorithm` visitor and passes itself to this instance.
4. The `Inverse` initiates a `depth_first` traversal with the `Factorize_Algorithm` instance.
5. The visitor manages the factorization of the KKT matrix and invokes the node operations `factorize_node` and `factorize_forest` defined by the problem-specific `Inverse_Base`.

6.4.4. Features of the Tree-Sparse Library

First, the TSL makes use of the element design in Clean::DTE (cf. Sect. 6.3.3) and provides further element types that are useful in the context of tree-sparse optimization. The new introduced elements are listed in Table 6.6. Second, the TSL provides for both control cases standard implementations of the KKT matrix as well as its inverse that do not exploit problem-specific local sparsities. The `Dense_Core` defines the problem structure without local sparsities and the base classes `Dense_Matrix_Base` as well as `Dense_Inverse_Base` provide the respective node operations. These `Dense_`-classes are representatives of the problem-specific layer of the inheritance hierarchy in Fig. 6.9. This way, implementing a new tree-sparse optimization problem without exploiting local sparsities is reduced to implementing a new problem-specific `Problem_Base` that provides the node operations for the `NLP_Eval_Algorithm` visitor.

[3] To be more precise in terms of C++: the node operations are `protected` members of the object-specific problem-specific base class (e.g. `Inverse_Base`) and the algorithm visitor (e.g. `Factorize_Algorithm`) is a `friend class` of the corresponding object class (e.g. `Inverse`).

Finally, the TSL comes with an infrastructure for exporting the tree-sparse KKT matrix into triplet sparse format. Clean::IPM provides a default KKT library in triplet sparse format including the triplet sparse KKT server [72]. The tree-sparse object `TS_Problem` evaluates a tree-sparse optimization problem and stores the evaluated data into the triplet sparse KKT server.

Chapter 7

Numerical Results

The main goal of this chapter is to demonstrate the performance of the problem-tailored algorithms for the tree-sparse problems (TSPs) dealing with nonconvexities and missing second-order evaluations in nonlinear optimization as well as facing the computational demands of huge-scale problems. All optimization problems are solved using the interior-point solver Clean::IPM [72] employing the tree-sparse KKT solver for the problem-specific linear algebra. The TSPs are implemented in the framework of the Tree-Sparse Library (TSL) (cf. Sect. 6.4).

First, Section 7.1 presents examples that are modeled by means of the nonlinear tree-sparse problems and this way demonstrates the potentials of the modeling framework of the tree-sparse formulations. The resulting TSPs are nonconvex and some do not provide explicit evaluations of second-order derivatives, i.e. the Hessian of the Lagrangian is not available. As a remedy, Hessian approximations are evaluated in a quasi-Newton approach based on tree-sparse Hessian updates (cf. Sect. 4.5). The KKT solver incorporates the proposed inertia correction heuristic (cf. Sect. 4.4) to deal with nonconvexities in the arising KKT systems. Both the Hessian updates and the inertia correction are successfully combined to solve the optimization problems that arise in the control of the bioreactor in Sect. 7.1.3. Moreover, the results in Sect. 7.1.2 demonstrate that the tree-sparse Hessian updates are a competitive alternative even if explicit evaluations of second-order derivatives are available.

Second, Section 7.2 presents computational results that prove the practicability of the concept of distribution for the tree-sparse problems (cf. Sect. 5) and show the efficiency of the implementation. For this, huge-scale portfolio selection problems are solved in parallel and the parallel performance is analyzed. The very good performance results of the tree-sparse algorithms conform with reports of similar approaches in the literature [10, 34]. The complete IPM algorithm also features good performance results that are shaped by the performance of the tree-sparse algorithms and benefit from the latter. Moreover, even the largest considered

problems are solved in a few iterations and less than three minutes showing the scalability of the distribution.

In Sect. 7.2.4, results are presented that demonstrate the efficiency of the post-distribution communication reduction (cf. Sect. 5.3.3) for the trees that correspond to the portfolio selection problems. Applying the reduction heuristic often leads to an actual number of communication calls that is close to its lower bound. Finally, the results in Sect. 7.2.5 confirm the potentials of exploiting local sparsities in tree-sparse problems and demonstrate the flexibility of the design of the TSL.

7.1. Examples for Nonlinear Tree-Sparse Problems

The examples in this section are dynamic processes that are manipulated by optimization-based controllers leading to nonlinear tree-sparse problems that are used to analyze the performance of the tree-sparse algorithms. Most of the computations except for those in Sect. 7.1.2 are done in sequential on a workstation with 16 GB of RAM and an Intel(R) Core i7-3770 comprising 4 cores running at 3.40 GHz. First, the motion of the magnetic levitation vehicle considered in Sect. 7.1 is described by a simple deterministic ordinary differential equation (ODE). This allows to compare two formulations of TSPs with one using analytical solutions of initial value problems and the other using numerical approximations. Second, the dynamics of the perturbed nonlinear double integrator examined in Sect. 7.1.2 is already given in time-discretized form, leading to TSPs that provide explicit evaluations of second-order derivatives. This allows comparing the performance of the structured quasi-Newton approach with the performance of the IPM using explicit second-order evaluations. Finally, the nonlinear bioreactor studied in Sect. 7.1.3 is modeled by ODEs with uncertain parameters that lead to challenging TSPs including numerical evaluations of initial value problems. These problems are solved by means of the structured quasi-Newton approach using SR1 updates and relying on inertia corrections.

Clean::IPM is configured as shown in Table 7.1. The algorithm terminates with an optimal solution if the KKT error (cf. Sect. 2.2.1) is reduced to the prescribed optimality tolerance, and it terminates with an almost optimal solution if the KKT error is reduced to the respective tolerance and there is no significant progress in the subsequent IPM iterations. The progress towards a solution is monitored and enforced by a filter line-search approach (cf. Sect. 2.2.2). The update rule for the barrier parameter can be varied between the following three variants that are implemented in Clean::IPM: the LOQO rule proposed by Vanderbei and Shanno [103] and originally implemented in [102], the Ipopt rule proposed by Wächter and Biegler [105] and originally implemented in [106], and the rule of Mehrotra's predictor-corrector (MPC) algo-

Table 7.1.: Default configuration of Clean::IPM

IPM Building Blocks			
NLP globalization strategy:		filter line-search	
BP globalization strategy:		filter line-search	
Barrier update strategy:		LOQO, Ipopt, MPC	
Convergence Criteria		**Algorithmic Extensions**	
Maximum iterations:	3000	Starting point strategy:	enabled
Optimality tolerance:	10^{-6}	Automatic scaling:	disabled
Almost optimal tolerance:	10^{-4}	Slack shifting:	enabled
Almost optimal iterations:	15	Emergency mode:	enabled

rithm [65, 66]. Moreover, the framework of Clean::IPM provides several algorithmic extensions such as the modification of a user-defined starting point, the shifting of jamming slack variables, and an automatic problem scaling. Only the automatic problem scaling is disabled for reasons discussed in Sect. 4.6.3. Details about the used extensions and heuristics including default values for the parameters can be taken from [72].

In the quasi-Newton framework, the tree-sparse Hessian update strategy is based either on the SR1 formula or on the PSB formula given in (2.32). The implementation features typical heuristics such as a skip of the update if denominators become too small as well as an automatic reset of the Hessian approximation [66]. The latter is enabled by default and the approximation is reset after 30 iterations. The heuristic for skipping an update is applied individually for each node, the default skip tolerance is set to 10^{-8}.

Numerical solutions of initial value problems are computed with the integrator Metanb [5] that also provides approximations of first-order derivatives via internal numerical differentiation [14].

7.1.1. High-Velocity Magnetic Levitation Vehicle

The first example models the one-dimensional frictionless motion of a high velocity magnetic levitation vehicle, which is in the literature better known as the example of the rocket car [61]. The motion of the vehicle follows Newton's third law of motion $\ddot{s}(t) = \dot{v}(t) = F/m$ where s denotes the position of the vehicle, v its velocity, F the driving power and m its mass. Starting at $s(0) = s_0$ and $v(0) = v_0$, the task is to stop the vehicle in minimal time T at position $s(T) = s_e = 0$ with zero velocity $v(T) = v_e = 0$. For this, it can be controlled by adjusting its acceleration $u = \ddot{s}$ within the bounds $u = F/m \in [-\hat{u}, +\hat{u}]$. Writing the initial value

problem (IVP) for the equation of motion as first-order system yields

$$\begin{pmatrix} \dot{s}(t) \\ \dot{v}(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ u(t) \end{pmatrix} \quad \text{with} \quad \begin{pmatrix} s(0) \\ v(0) \end{pmatrix} = \begin{pmatrix} s_0 \\ v_0 \end{pmatrix}. \tag{7.1}$$

Assuming a constant acceleration $\bar{u}$, the analytical solution of the IVP (7.1) is easy to compute and leads to a polynomial of the second degree describing the position s of the vehicle, i.e.

$$s(t; \bar{u}, s_0, v_0) = \frac{\bar{u}}{2} t^2 + v_0 t + s_0 \quad \text{and} \quad v(t; \bar{u}, v_0) = \bar{u} t + v_0. \tag{7.2}$$

The optimal control for the magnetic levitation vehicle with the objective of minimizing T is well-known and follows a so-called *bang-bang* control strategy for the acceleration u. In this strategy, the vehicle first accelerates with maximum power $|\hat{u}|$ in one direction until reaching the switching point $0 \leq \tilde{t} \leq T$. After that, the vehicle brakes with maximum power, i.e. it accelerates with maximum power $|\hat{u}|$ in the opposite direction. The analytical solution of the levitation vehicle example with optimal times T and switching points $\tilde{t}$ in dependence on the initial states (s_0, v_0) can be found, e.g., in [85].

Tree-Sparse Problem Formulation

In a multiple shooting approach, the time interval $[0, T]$ is split into $N-1$ subintervals $[t_j, t_{j+1}]$ with $t_j = \frac{j-1}{N-1}$ for $j > 0$ and $N > 1$. The state and control variables for $j \in \{1, \ldots, N\}$ are $x_j = (s(t_j), v(t_j), T)^T$ and $u_j = u(t_j)$, respectively. The root is a dangling node with $x_0 = (s_0, v_0, 0)$ and $u_0 = T$. This way, the tree-sparse formulation of the magnetic levitation vehicle problem becomes an outgoing control problem (4.3) with the objective

$$\phi_N(x_N, u_N) = x_{N,3} \quad \text{and} \quad \phi_j(x_j, u_j) = 0 \quad \text{for} \quad j < N, \tag{7.3}$$

and the dynamics

$$g_1(x_0, u_0) = \begin{pmatrix} x_{0,1} \\ x_{0,2} \\ x_{0,3} + u_0 \end{pmatrix} \quad \text{and} \quad g_j(x_i, u_i) = \begin{pmatrix} g_{j,1}(x_i, u_i) \\ g_{j,2}(x_i, u_i) \\ x_{i,3} \end{pmatrix} \quad \text{for} \quad j > 1, \tag{7.4}$$

where the dynamic node functions $g_{j,1}$ and $g_{j,2}$ represent the solutions of the IVP (7.2), i.e.

$$g_{j,1}(x_i, u_i) = s(t_j; u_i, x_{i,1}, x_{i,2}) \quad \text{and} \quad g_{j,2}(x_i, u_i) = v(t_j; u_i, x_{i,2}). \tag{7.5}$$

The physical bound $|u| \leq \hat{u}$ for the acceleration leads to the simple bounds

$$[b^u_{l0}, b^u_{u0}] = [0, \infty) \quad \text{and} \quad [b^u_{lj}, b^u_{uj}] = [-\hat{u}, \hat{u}] \quad \text{for} \quad j > 0. \tag{7.6}$$

The initial conditions $s(0) = s_0$ and $v(0) = v_0$ as well as the terminal conditions $s(T) = 0$ and $v(T) = 0$ are incorporated into the dynamics and the global constraints by setting

$$g_0 \equiv (s_0, v_0, 0)^T \quad \text{and} \quad f_N(x_N, u_N) = (x_{N,1}, x_{N,2})^T, \quad \text{respectively.} \tag{7.7}$$

Note that this example does not consider uncertainties, i.e. the resulting optimization problem is deterministic and the corresponding scenario tree is a chain.

Test Run

The magnetic levitation vehicle problem is solved for a fixed setup, i.e. the initial position is $s_0 = -4.0$, the initial velocity is $v_0 = 0.0$ and the time interval is split into 100 equidistant subintervals. Moreover, two variants of the tree-sparse problem (7.3)–(7.7) are considered. In the first variant $\mathrm{V_I}$, the analytical solution (7.2) of the IVP is used to compute the next states $x_{j,1}$ and $x_{j,2}$ from (7.5). In the second variant $\mathrm{V_{II}}$ these states are obtained from solving the IVP (7.1) numerically using the integrator Metanb.

Both variants $\mathrm{V_I}$ and $\mathrm{V_{II}}$ are solved using the IPM with different configurations of the algorithm, and the respective solution times and numbers of iterations are compared to each other. Each IPM configuration comprises the selection of the barrier update rule (Ipopt or LOQO), the way the Hessians of the Lagrangian are computed and, finally, the choice of the KKT solver (tree-sparse or triplet sparse). The Hessians are either computed using explicit expressions for second-order derivatives or they are approximated in a quasi-Newton framework using tree-sparse Hessian updates (SR1 or PSB). The arising KKT systems are solved in three different ways, i.e. by the tree-sparse KKT solver using only local convexifications, or by the tree-sparse KKT solver using only the outer convexification, or by a triplet sparse KKT solver based on the HSL library [47]. The triplet sparse KKT solver uses the same convexification strategy as the outer convexification of the tree-sparse solver [72].

The performance results for all IPM configurations and both tree-sparse problem variants $\mathrm{V_I}$ and $\mathrm{V_{II}}$ are presented in Table 7.2. For each case, the table lists the number of required IPM iterations, the KKT solution time accumulated over all iterations as well as the complete solution time of the IPM solver. In all cases, the optimal solution found by the IPM solver coincides with the analytical solution of the magnetic levitation vehicle problem [85]. The upper half in Table 7.2 shows the results for the Ipopt barrier update rule and the lower half those for

Table 7.2.: Performance results for the magnetic levitation vehicle problem – Number of iterations, KKT solution time (s) and IPM solution time (s) for all algorithm configurations and both tree-sparse problem variants V_I and V_{II}

	Exact Hessian			SR1 updates			PSB updates		
KKT Solver	Iter.	KKT	IPM	Iter.	KKT	IPM	Iter.	KKT	IPM
Tree-sparse (local)	11	0.003	0.005	*no convergence*			*34	0.007	0.016
Tree-sparse (outer)	21	0.008	0.012	84	0.024	0.045	*34	0.007	0.016
Triplet sparse	*11	0.022	0.024	58	0.233	0.249	*34	0.065	0.074
V_I (analytical IVP solutions) ↑				V_{II} (numerical IVP solutions) ↓					
Tree-sparse (local)				*no convergence*			*30	0.007	0.034
Tree-sparse (outer)				82	0.024	0.096	*30	0.007	0.034
Triplet sparse				58	0.250	0.303	*30	0.064	0.098
Ipopt barrier update rule									

	Exact Hessian			SR1 updates			PSB updates		
KKT Solver	Iter.	KKT	IPM	Iter.	KKT	IPM	Iter.	KKT	IPM
Tree-sparse (local)	18	0.004	0.009	*no convergence*			32	0.007	0.015
Tree-sparse (outer)	16	0.006	0.009	71	0.022	0.038	48	0.015	0.028
Triplet sparse	*12	0.024	0.024	74	0.319	0.340	*43	0.160	0.171
V_I (analytical IVP solutions) ↑				V_{II} (numerical IVP solutions) ↓					
Tree-sparse (local)				*no convergence*			37	0.008	0.039
Tree-sparse (outer)				53	0.016	0.078	47	0.016	0.061
Triplet sparse				80	0.367	0.439	43	0.171	0.211
LOQO barrier update rule									

the LOQO update rule. Each of these halves first lists the results for problem variant V_I using the analytical IVP solutions in the dynamic node functions (7.5) and then those results for the second variant V_{II} using numerical integration. In some test cases, the optimal solution of the problem is found without modifying any KKT system, i.e. without using convexification during the run of the IPM algorithm. In Table 7.2, these cases feature iteration numbers labeled by a star (*).

First of all, in almost all test cases, except for those configurations that combine the SR1 Hessian updates with the tree-sparse KKT solver using only local convexifications, the optimal solution of the problem is found by the IPM solver showing that the different aspects of the solution approach are successfully combined. First, the incorporated inertia correction extends the tree-sparse KKT solver to deal with KKT systems arising in nonconvex optimization. Second, the structure-preserving tree-sparse Hessian update strategies in the quasi-Newton approach generate useful approximations of the Hessians of the Lagrangian.

The subsequent observations and conclusions are made from Table 7.2 by comparing the test cases with respect to one aspect of the algorithm.

Tree-Sparse (outer) vs. Tree-Sparse (local): Using only the outer convexification means that the KKT system is convexified uniformly, i.e. in the standard way by adding a multiple of the identity to the Hessian of the Lagrangian (cf. Sect. 2.2.2). This strategy leads for all configurations to the optimal solution. On the other hand, using only local convexifications, the IPM algorithm diverges for those configurations where the Hessians are approximated based on the SR1 update formula. Comparing the configurations where both convexification strategies succeed, local convexifications lead to smaller KKT solution times per iteration, e.g. 0.22 ms against 0.31 ms for V_I using the LOQO rule and PSB updates. Moreover, in all configurations but the one combining the LOQO rule with exact Hessians, the IPM requires fewer iterations and less computing time to find the solutions.

Tree-Sparse (outer) vs. Triplet Sparse: First, there are IPM configurations (e.g. for exact Hessians) for which the tree-sparse KKT solver requires convexification to solve the arising KKT systems whereas the triplet sparse KKT solver provides KKT solutions in all IPM iterations without modifying the KKT matrices. Hence, in some IPM iterations, the regularity assumptions ensuring the success of the tree-sparse KKT solution procedure (cf. As. 2) are not satisfied, which then affects the tree-sparse but not the triplet sparse KKT solver. Second, as expected, the problem-tailored tree-sparse KKT solver is significantly faster than the triplet sparse KKT solver, which more than compensates for a possibly higher number of required IPM iterations (see Ipopt test cases in Table 7.2).

Exact Hessians vs Hessian Updates: As can be expected, replacing exact Hessians of the Lagrangians with approximations based on tree-sparse Hessian updates results in a higher number of IPM iterations required to attain the solution, which consequently implies a larger IPM solution time.

SR1 vs. PSB: In all cases in Table 7.2, the tree-sparse Hessian updates based on the PSB formula perform better compared to the SR1 formula, i.e. using PSB updates leads to less IPM iterations in the solution procedure in comparison to using SR1 updates. Moreover, using the Ipopt barrier update rule, the PSB updates lead to approximations of the Hessian of the Lagrangian such that the corresponding KKT systems are solved without convexification, which results in a lower KKT solution time per iteration. For example, considering the solution times when employing the tree-sparse KKT solver using the outer convexification, the PSB updates lead to 0.21 ms per iteration whereas the SR1 updates lead to 0.28 ms.

Ipopt vs. LOQO: In most cases, using the Ipopt rule to update the barrier parameter (upper half in Table 7.2) leads to better performance results than using the LOQO barrier update rule (lower half in Table 7.2). However, the LOQO update rule performs better for Hessian approximations based on the SR1 formula when solved with the tree-sparse KKT solver using the outer convexification.

Analytical IVP Solutions vs. Numerical Integration: Replacing the analytical IVP solutions in the dynamic node functions (7.5) with approximations based on numerical integration affects the complete IPM solution time for the worse since evaluating the dynamics (7.4) and the respective first-order derivatives becomes more expensive. However, most test cases feature only minor discrepancies in the number of required IPM iterations, showing the IVP solutions and first-order derivatives obtained from the integrator are reliable.

7.1.2. Nonlinear Double Integrator

In the following example, a moving horizon controller (MHC) regulates a perturbed nonlinear double integrator to keep the considered system in a position of rest. For this, the plant representing the double integrator uses the already time-discretized dynamic model proposed by Lazar et. al [56],

$$x_1(k+1) = x_1(k) + x_2(k) + \frac{1}{40}\left(x_1^2(k) + x_2^2(k)\right) + \frac{1}{2}u(k) + d(k), \tag{7.8a}$$

$$x_2(k+1) = x_2(k) + \frac{1}{40}\left(x_1^2(k) + x_2^2(k)\right) + u(k). \tag{7.8b}$$

In this model, the state (x_1, x_2) of the system is manipulated by the control signal $u \in [-2, 2]$, and the first state variable x_1 is perturbed additively by the disturbance d with the nominal value $d^{\text{nom}} = 0$. The task is to bring the system into the position of rest $(x_1^*, x_2^*) = (0, 0)$ and keep it there. In the absence of the disturbance, the reference state (x_1^*, x_2^*) is a fixed point of the dynamics (7.8), i.e. the double integrator remains in the reference state without the need to regulate it.

The considered uncertainties and the applied cost function are chosen in the same way as by Lucia and Engell [59]. The disturbance is assumed to take the values $d \in \{-0.05, 0, 0.05\}$ with the respective probabilities $\{0.2, 0.4, 0.4\}$. Note that with $\bar{d} = 0.01$, the expected value $\bar{d}$ of the disturbance does not coincide with the nominal value d^{nom}. The costs are measured by a standard quadratic cost function that penalizes the deviation from the reference state as well

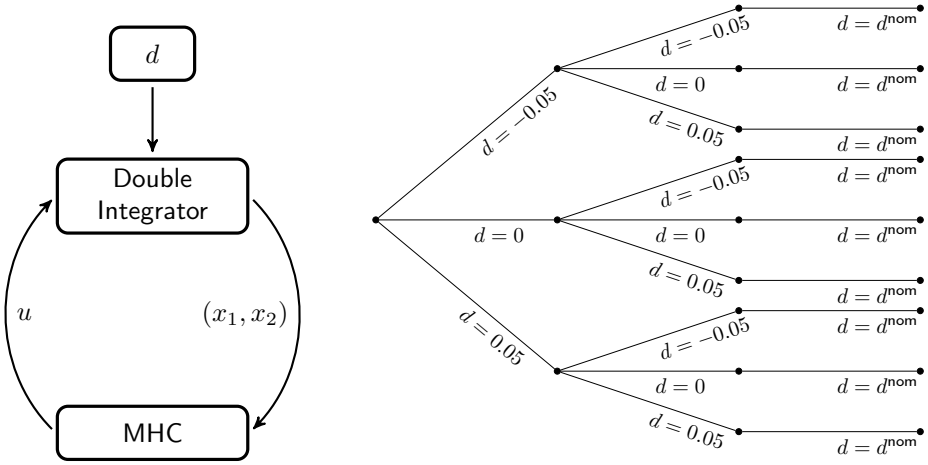


Figure 7.1.: Process control of the double integrator (left) and scenario tree considered in the optimization problem with prediction horizon $T = 3$ and number of branching levels $T_b = 2$ (right)

as the necessity to apply a control signal, i.e.

$$L(x, u) = (x - x^*)^T Q(x - x^*) + u^T R u \quad \text{with} \quad Q = I \quad \text{and} \quad R = 0.15. \tag{7.9}$$

In each sampling time, the double integrator receives the control signal u from the MHC, and the disturbance is determined in a random experiment varying between the specified values with the corresponding probabilities. The resulting new state (x_1, x_2) of the plant is then sent back to the controller (see Fig. 7.1).

Tree-Sparse Problem Formulation

The optimization problem that is solved for each sampling time to determine the new control signal u includes the costs (7.9) and the same dynamic model (7.8) as the plant. The problem is formulated as an outgoing control TSP (4.3) with the objective

$$\phi_j(x_j, u_j) = p_j \left(x_j^T Q x_j + u_j^T R u_j \right) \quad \text{for} \quad j \in V, \tag{7.10}$$

the dynamics

$$g_j(x_i, u_i) = \frac{1}{40} \begin{pmatrix} x_i^T x_i \\ x_i^T x_i \end{pmatrix} + \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x_i + \frac{1}{2} \begin{pmatrix} u_i \\ 2u_i \end{pmatrix} + \begin{pmatrix} d_j \\ 0 \end{pmatrix} \quad \text{for} \quad j \in V, \tag{7.11}$$

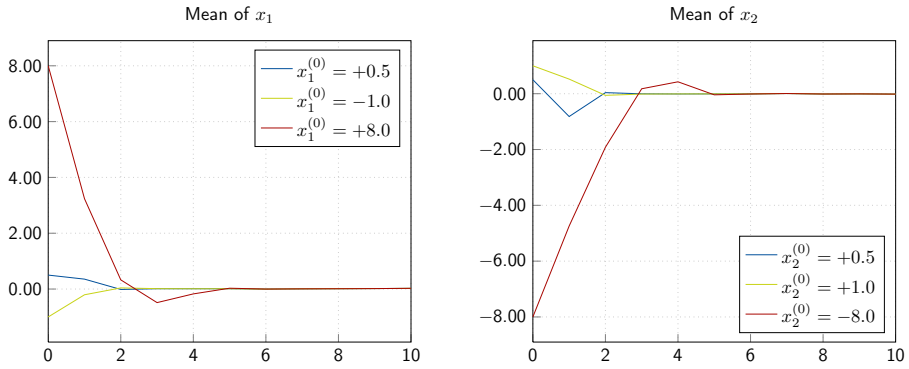


Figure 7.2.: Progress of the perturbed double integrator for different initial states

and the simple bounds

$$x_j \in (-\infty, \infty) \quad \text{as well as} \quad u_j \in [-2, 2] \quad \text{for} \quad j \in V. \tag{7.12}$$

The current state of the plant is incorporated as initial condition into the dynamics, i.e. by setting $g_0 \equiv (x_1^{(0)}, x_2^{(0)})$. The scenario tree of the tree-sparse problem has the form shown in Fig. 7.1 and is characterized by its depth (T) as well as the number of levels with tree branching (T_b). The tree depth corresponds to the prediction horizon, i.e. the number of time steps the problem forecasts into the future. For a node j with $t(j) \leq T_b$, it is $|S(j)| = 3$, hence all three possible outcomes for d are considered in the next time period. For a node j with $|S(j)| = 1$, the disturbance in the following time period is assumed to take the nominal value $d^{\text{nom}} = 0$.

Control into Position of Rest

In a first test, the performance of the controller is checked for bringing the perturbed double integrator from its initial state $(x_1^{(0)}, x_2^{(0)})$ close to the position of rest (x_1^*, x_2^*). For this, the MHC uses a deterministic optimization problem ($T_b = 0$) with prediction horizon $T = 3$ to regulate the plant. The optimization problems are solved using the Ipopt barrier update rule and explicit evaluations of second-order derivatives, i.e. exact Hessians of the Lagrangian. Each test of the series is run 50 times for different outcomes of the random experiment to determine the realization of the disturbance d at each sampling time. Figure 7.2 illustrates the means of the states in the progress of the double integrator for the first 10 time steps (x-axes). For each considered initial state, the double integrator is controlled within the first 5 time steps close to the position of rest (x_1^*, x_2^*) showing the proper operation of the controller.

Hold the Position of Rest with Minimal Costs

In the second test series, the performance of the controller is tested for keeping the perturbed double integrator in the position of rest. For this, the MHC uses tree-sparse optimization problems with prediction horizons $T \in \{3, 10\}$ and different numbers of branching levels T_b. Again, the problems are solved using the lpopt rule and exact Hessians of the Lagrangians, and each test is run 50 times for different outcomes of the random experiment at each sampling time.

Figure 7.3 illustrates the resulting average accumulate costs, the means of the states x_1 and x_2 as well as the average control signal u for 20 time steps (x-axes). As first striking observation, the graphs in Fig. 7.3 show that, in each considered case, the controller requires 5 time steps to adjust the control signal to the occurring disturbances. After that, the control signal is for all cases the same, resulting in the same development of the second state x_2 and a similar course of the first state x_1. Therefore, the first 5 time steps are crucial for the development of x_1. Secondly, the length of the prediction horizon has no significant influence on the performance of the controller. Using the prediction horizon $T = 3$ (left-hand side diagrams in Fig. 7.3) features the same progress of the double integrator as using the larger prediction horizon $T = 10$ (right-hand side diagrams in Fig. 7.3).

Now, the average accumulated costs (diagrams at the top of Fig. 7.3) demonstrate that the performance of the controller improves if including the uncertainties in the optimization problem. Considering uncertainties in the first time step of the controller, i.e. setting T_b from 0 to 1, the incurred costs at the final time step 20 are reduced by approximately 9 %. Increasing the number of branching levels to $T_b = 2$ leads to a further cost reduction of approximately 3 %. Then, larger numbers of branching levels ($T_b > 2$) have no significant affect on the costs. The saved costs are obtained by balancing the disturbances in the first 5 time steps of the process. The control signal is adjusted to reduce the deviation of x_1 from the reference state $x_1^* = 0$. This results in lower costs caused by deviations of x_1 and comes with higher costs caused by deviations of x_2 and by applying nontrivial control signals. Figuratively speaking, in the critical first 5 time steps, the incurred costs are moved from state x_1 to state x_2 and the control u. This strategy then pays off in the further development of the double integrator where the costs caused by u and x_2 are the same for all considered numbers of branching levels T_b while the costs caused by x_1 depends on its state at time step 5.

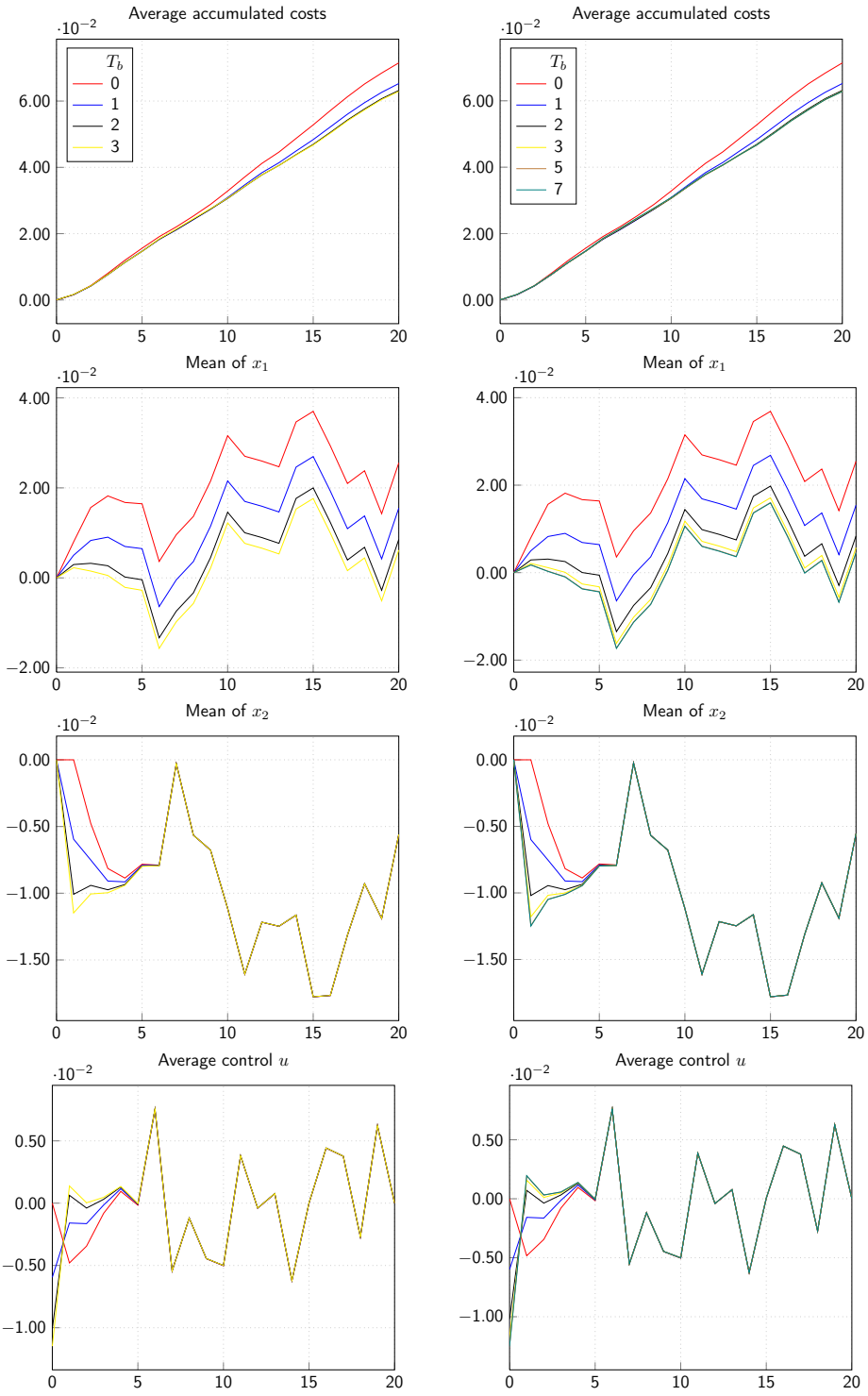


Figure 7.3.: Progress of the double integrator – Average accumulated costs, means of the states x_1, x_2 and average control u for prediction horizons $T = 3$ (left) and $T = 10$ (right) and an increasing number of branching levels T_b

Table 7.3.: Double integrator – Problem sizes and IPM solution times (s) using the Ipopt barrier update rule and exact Hessians of the Lagrangians

T_b	Nodes	Scenarios	Variables	Equalities	Bounds	Solution time
1	37	3	111	74	222	0.00
2	103	9	309	206	618	0.01
3	283	27	849	566	1 698	0.01
4	769	81	2 307	1 538	4 614	0.04
5	2 065	243	6 195	4 130	12 390	0.09
6	5 467	729	16 401	10 934	32 802	0.24
7	14 215	2 187	42 645	28 430	85 290	0.62
8	36 085	6 561	108 255	72 170	216 510	1.63
9	88 573	19 683	265 719	177 146	531 438	4.14
10	206 671	59 049	620 013	413 342	1 240 026	9.56
11	442 867	177 147	1 328 601	885 734	2 657 202	22.07
12	797 161	531 441	2 391 483	1 594 322	4 782 966	39.50

Exact Hessians vs. Hessian Approximations

In the previous tests, the IPM solver is configured to use the Ipopt barrier update rule and exact Hessians of the Lagrangians. Exchanging the Ipopt rule with the LOQO rule or replacing exact Hessians with approximated ones using SR1 or PSB updates, the IPM solver finds the same optimal solutions leading to the same developments of the double integrator. In the following, these six configurations of the IPM solver (Ipopt or LOQO combined with exact or SR1 or PSB) are compared to each other. For this, the tree-sparse problem (7.10)–(7.12) is considered for a fixed prediction horizon $T = 12$ and increasing numbers of branching levels $0 \leq T_b \leq 12$. Each problem is solved measuring the complete IPM solution time as well as the accumulated time to evaluate the NLP data, which includes evaluating the Hessian updates in the quasi-Newton approach. Each test is run 50 times for different initial states, and the averages of the IPM solution time, the NLP evaluation time and the numbers of iterations are computed. The resulting problems sizes are listed in Table 7.3, the stated solution times are the complete runtimes of the IPM in seconds in the standard configuration, i.e. using the Ipopt barrier update rule and exact Hessians. The computations are done in sequential on a workstation comprising 12 X5675 cores running at 3.07 GHz and 48 GB of RAM.

Figure 7.4 illustrates the performance results for the different configurations. In all diagrams, the x-axes indicate the number of branching levels T_b. First, fixing the evaluation of the Hessian of the Lagrangian and comparing both barrier update rules with each other, the Ipopt rule usually performs better than the LOQO rule. In most cases, the resulting number of required IPM iterations (lower left diagram in Fig. 7.3) as well as the NLP evaluation time per iteration

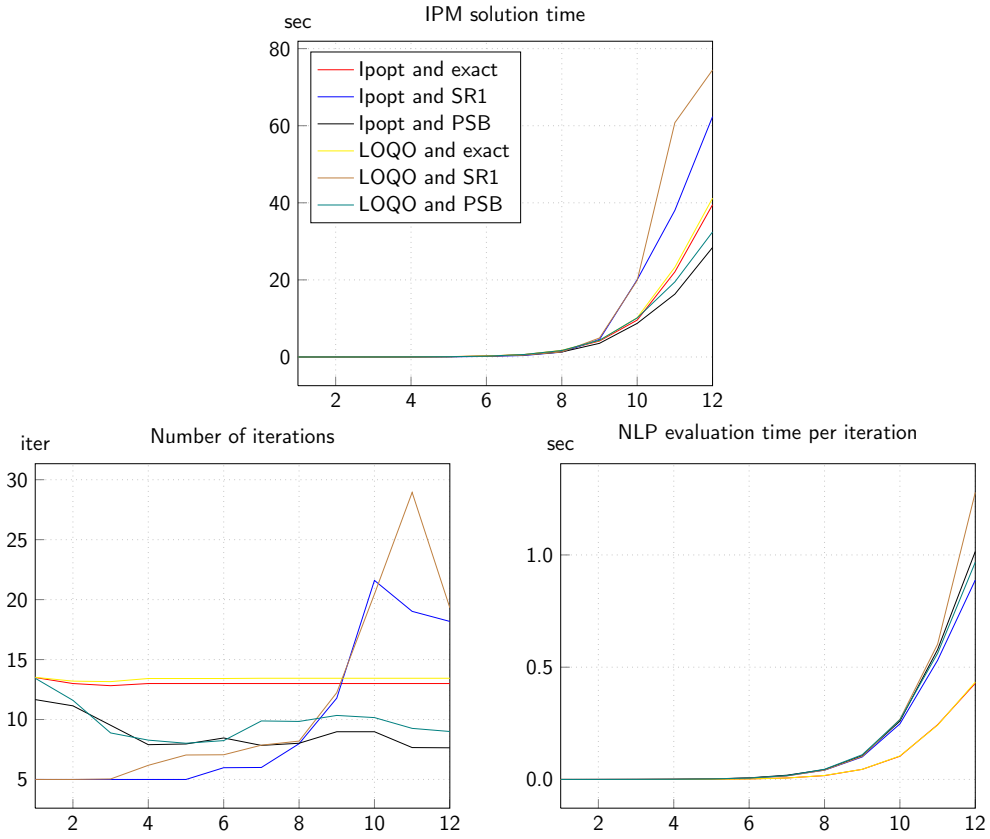


Figure 7.4.: Double Integrator – IPM performance results for different algorithm configurations and an increasing number of branching levels T_b (x-axes)

(lower right diagram) are less for the Ipopt rule. Now, fixing the barrier update rule, the following observations and conclusion can be made from the graphs in Fig. 7.4.

Exact Hessians of the Lagrangian: First, the number of required IPM iterations remains almost constant showing the scalability of the solution approach. Second, the NLP evaluation time per iteration is the lowest compared to the IPM configurations using Hessian updates. Hence, for the double integrator, evaluating exact Hessians is cheaper than approximating them in the quasi-Newton framework. Finally, let it be noted that in each IPM run, all arising KKT systems remain unmodified, i.e. no convexification is needed.

SR1 Hessian Updates: The quasi-Newton approach using the SR1 update formula starts with a low number of IPM iterations for small problems but this number increases together with the size of the problem. Compared to exact Hessians and PSB updates, SR1 updates feature the lowest numbers of iterations for small problems ($T_b < 8$) and the highest numbers for large problems ($T_b > 9$). Analogously to the case of using exact Hessians

of the Lagrangian, the arising KKT systems are solved without using convexifications indicating that the tree-sparse SR1 updates provide good approximations reflecting the curvature of the Hessian of the Lagrangian.

PSB Hessian Updates: Using the PSB formula to generate Hessian updates, the number of IPM iterations fluctuates but is in all cases significantly less than those computations using exact Hessians. This reasons the best performance results for the PSB updates on large problems ($T_b > 9$) although the evaluation of the NLP data is significantly higher than those for exact Hessians. Surprisingly, in most test runs, convexification is required to solve the corresponding optimization problems, i.e. there is at least one IPM iteration in which the KKT system is modified. The convexification parameters are determined uniformly, i.e. only the outer convexification is used (cf. Sect. 4.4.3).

7.1.3. Nonlinear Bioreactor

In the following example, a moving horizon controller regulates a bioreactor to keep the plant in a steady state of production. The considered system models a continuous flow stirred tank reactor that is proposed as a nonlinear bioreactor benchmark by Ungar [93] and studied in the context of robust model predictive control, e.g., by Lucia and Engell [59] and Lucia et al. [60]. The plant consists of a tank containing a mixture of water and cells that consume nutrients and produce (desired and undesired) products and more cells. The volume of the mixture is constant, and its composition is adjusted by a water stream that at the inlet feeds new nutrients into the tank and at the outlet contains nutrients and cells.

The dynamic model of the bioreactor is given by a set of ODEs reading

$$\dot{x}_1 = -x_1 u + x_1 (1 - x_2) e^{\frac{x_2}{\gamma}}, \tag{7.13a}$$

$$\dot{x}_2 = -x_2 u + x_1 (1 - x_2) e^{\frac{x_2}{\gamma}} \frac{1 + \beta}{1 + \beta - x_2}, \tag{7.13b}$$

where x_1 is the dimensionless cell mass, x_2 the nutrient conversion and u the flow rate of the water stream with the respective physical bounds

$$x_1(t) \in [0, 1], \quad x_2(t) \in [0, 1] \quad \text{and} \quad u(t) \in [0, 2]. \tag{7.14}$$

The equations (7.13) describe the rates of changes in the amounts of cells x_1 and nutrients x_2, respectively, that result from the respective amounts $-x_1 u$ and $-x_2 u$ carried out of the tank as well as the metabolism of the cells. The cell growth is represented by $x_1 (1 - x_2) e^{\frac{x_2}{\gamma}}$, which includes the uncertain nutrient consumption parameter γ with nominal value $\gamma^{\text{nom}} = 0.48$.

The rate of cell growth β depends on the composition of the mixture in the tank and is also represented by an uncertain parameter with nominal value $\beta^{\text{nom}} = 0.02$.

Feeding nutrients to the bioreactor with a constant flow rate u, the mixture has a Hopf bifurcation at a certain flow rate u^H that depends on the values for the parameters γ and β. For flow rates $u < u^H$, system (7.13) stabilizes at a unique fixed point (x_1^*, x_2^*), and for $u \geq u^H$, the system becomes unstable. For the nominal parameter values, the Hopf bifurcation occurs at the flow rate $u^H = 0.829$. This value decreases with an increasing value for γ or a decreasing value for β as shown in Fig. 7.5.

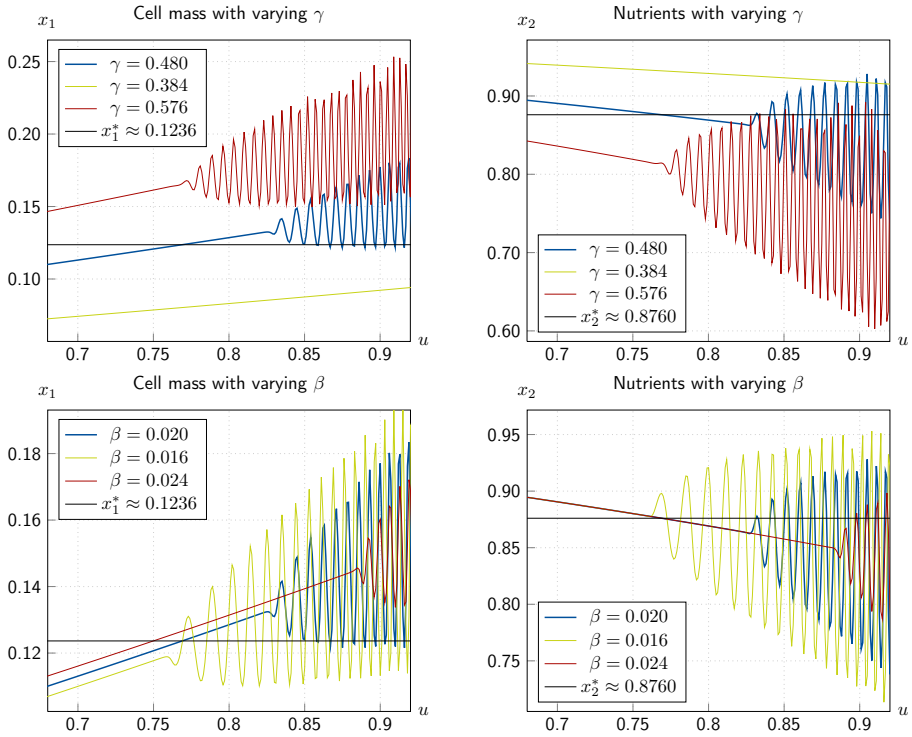


Figure 7.5.: Hopf bifurcations of the bioreactor for varying parameters γ and β

Now, the task of the controller is to keep the bioreactor in a steady state of production under perturbation of the parameters. The desired reference state $(x_1^*, x_2^*) \approx (0.1236477, 0.8760318)$ is close to the Hopf bifurcation and obtained for the constant flow rate $u^* = 0.769$ in case of the nominal parameter values. The perturbations in the parameters are assumed to be normal distributed with the respective nominal values as expected values and small variances, i.e. 0.005 for the distribution of γ and 0.001 for the distribution of β. Standard quadratic costs are applied that penalizes deviations from the reference state x_1^* with the factor 200 and changes in the flow rate with the factor 75 [59].

Tree-Sparse Problem Formulation

The optimization problem of the bioreactor is formulated as a TSP with incoming controls (4.4). For this, the state variables $x_j \in \mathbb{R}^3$ comprise the two states of the ODE (7.13) as well as the flow rate, and the tree-sparse controls $u_j \in \mathbb{R}$ are the changes in the flow rate, i.e.

$$x_j = (x_1(t(j)), x_2(t(j)), u(t(j)))^T \quad \text{and} \quad u_j = \Delta u(t(j)), \tag{7.15}$$

where $t(j)$ denotes the time corresponding to node $j \in V$. The resulting objective function incorporating the applied costs then reads

$$\phi_{ij}(x_i, u_j) = \frac{p_j}{2} u_j^T R u_j \quad \text{and} \quad \phi_j(x_j) = \frac{p_j}{2}(x_j - x^{\text{ref}})^T Q (x_j - x^{\text{ref}}) \tag{7.16}$$

with $R = 75$, $Q = \text{Diag}(200, 0, 0)$ and $x^{\text{ref}} = (x_1^*, x_2^*, 0)^T$. The dynamics of the problem has the form $g_j(x_i, u_j) = (\Psi_j(x_i, u_j), x_{i,2} + u_j)^T$ with $\Psi_j(x_i, u_j)$ representing the solution of the initial value problem in integral form,

$$\Psi_j(x_i, u_j) = \int_{t_j^s}^{t_j^e} \begin{pmatrix} -x_1(t)(x_{i,2} + u_j) + x_1(t)(1 - x_2(t))e^{\frac{x_2(t)}{\gamma_j}} \\ -x_2(t)(x_{i,2} + u_j) + x_1(t)(1 - x_2(t))e^{\frac{x_2(t)}{\gamma_j}} \frac{1+\beta_j}{1+\beta_j - x_2(t)}, \end{pmatrix} dt, \tag{7.17}$$

where $(x_1(t_j^s), x_2(t_j^s)) = (x_{i,0}, x_{i,1})$. The start time t_j^s is the time $t(j)$ for node $j \in V$, and the end time is $t_j^e = t(k)$ for the successor $k \in S(j)$. The physical bounds (7.14) are incorporated as simple bounds by setting $b_{lj}^x = (0, 0, 0)^T$ and $b_{uj}^x = (1, 1, 2)^T$. The scenario tree of the tree-sparse problem is a fan as shown in Fig. 7.6 and is characterized by its depth (T) as well as the number of branchings at the root (n_c^r) coinciding with the number of scenarios. Hence, uncertainties are only considered in the first time period of the optimization problem.

Keep a Steady State of Production

In the following, the bioreactor is run for a fixed setup and regulated by the MHC using different optimization problems varying in the length of the prediction horizon T and in the discretization of the uncertainty, i.e. in the number of scenarios n_c^r. In total, 13 cases of different optimization problems are considered and the performance of the MHC with respect to the incorporated optimization problem is analyzed.

The fixed setup of the bioreactor is as follows. First, small perturbations in the parameter γ lead to larger deviations from the reference state x_1, and, thus, to higher costs than small perturbations in β. For the subsequent tests, only the parameter γ is assumed to be uncertain while β is fixed to its nominal value. Second, starting in the reference state (x_1^*, x_2^*), the

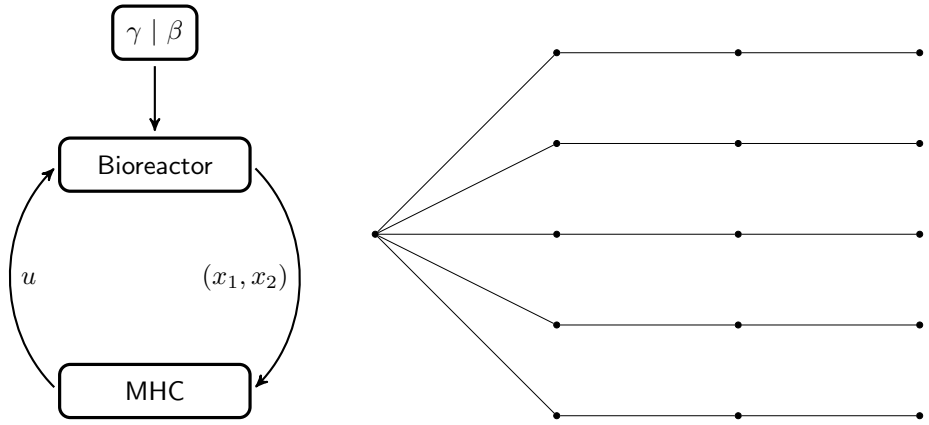


Figure 7.6.: Process control of the bioreactor (left) and scenario tree considered in the optimization problem with prediction horizon $T = 3$ and number of branchings at the root $n_c^r = 7$ (right)

bioreactor is run for 40 s and samples are taken every 0.1 s. At each sampling time, the plant receives a new control signal from the MHC and the value of the parameter γ is determined from a random experiment (see Fig. 7.6). Thus, each test run of the bioreactor includes solving 400 optimization problems. Third, each test is run 50 times for different outcomes of the random experiment, and the average accumulated costs, the means and variances of the cell mass (x_1) as well as the average flow rates (u) are computed.

The tree-sparse problems in the MHC are solved using the IPM solver with the Ipopt barrier update rule. Solutions of the IVP (7.17) are computed with the integrator Metanb and Hessian approximations are generated with tree-sparse Hessian updates based on the SR1 formula, which turned out to be more reliable for these problems than the PSB update formula. The tree-sparse KKT solver uses only the outer convexification to modify the arising KKT systems (cf. Sect. 4.4.3)

In some test runs, the MHC fails to regulate the bioreactor because the IPM solver does not find a solution for all 400 optimization problems showing that these problems are difficult to solve. To keep the results comparable, all test runs are excluded from the results that do not succeed in all 13 cases of the considered optimization problems. In the end, the means and variances are computed for 28 different outcomes of the random experiment.

Now, the performance results of the bioreactor are presented in figures 7.7 and 7.8. The diagrams in these figures illustrate the average accumulated costs (upper left), the mean of the cell mess x_1 (upper right), the variance of x_1 (lower left) and the average flow rates (lower right) with respect to the time (x-axes). Two situations are shown in these figures. First, for the results in Fig. 7.7, the bioreactor is regulated by the MHC using deterministic optimization problems ($n_c^r = 1$) and an increasing prediction horizon T. Second, for the results in Fig. 7.8,

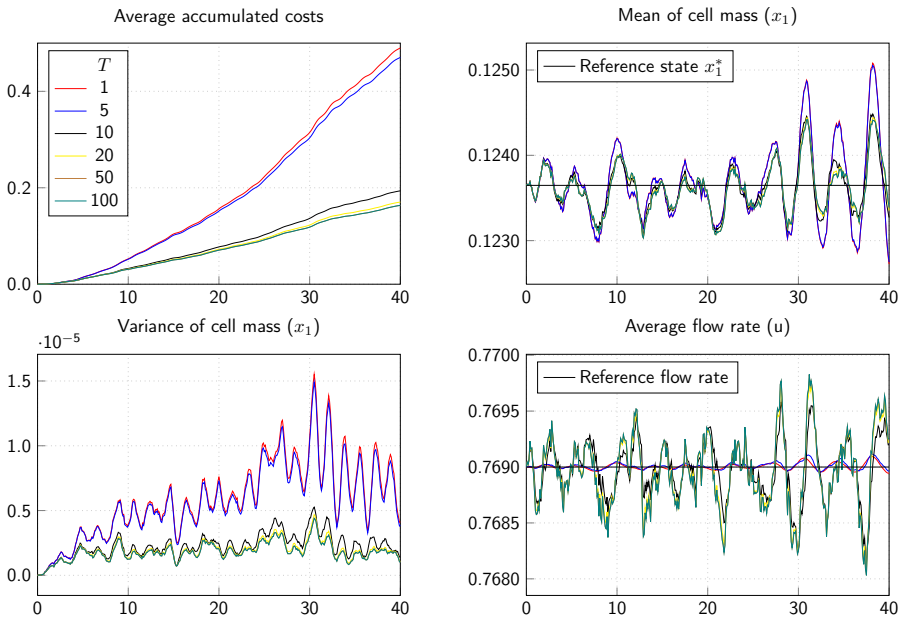


Figure 7.7.: Performance of the bioreactor for deterministic problems ($n_c^r = 1$) in the MHC with increasing predictions horizons (T)

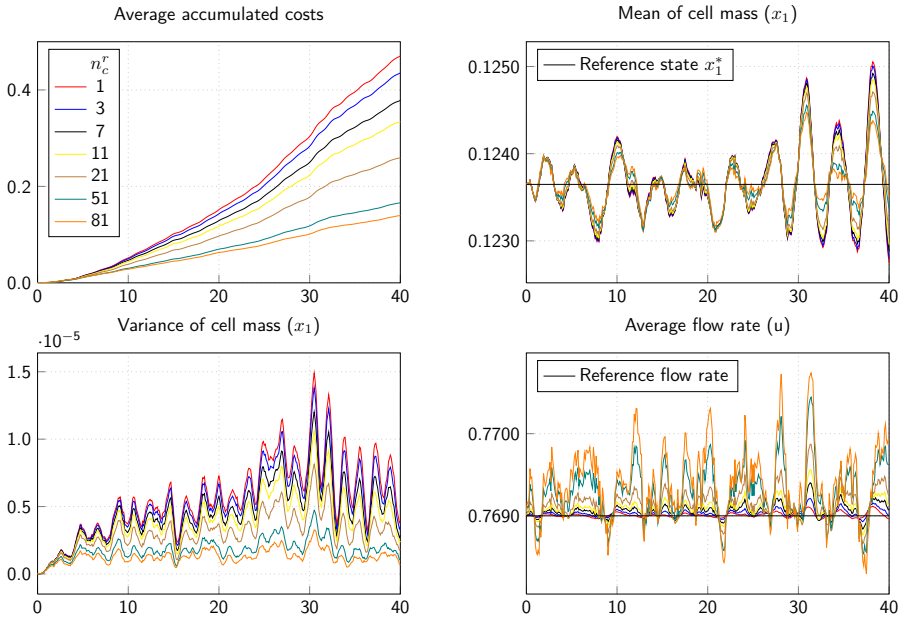


Figure 7.8.: Performance of the bioreactor for stochastic problems in the MHC with fixed prediction horizon $T = 5$ and increasing number of scenarios (n_c^r)

Table 7.4.: Problem sizes, IPM solution times (ms) and NLP evaluation times (ms) corresponding to the problems in Fig. 7.9

No.	T	n_c^r	Var.	Equ.	Bnd.	Iter.	IPM	NLP	IPM/Iter.	NLP/Iter.
1	100	1	404	303	606	8	15.475	11.365	1.753	1.298
2	5	51	1024	768	1536	9	37.178	28.155	4.113	3.118
3	10	3	124	93	186	6	4.925	3.042	0.709	0.438

the MHC uses stochastic optimization problems fixating the prediction horizon ($T = 5$) and varying the number of considered scenarios n_c^r.

In both situations, increasing the free tree parameter (T or n_c^r) improves the performance of the bioreactor. The average accumulated costs decrease monotonously, which is achieved by damping the effect of the perturbation on the cell mass x_1. After experiencing disturbances, x_1 oscillates around the reference state x_1^*. The amplitude of this oscillation is reduced by adjusting the flow rate u leading to fewer costs caused by the deviations of x_1.

Now, the results in Fig. 7.7 show that a minimal length of the prediction horizon is required to affect the performance significantly. Using the horizon $T = 5$ reduces the costs by less than 5 % whereas the prediction horizon $T = 10$ leads to a reduction of costs of more than 60 % with respect to $T = 1$. Doubling the length of the prediction horizon, i.e. setting $T = 20$, reduces the costs by 65 % in total. Hence, additional 5 % are saved. After that, increasing the length of the horizon has no significant affect on the performance of the bioreactor.

Next, the results in Fig. 7.8 show that including uncertainties in the optimization problem leads to better performance results of the bioreactor. Moreover, an increasing number of scenarios (n_c^r) also improves the performance. The total costs are reduced by 7 % for $n_c^r = 3$ considered scenarios, and considering $n_c^r = 81$ scenarios reduces the costs by 70 %. However, each additional scenario causes higher computational costs for solving the optimization problem while the relative reduction of costs per scenario decreases, e.g. it is 3.5 % reduction of cost per scenario for $n_c^r = 3$ and 0.8 % per scenario for $n_c^r = 81$.

Comparing the results in figures 7.7 and 7.8, the goal of reducing the costs is achieved in both situations. However, saving costs by increasing the number of scenarios n_c^r in the second situation is computationally more expensive than enlarging the prediction horizon T in the first situation as it is shown in Table 7.4. The optimization problems corresponding to item 1 ($T = 100$ and $n_c^r = 1$) are significantly smaller and solved faster than those corresponding to item 2 ($T = 5$ and $n_c^r = 51$), while both problems lead to almost the same performance results shown in Fig. 7.9. Finally, item 3 in Table 7.4 represents a fair comprise between the length of the prediction horizon ($T = 10$) and the incorporation of uncertainties ($n_c^r = 3$). Comparing the accumulated costs in Fig. 7.9, this compromise performance best. Moreover, this problem is

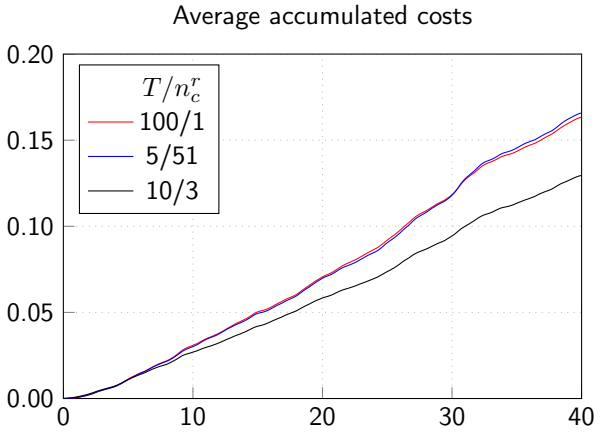


Figure 7.9.: Performance of the bioreactor – Large prediction horizon vs. large number of scenarios

smallest of those listed in Table 7.4 and is solved 3 times faster than item 1 as well as 9 times faster than item 2.

7.2. Parallel Performance of Distributed Algorithms

This section presents computational results for the approach of distribution discussed in Chap. 5. For this, huge-scale portfolio selection problems are solved using Clean::IPM, and the parallel performance of the tree-sparse algorithms and the interior-point solver is analyzed. Parts of the presented results are also going to be published in [48].

7.2.1. Test Environment

Parallel Platform

All subsequent computations are done on the compute cluster milet of the Institute of Applied Mathematics at the Leibniz Universität Hannover. The cluster consists of 8 compute nodes with 48 GB of RAM each. Four compute nodes comprise 8 Intel(R) Xeon(R) X5570 cores running at 2.93 GHz, the other four nodes consists of 12 of the newer X5675 cores running at 3.07 GHz. In total, the cluster comes with 80 cores and 384 GB of RAM and has an InfiniBand interconnect. One core is already working at full capacity on one process. Hence, for the subsequent computations, no more processes are used than available cores.

Measures of Parallel Performance

The performance of parallel algorithms is measured by means of *speedup* S_{n_p} and *efficiency* E_{n_p},

$$S_{n_p} := \frac{t_{\text{seq}}}{t_{n_p}} \quad \text{and} \quad E_{n_p} := \frac{S_{n_p}}{n_p}, \tag{7.18}$$

where t_{seq} is the runtime of the sequential algorithm and t_{n_p} is the runtime of the parallel algorithm on n_p processes. Neglecting inaccuracies in measuring computing time, the speedup is bounded by the number of used processes and considered as perfect if it is linear, i.e. $S_{n_p} = n_p$. Hence, perfect efficiency is attained at $E_{n_p} = 1$. For more details on performance analysis of parallel programs the reader is referred to standard textbooks such as [70].

For the numerical results, the wall-clock times $t_{n_p}^{\text{wc}}$ and the *accumulated* CPU times $t_{n_p}^{\text{cpu}}$ of the respective algorithms are monitored. The algorithm is executed several times and the best result is taken for each setting to determine the speedups and efficiencies. The speedup is computed with respect to the wall-clock time by

$$S_{n_p} = \frac{t_{\text{ref}}^{\text{wc}}}{t_{n_p}^{\text{wc}}} \quad \text{with} \quad t_{\text{ref}}^{\text{wc}} = \min_{n_p}\{n_p \cdot t_{n_p}^{\text{wc}}\}, \tag{7.19}$$

and the efficiency is computed with respect to the CPU time by

$$E_{n_p} = \frac{t_{\text{ref}}^{\text{cpu}}}{t_{n_p}^{\text{cpu}}} \quad \text{with} \quad t_{\text{ref}}^{\text{cpu}} = \min_{n_p} t_{n_p}^{\text{cpu}}. \tag{7.20}$$

The wall-clock reference time $t_{\text{ref}}^{\text{wc}}$ is not the runtime of the sequential algorithm but the best result when scaling the runtimes $t_{n_p}^{\text{wc}}$ with the corresponding numbers of processes n_p. This way, computing the speedup remains consistent for test cases where sequential runtimes are not available and, moreover, superlinear speedup is avoided.

Configuration of the IPM Algorithm

Clean::IPM is configured in a standard way to solve the convex quadratic problems in portfolio optimization, meaning that no globalization strategy is applied, the MPC barrier update rule is used and the emergency mode is disabled (cf. Table 7.1).

7.2.2. Huge-Scale Portfolio Optimization Problems

The problems solved in this section are portfolio optimization problems as outlined in Sect. 3.1.3. For this, artificial portfolios are considered that comprise n_a different assets of risks classified into the three categories low, medium and high. It is assumed that the development of the

stock prices is characterized by an n_a-dimensional geometric Brownian motion, and this process is discretized in time following the lines of [54]. In doing so, at each time step t of the planning horizon, the return of the portfolio is multidimensional lognormal distributed with μ_t and Σ_t as first and second moments, respectively. For each node j in the scenario tree, the corresponding moments are perturbed randomly, i.e. $\mu_j = \mu_{t(j)} + \delta_{\mu_j}$ and $\Sigma_j = \Sigma_{t(j)} + \delta_{\Sigma_j}$, and the perturbed distributions are approximated by discrete ones. Obtaining a discrete approximation with the same first and second moments of the distribution, i.e. an approximation satisfying

$$\mu_j = \sum_{k \in S(j)} r_k \quad \text{and} \quad \Sigma_j = \sum_{k \in S(j)} (r_k - \mu_j)(r_k - \mu_j)^T \quad \text{for all} \quad j \in V, \tag{7.21}$$

one needs to consider at least $n_a + 1$ events of returns.

The tree-sparse formulation of the portfolio selection problem reads [78]:

$$\begin{aligned}
\min_{u,x} \quad & \sum_{j \in L} p_j x_j^T \Sigma_j x_j && && \text{(7.22a)}\\
\text{s.t.} \quad & x_j = \operatorname{Diag}(r_j) x_i + u_j^+ - u_j^-, && j \in V, && \text{(7.22b)}\\
& B_j x_j \in [b_{lj}, b_{uj}], && j \in V, && \text{(7.22c)}\\
& u_j^+, u_j^- \geq 0, && j \in V, && \text{(7.22d)}\\
& x_j \geq 0, && j \in V, && \text{(7.22e)}\\
& \sum_{j \in L} p_j \bar{\mu}_j^T x_j = \rho. && && \text{(7.22f)}
\end{aligned}$$

The state variables $x_j \in \mathbb{R}^{n_a}$ represent the amount of money invested in the respective assets. Money transfers between the assets are modeled as controls $u_j \in \mathbb{R}^{2n_a}$ split into buys u_j^+ and sells u_j^-. The portfolio selection problem includes several composition constraints, e.g. at least 30 % of the money should be invested in assets with low risk. These composition constraints lead to the linear inequalities (7.22c) that are required to hold at each node $j \in V$. The portfolio selection problem (7.22) is a convex tree-sparse QP in incoming control form with the matrix node subblocks (cf. Sect. 4.2.3)

$$G_j = \operatorname{Diag}(r_j), \quad E_j = \begin{bmatrix} I & -I \end{bmatrix} \quad \text{and} \quad F_j^r = B_j \quad \text{for} \quad j \in V \tag{7.23}$$

as well as

$$H_j = 2 p_j \Sigma_j \quad \text{and} \quad F_j = p_j \bar{\mu}_j^T \quad \text{for} \quad j \in L. \tag{7.24}$$

Table 7.5.: Portfolio test collections PT1 and PT2

No.	d	n_a	Nodes	Scenarios	Variables	Equalities	Inequalities
				PT1			
1	11	3	$5.59 \cdot 10^6$	$4.19 \cdot 10^6$	$5.03 \cdot 10^7$	$1.68 \cdot 10^7$	$1.68 \cdot 10^7$
2	9	4	$2.44 \cdot 10^6$	$1.95 \cdot 10^6$	$2.93 \cdot 10^7$	$9.77 \cdot 10^6$	$1.22 \cdot 10^7$
3	8	5	$2.02 \cdot 10^6$	$1.68 \cdot 10^6$	$3.02 \cdot 10^7$	$1.01 \cdot 10^7$	$1.21 \cdot 10^7$
4	7	7	$2.40 \cdot 10^6$	$2.10 \cdot 10^6$	$5.03 \cdot 10^7$	$1.68 \cdot 10^7$	$1.68 \cdot 10^7$
5	6	9	$1.11 \cdot 10^6$	$1.00 \cdot 10^6$	$3.00 \cdot 10^7$	$1.00 \cdot 10^7$	$1.00 \cdot 10^7$
6	5	14	$8.14 \cdot 10^5$	$7.59 \cdot 10^5$	$3.42 \cdot 10^7$	$1.14 \cdot 10^7$	$1.06 \cdot 10^7$
7	4	23	$3.46 \cdot 10^5$	$3.32 \cdot 10^5$	$2.39 \cdot 10^7$	$7.96 \cdot 10^6$	$6.58 \cdot 10^6$
				PT2			
1	12	3	$2.24 \cdot 10^7$	$1.68 \cdot 10^7$	$2.01 \cdot 10^8$	$6.71 \cdot 10^7$	$6.71 \cdot 10^7$
2	10	4	$1.22 \cdot 10^7$	$9.77 \cdot 10^6$	$1.46 \cdot 10^8$	$4.88 \cdot 10^7$	$6.10 \cdot 10^7$
3	8	7	$1.92 \cdot 10^7$	$1.68 \cdot 10^7$	$4.03 \cdot 10^8$	$1.34 \cdot 10^8$	$1.34 \cdot 10^8$
4	7	9	$1.11 \cdot 10^7$	$1.00 \cdot 10^7$	$3.00 \cdot 10^8$	$1.00 \cdot 10^8$	$1.00 \cdot 10^8$
5	6	12	$5.23 \cdot 10^6$	$4.83 \cdot 10^6$	$1.88 \cdot 10^8$	$6.27 \cdot 10^7$	$5.75 \cdot 10^7$
6	5	19	$3.37 \cdot 10^6$	$3.20 \cdot 10^6$	$1.92 \cdot 10^8$	$6.40 \cdot 10^7$	$5.39 \cdot 10^7$
7	4	33	$1.38 \cdot 10^6$	$1.34 \cdot 10^6$	$1.36 \cdot 10^8$	$4.54 \cdot 10^7$	$3.44 \cdot 10^7$

Portfolio Test Collections

The scenario trees corresponding to the tree-sparse portfolio problems (7.22) are specific ones featuring the same branching $n_a + 1$ at each node $j \in V \setminus L$. Thus, the size of a problem (7.22) is characterized by the number of assets (n_a) as well as by the depth of the scenario tree (d) corresponding to the time discretization of the planning horizon. In the following, two portfolio test collections PT1 and PT2 (shown in Table 7.5) are considered that are designed for one and eight compute nodes, respectively.

In the implementation used here, no local sparsities are exploited. The matrix node subblocks (7.23) and (7.24) are stored in dense storage format (cf. Sect. 6.4.4), and each entry in these subblocks is considered as a nonzero. The resulting numbers of nonzero entries in the tree-sparse KKT matrices and the corresponding matrix factors are listed in Table 7.6. In the implementation, the identity block for the states x_j in the dynamics (7.22b) is not stored, hence there are more nonzero entries in total than stored. Furthermore, since storing only those node subblocks that are modified during the factorization, the factors require less memory than the corresponding KKT matrices; the node subblocks (7.23) are not duplicated for the factors.

All tests are chosen such that they are the largest of their kind, i.e. increasing one of the parameters n_a or d lead to problems that do not fit into the provided memory resources any longer. The tests in Table 7.5 are sorted decreasingly with respect to the tree depth d, which

Table 7.6.: PT1 and PT2 – Nonzero entries in KKT matrices and their factors

No.	Nodes	Scenarios	Nnz Matrix (total)	Nnz Matrix (stored)	Nnz Factors (stored)
			PT1		
1	$5.59 \cdot 10^6$	$4.19 \cdot 10^6$	$5.20 \cdot 10^8$	$5.03 \cdot 10^8$	$3.02 \cdot 10^8$
2	$2.44 \cdot 10^6$	$1.95 \cdot 10^6$	$3.96 \cdot 10^8$	$3.86 \cdot 10^8$	$2.20 \cdot 10^8$
3	$2.02 \cdot 10^6$	$1.68 \cdot 10^6$	$4.94 \cdot 10^8$	$4.84 \cdot 10^8$	$2.72 \cdot 10^8$
4	$2.40 \cdot 10^6$	$2.10 \cdot 10^6$	$1.09 \cdot 10^9$	$1.07 \cdot 10^9$	$6.04 \cdot 10^8$
5	$1.11 \cdot 10^6$	$1.00 \cdot 10^6$	$8.20 \cdot 10^8$	$8.10 \cdot 10^8$	$4.50 \cdot 10^8$
6	$8.14 \cdot 10^5$	$7.59 \cdot 10^5$	$1.41 \cdot 10^9$	$1.40 \cdot 10^9$	$7.69 \cdot 10^8$
7	$3.46 \cdot 10^5$	$3.32 \cdot 10^5$	$1.57 \cdot 10^9$	$1.56 \cdot 10^9$	$8.60 \cdot 10^8$
			PT2		
1	$2.24 \cdot 10^7$	$1.68 \cdot 10^7$	$2.08 \cdot 10^9$	$2.01 \cdot 10^9$	$1.21 \cdot 10^9$
2	$1.22 \cdot 10^7$	$9.77 \cdot 10^6$	$1.98 \cdot 10^9$	$1.93 \cdot 10^9$	$1.10 \cdot 10^9$
3	$1.92 \cdot 10^7$	$1.68 \cdot 10^7$	$8.72 \cdot 10^9$	$8.59 \cdot 10^9$	$4.83 \cdot 10^9$
4	$1.11 \cdot 10^7$	$1.00 \cdot 10^7$	$8.20 \cdot 10^9$	$8.10 \cdot 10^9$	$4.50 \cdot 10^9$
5	$5.23 \cdot 10^6$	$4.83 \cdot 10^6$	$6.68 \cdot 10^9$	$6.62 \cdot 10^9$	$3.67 \cdot 10^9$
6	$3.37 \cdot 10^6$	$3.20 \cdot 10^6$	$1.05 \cdot 10^{10}$	$1.04 \cdot 10^{10}$	$5.76 \cdot 10^9$
7	$1.38 \cdot 10^6$	$1.34 \cdot 10^6$	$1.26 \cdot 10^{10}$	$1.26 \cdot 10^{10}$	$6.95 \cdot 10^9$

implies an increasing order with respect to the number of assets n_a and, thus, the computational workload per node.

7.2.3. Parallel Performance of the Tree-Sparse Algorithms and the IPM

For the sake of a cheap scenario generation, the requisites of the second moments in (7.21) are neglected[1]. The numerical results here are based on scenario generations that only preserve the means of the distributions. Same results are obtained when using approximations for the distributions that also preserve the second moments (see sections A.2 and A.3).

The IPM solves the respective problems in the collections PT1 and PT2 upon optimality. Time measurements are taken for the *complete runtime* the algorithm spends in its iterative stage, i.e. the overall runtime without the initialization stage, and for the runtimes of the tree-sparse algorithms, i.e. the three stages of the KKT solution procedure and the matrix-vector product. The remaining operations of the iterative stage, e.g. vector operations and scalar operations, are summarized into the *iteration time*. Table 7.7 lists the resulting wall-clock reference times for the four tree-sparse algorithms and the iteration.

As designed, the computations for the problems in PT1 are done on one compute node of the

[1] Discretizations of the continuous distributions with the same second moments are obtained, for example, by solving a suitable feasibility problem for each node in the tree. For the problems in the test collection PT2, this approach of scenario generation takes up to several days.

Table 7.7.: PT1 and PT2 – Wall-lock reference times ($t_{\text{ref}}^{\text{wc}}$ in s) for the factorization (Fact.), inward substitution (In), outward substitution (Out), the matrix-vector product (MVP) and the iteration time (Iter.)

	PT1					PT2				
No.	Fact.	In	Out	MVP	Iter.	Fact.	In	Out	MVP	Iter.
1	12.95	3.60	3.49	1.37	29.91	56.88	13.23	15.71	5.78	126.97
2	8.90	1.87	2.43	0.75	17.59	45.42	9.58	10.73	3.83	91.23
3	10.67	1.98	1.99	0.72	18.32	215.82	29.70	27.60	9.15	246.95
4	24.32	3.63	3.32	1.12	29.97	196.74	21.94	21.21	6.37	180.43
5	19.32	2.14	2.10	0.63	17.75	195.80	14.17	15.35	3.80	114.91
6	42.90	2.59	2.76	0.68	20.56	414.33	16.71	17.58	4.15	123.49
7	64.38	2.27	2.39	0.55	15.53	756.12	17.42	17.89	4.60	103.72

Table 7.8.: PT1 and PT2 – Number of iterations and IPM solution times ($t_{n_p}^{\text{wc}}$ in s) using one compute node (1) and eight compute nodes (8) of the cluster as well as various numbers of processes $n_p \in \{1, 2, 8, 12, 16, 64\}$

	PT1 (1)					PT1 (8)				PT2 (8)		
No.	Iter.	1	2	8	12	Iter.	8	16	64	Iter.	16	64
1	13	912	447	150	120	13	117	66	23	13	241	86
2	13	544	270	92	79	13	71	38	14	14	188	72
3	11	495	248	85	70	11	64	33	12	10	426	177
4	9	809	375	126	102	9	98	53	23	9	319	119
5	6	351	172	59	49	6	45	23	10	7	188	73
6	8	766	358	117	96	8	93	47	17	6	361	119
7	6	716	341	106	80	6	90	45	14	8	552	178

cluster and those for the problems in PT2 on all eight compute nodes, respectively. Additionally, the problems in PT1 are also solved using the available resources of all eight computes nodes. The required numbers of iterations as well as the resulting solution times, which also include the initialization phase of the IPM, are listed in Table 7.8. Using all available resources, the smaller problems in PT1 with 10^7 variables and constraints are solved in 23 seconds or less. The huge problems in PT2 with 10^8 variables and constraints are solved in less than three minutes. Moreover, solving problems with a heavy workload per tree node tends to require less iterations than solving those with a smaller workload.

The data in Table 7.8 show the scalability of the approach of distribution in three ways. First, with more resources available larger, problems can be solved. Second, with more resources available, the same problems can be solved in less time. Third, the number of required IPM iterations does not depend on the amount of resources used to solve the problem. The numbers are the same when solving PT1 on one compute as well as on all eight compute nodes. Furthermore, the number of required iterations does not depend on the size of the problem but mostly on the workload per node. The numbers are approximately the same when solving

problems in the PT collections with a similar workload, e.g. for portfolios comprising a similar number of assets n_a.

Finally, the solution times in Table 7.8 for PT1 using 8 processes are significantly better when employing eight compute nodes for the computations instead of one. Running 8 processes on the same compute node, these share common hardware components such as memory accesses as well as network and communication resources, which they do not when each process runs on another compute node.

In the following, the parallel performance of the tree-sparse algorithms is analyzed first, the complete optimization solver is investigated afterwards. Multistage stochastic problems are known to be well-suitable for parallelization [109] and good performance results of similar approaches, i.e. direct methods for solving the KKT systems arising in stochastic optimization, are reported, e.g., by Gondzio and Grothey [34] and by Blomvall [10]. Hence, it can be expected that the tree-sparse algorithms perform well and best results are obtained for the factorization with an almost linear speedup. The performance of the IPM is expected to be dominated by the KKT solution but may feature worse performance results since the optimization algorithm includes several synchronization points and sequential parts.

Parallel Performance of the Tree-Sparse Algorithms

Figure 7.10 shows the parallel performance results of the tree-sparse algorithms for the problems in the test collection PT2[2]. In all diagrams, the x-axes indicate the number of used processes and the first angle bisectors in the speedup plots represent the linear speedup.

As expected, the tree-sparse factorization achieves the best performance results with almost linear speedup and efficiencies in the range from 90 % to 100 %. This is consistent with the reports in [34] and [10]. With one Cholesky factorization and several level 3 BLAS operations (matrix-matrix products) performed for each node in the tree, the factorization features a heavy workload, letting the communication times and the idle times become relatively small.

The performance of the substitutions is significantly worse compared to the factorization but still good with most of the efficiencies in the range from 70 % to 100 %. This also conforms with the reports in [34]. In contrast to the factorization, the workload of a tree-sparse substitution is small and consists only of level 1 and level 2 BLAS operations (vector and matrix-vector operations). Comparing the two substitutions with each other, the outward substitution shows

[2]The tests in collection PT2 start with two processes per compute node instead of one. This is not due to too less memory resources or higher memory requirement when using less processes, but because the lengths of the data arrays for the KKT matrices and its factors exceed the integer range used in BLAS and LAPACK routines.

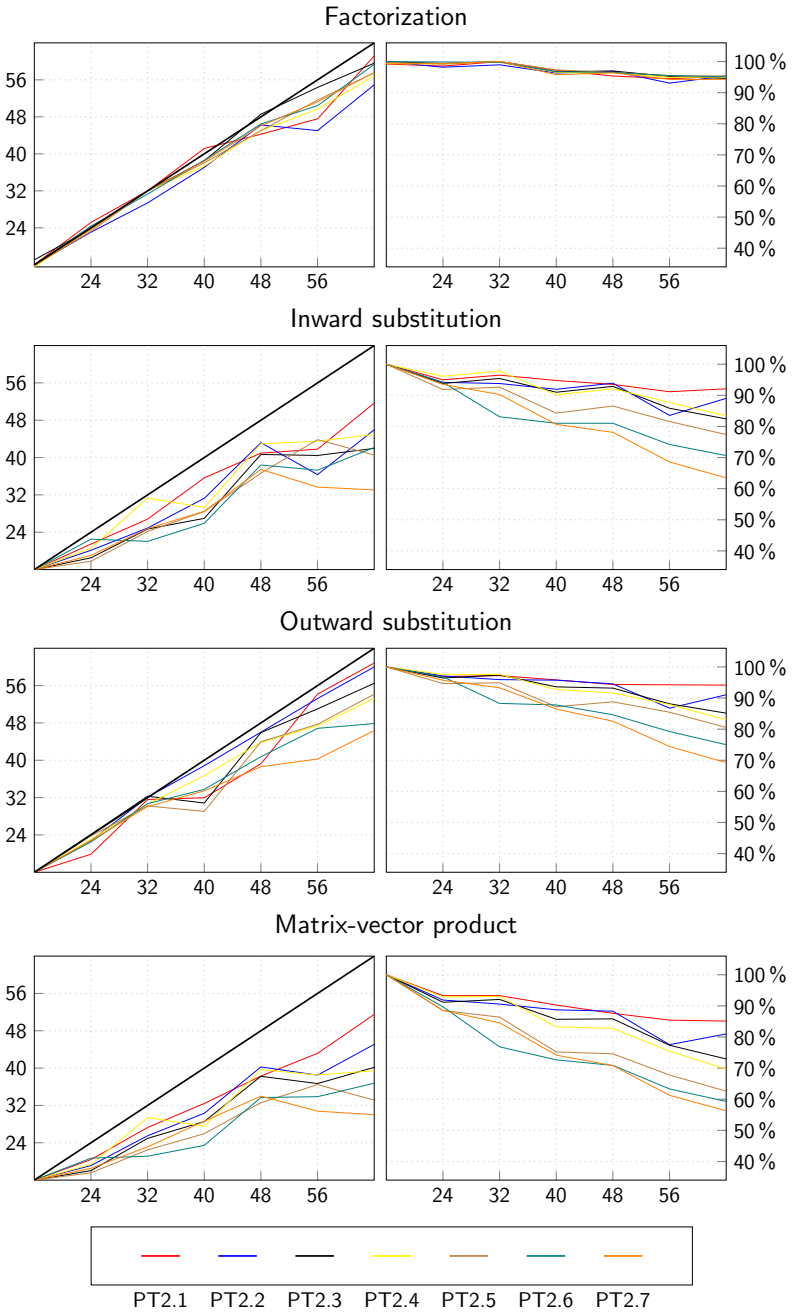


Figure 7.10.: PT2 – Speedups (left) and efficiencies (right) of the tree-sparse algorithms

better performance results than the inward substitution. Both substitutions have approximately the same amount of workload per node, hence this performance discrepancy is reasoned in the underlying distributed DFS-based tree algorithm (cf. Sect. 5.3).

The matrix-vector product (MVP) offers the worst performance results of the four tree-sparse algorithms in Fig. 7.10 with efficiencies decreasing below the mark of 60 %. The development of the performance is the same as the one of the inward substitution but the efficiencies of the latter are up to 10 % higher. Both tree-sparse algorithms are inward algorithms but the MVP features less workload per node than the substitution. However, with reports of efficiencies that go down to 23 % in [34], the performance of the tree-sparse MVP is still good.

In all diagrams of Fig. 7.10, the parallel performance decreases monotonously with an increasing number of used processes except for several peaks when using an even number of processes per compute node. These bumps occur not only for the tree-sparse algorithm but also for the remaining parts of the IPM algorithms (as later shown in Fig. 7.12). Also, the investigation of the post-distribution communication reduction (cf. Sect. 7.2.4) offers no conclusion to this effect. On the used parallel platform, the cores seem to work best in pairs.

Now, Figure 7.11 shows the parallel performance results of the tree-sparse algorithms for the problems in the test collections PT1. The computations are done on one compute node of the cluster using numbers of processes in the range from 1 to 12, and also on eight compute nodes with numbers of processes in the range from 8 to 64. In Fig. 7.11, the inward substitution represents the tree-sparse algorithms with small workload per node. The performance results for the outward substitution and the MVP are supplemented in App. A.1.

First of all, the performance diagrams for PT1 in Fig. 7.11 show the same development with increasing number of used processes per compute node as the diagrams for PT2 in Fig. 7.10. The efficiencies of the factorization are still in the range from 90 % to 100 % and the performance of the inward substitution decreases monotonously when neglecting the peaks for the even number of processes per compute node. Furthermore, tree-sparse algorithms with heavy workload per node such as the factorization tend to perform better on less deep but broad trees whereas algorithms with small workload per node (the substitutions) show better results on deep trees with small branchings.

The parallel performance of the inward substitution is significantly better when employing eight compute nodes for the computations instead of one. In the latter case, the efficiencies decrease down to almost 50 % instead of 70 % when using eight compute nodes. Again, employing one compute node instead of eight means sharing common hardware resources, which could

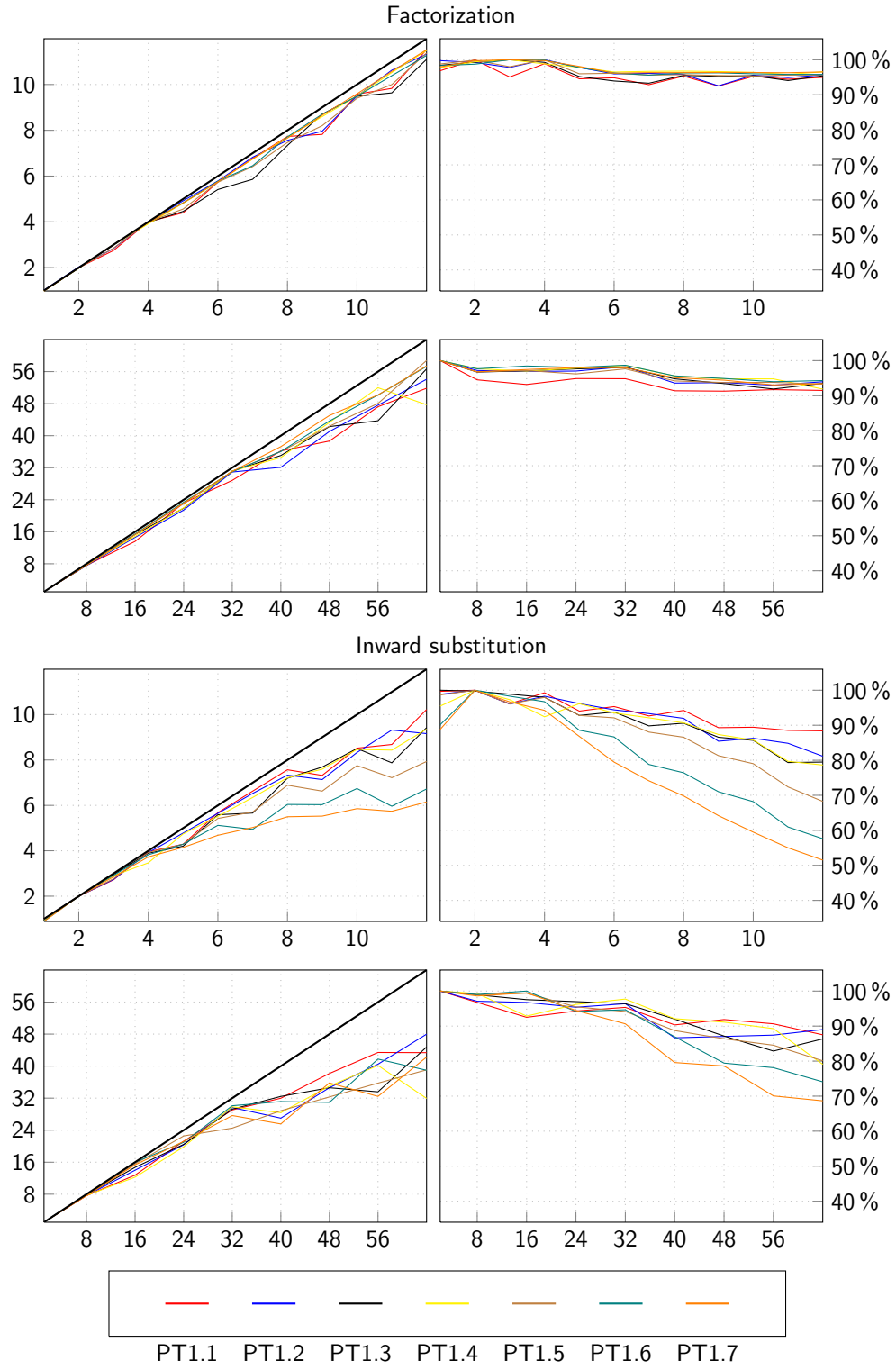


Figure 7.11.: PT1 – Speedups (left) and efficiencies (right) of the tree-sparse algorithms

Table 7.9.: PT1 and PT2 – IPM statistics with numbers of state variables (n_j^x), control variables (n_j^u) and inequalities (l_j^{rx}) per tree node, the number of IPM iterations (It.), the runtime ($t_{\text{ref}}^{\text{wc}}$ in s) of the iterative stage and the respective shares in runtime for the factorization (F), inward substitution (I), outward substitution (O) and matrix-vector product (M) as well as the remaining operations (IPM)

No.	n_j^x	n_j^u	l_j^{rx}	It.	Runtime	F (1)	I (2)	O (2)	M (6)	IPM
					PT1					
1	3	6	3	13	834	20.18 %	9.54 %	10.89 %	12.77 %	46.62 %
2	4	8	5	13	505	22.91 %	9.65 %	10.52 %	11.62 %	45.29 %
3	5	10	6	11	454	25.88 %	9.62 %	9.65 %	10.41 %	44.45 %
4	7	14	7	9	674	32.46 %	9.69 %	8.87 %	8.98 %	40.01 %
5	9	18	9	6	297	39.15 %	8.70 %	8.52 %	7.65 %	35.98 %
6	14	28	13	8	626	54.83 %	6.63 %	7.05 %	5.21 %	26.28 %
7	23	46	19	6	574	70.54 %	4.75 %	5.01 %	3.44 %	16.26 %
					PT2					
1	3	6	3	13	3,576	20.68 %	9.62 %	10.96 %	12.60 %	46.15 %
2	4	8	5	14	2,804	22.68 %	9.57 %	10.72 %	11.47 %	45.56 %
3	7	14	7	10	6,323	34.13 %	9.39 %	8.73 %	8.69 %	39.06 %
4	9	18	9	9	2,743	39.22 %	8.75 %	8.46 %	7.62 %	35.97 %
5	12	24	11	7	4,515	49.79 %	7.23 %	7.83 %	5.81 %	29.33 %
6	19	38	16	8	5,050	65.63 %	5.29 %	5.57 %	3.94 %	19.56 %
7	33	66	25	8	7,639	79.19 %	3.65 %	3.75 %	2.55 %	10.86 %

reason this effect. However, the factorization as a tree-sparse algorithm with heavy workload per tree node is not affected by this.

Parallel Performance of the IPM Algorithm

In the following, the parallel performance of the interior-point solver is analyzed. For this, Table 7.9 provides the solver statistics for both portfolio test collections. Each IPM iteration consists of one factorization, two of each of the substitutions and six matrix-vector products[3]. Table 7.9 lists the number of required IPM iterations together with the corresponding runtime in seconds as wall-clock reference time ($t_{\text{ref}}^{\text{wc}}$), i.e. the time the algorithm would spend in the iterative stage when using one process for the computations. For each tree-sparse algorithm and the remaining IPM operations, the table also lists the respective shares in that runtime of the iterative stage.

First, Table 7.9 shows that the shares of the factorization in the runtime increase with an increasing workload per node whereas the shares of the other tree-sparse algorithms and the remaining operations decrease. This reflects that the complexity of the IPM algorithm is strongly affected by the complexity of the factorization of the KKT matrix. For those

[3]Matrix-vector products are used to evaluate the function values of the linear constraint functions and the gradient of the Lagrangian.

problems in the PT collections with heavy workload per node, i.e. tests 5 to 7 in PT1 and 4 to 7 in PT2, the runtime is dominated by the tree-sparse factorization with shares in the range from 39 % to 79 %. For the other tests, the remaining IPM operations feature the biggest shares in the range from 39 % to 46 %, which then again also implies that the tree-sparse algorithms account for more than 50 % of the runtime in each considered test. Hence, it can be expected that the parallel performance of the IPM is shaped by the tree-sparse algorithms.

Now, Figure 7.12 illustrates the parallel performance of the IPM algorithm for the problems in the collection PT2. The results for the collection PT1 feature similar developments and are supplemented in App. A.1. The graphics in Figure 7.12 show the speedups and the efficiencies of the complete iterative stage of the IPM algorithm, the iteration without the tree-sparse algorithms and the tree-sparse KKT solver, which comprises the factorization and the four substitutions of each IPM iteration.

The performance of the KKT solver is usually dominated by the performance of the factorization. Best results are achieved for the problems with a heavy workload per node, e.g. PT2.6 and PT2.7, where the share of the factorization overrules by far the shares of both of the substitutions. However, the worst performance results are not obtained for those problems with the smallest workload per node. For PT2.1, for example, the KKT solver spends as much time in the substitutions as in the factorization, and the substitutions achieve their best results compared to the other problems in PT2.

Analogously to the tree-sparse algorithms, the performance of the remaining operations of the IPM iteration (shown in the diagram in the middle of Fig. 7.12) decreases monotonously with increasing number of used processes while showing performance peaks when using an even number of processes per compute node. However, the performance of the iteration is significantly worse than each of the tree-sparse algorithms with efficiencies going down to almost 50 %. This is expected since the remaining operations include synchronization points and sequential parts, hence their performance suffers from Ahmdahl's law [70]. Furthermore, there are no striking discrepancies in the performance of the IPM iteration between the problems in one collection since those results are not affected by a workload per tree node.

The performance of the complete iterative stage of the IPM algorithm (shown in the upper diagram of Fig. 7.12) results from the performance of the tree-sparse algorithms and the remaining operations. The speedups and the efficiencies of the iterative stage develops the same way as the ones of the KKT solver but the performance of the iterative stage is worse with efficiencies about 10 % less than those of the KKT solver. Hence, as expected the performance

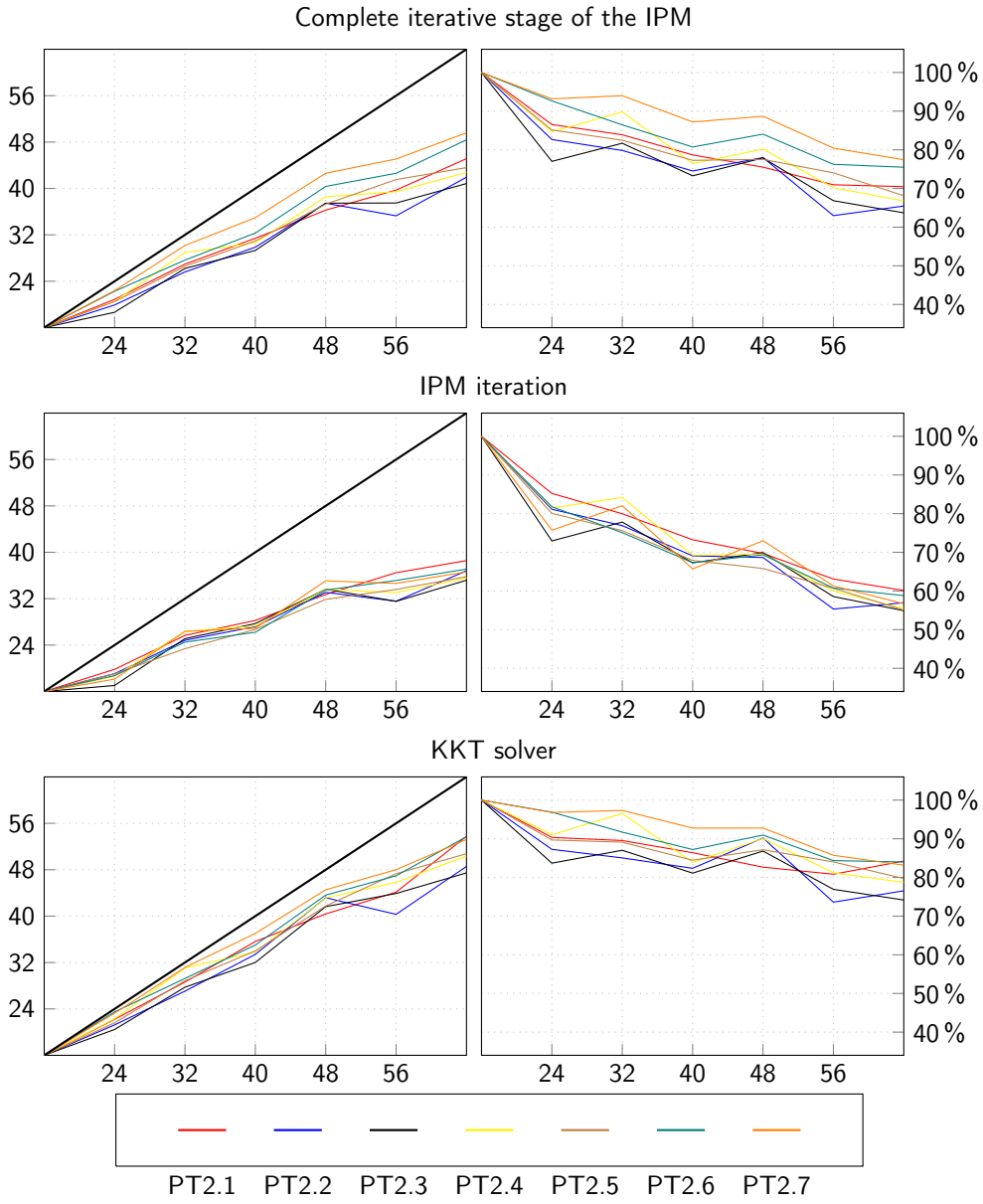


Figure 7.12.: PT2 – Speedups (left) and efficiencies (right) of the IPM

Table 7.10.: Tree collections TC1 and TC2

No.	d	b	Nodes	Scenarios	No.	d	b	Nodes	Scenarios
1	11	3	265 720	177 147	1	12	4	22 369 621	16 777 216
2	9	4	349 525	262 144	2	10	5	12 207 031	9 765 625
3	8	5	488 281	390 625	3	8	8	19 173 961	16 777 216
4	7	8	2 396 745	2 097 152	4	7	10	11 111 111	10 000 000
5	6	10	1 111 111	1 000 000	5	6	13	5 229 043	4 826 809
6	5	15	813 616	759 375	6	5	20	3 368 421	3 200 000
7	4	24	346 201	331 776	7	4	34	1 376 831	1 336 336

of the IPM is shaped by the performance of the tree-sparse algorithms but changed for the worse due to the remaining IPM operations.

7.2.4. Effect of the Post-Distribution Communication Reduction

This section analyzes the effect of the post-distribution communication (PDCR) (cf. Sect. 5.3.3) for the trees corresponding to the portfolio selection problems in Table 7.5, i.e. it investigates the number of communication calls that can be saved during one run of a distributed DFS-based tree algorithm by merging data for the same destination or by sending the same data to one process only once. After tree distribution, the numbers of senders ($|\mathcal{S}|$) and roots ($|\mathcal{R}|$) of the resulting distributed tree are counted and compared to the number of communication calls (n_c) after applying the PDCR. By definition, there is at least one communication call per sender and, without reducing the communication, each root $r \in \mathcal{R} \setminus \{0\}$ causes exactly one communication call. Hence, the number of communication calls satisfies

$$|\mathcal{S}| \leq n_c \leq |\mathcal{R}| - 1. \tag{7.25}$$

Additionally, the number of communication calls are counted for each process separately. The communications for a single process p are distinguished between *ingoing* and *outgoing communications*. Informally speaking, an ingoing communication is carried out from process p in direction to the tree root. Formally, a loose edge $e \in \mathcal{E}_p$ adjacent to the communicating root $r \in \mathcal{R}_p$ is an ingoing edge ($e = (\cdot, r)$). Outgoing communications occur along outgoing loose edges $e = (s, \cdot) \in \mathcal{E}_p$ adjacent to senders $s \in \mathcal{S}_p$. For both ingoing and outgoing communications, the respective maxima over all processes ($n^{\text{in}}_{\max}$ and $n^{\text{out}}_{\max}$) are computed.

The trees corresponding to the problems in the portfolio test collections PT1 and PT2 (cf. Sect. 7.2.2) feature the same branching b at each node $j \in V \setminus L$, and they are characterized by this branching and the tree depth (d). These trees are now arranged in the tree collections TC1 and TC2, respectively, and listed in Table 7.10. Figure 7.13 shows the results of three selected

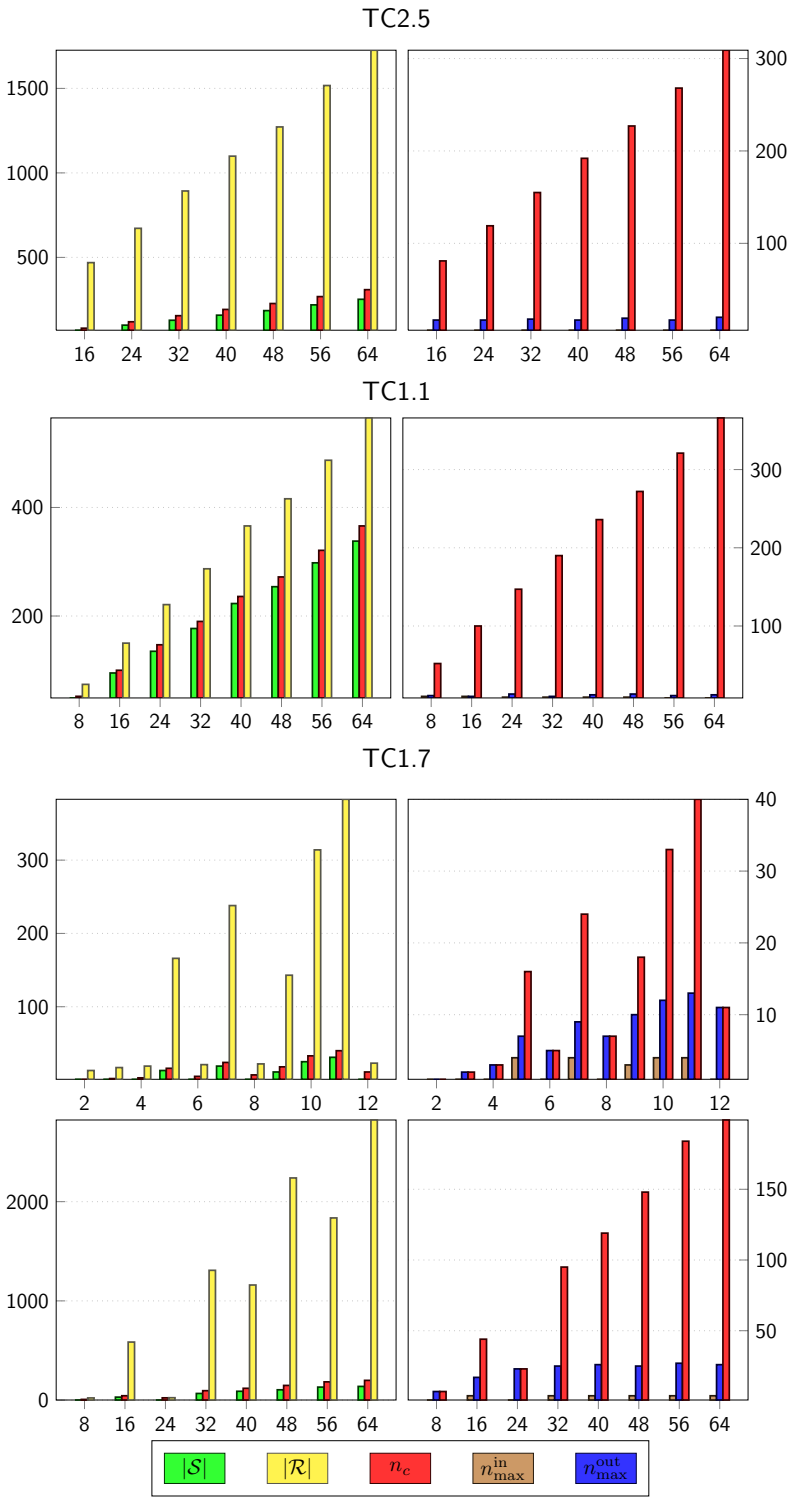


Figure 7.13.: Effect of the PDCR for selected trees in TC1 and TC2 using increasing numbers of processes (x-axes)

trees, which represent three typical cases when distributing the trees in TC1 and in TC2 among an increasing number of processes (x-axes). The results for the remaining trees are supplemented in App. B. The diagrams on the left-hand side in Fig. 7.13 visualize the effect of the PDCR and the ones on the right-hand side indicate how the overall number of communication calls (n_c) are distributed among the participating processes.

First, the results for tree TC2.5 exemplify an optimal exploitation of the PDCR. There is a large discrepancy between the numbers of roots $|\mathcal{R}|$ and senders $|\mathcal{S}|$, and the number of communication calls n_c is close to its lower bound $|\mathcal{S}|$. Moreover, the maximum communication numbers $n_{\max}^{\text{in}}$ and $n_{\max}^{\text{out}}$ are only small fractions of the overall number n_c implying that the communication calls are well-distributed among the participating processes.

The second diagrams in Fig. 7.13 show the results when distributing tree TC1.1 on eight compute nodes. The PDCR is less effective since the numbers $|\mathcal{R}|$ and $|\mathcal{S}|$ are relatively close to each other. This is reasoned by the small branching ($b = 3$) in comparison to the large depth of the tree ($d = 11$). Each part $\mathcal{P}_p$ but $\mathcal{P}_0$ of the resulting distributed tree $\mathcal{D}$ consists of many small subtrees $\mathcal{T}^p \in \mathcal{F}_p$. However, the communication calls are again well-distributed among the participating processes.

Tree TC1.7 demonstrates the distribution of a tree that is flat ($d = 4$) and widespread ($b = 24$). The PDCR is effective but a relative large number of outgoing communications are concentrated on one process. Many successors $S(0)$ of the tree root 0 are on different processes. Hence, the root 0 is involved in many of the outgoing communications.

Furthermore, the results of TC1.7 in Fig. 7.13 show that the number of communication calls is low if the tree branching b coincide with the number of participating processes n_p. The tree is only split at the root 0, the overall number of roots is $|\mathcal{R}| = n_p$. A similar effect also occurs several times in the collections TC1 and TC2 when the number of processes n_p is a multiple of the branching b.

7.2.5. Exploitation of Local Sparsities

The implementation of the portfolio selection problem (7.22) used for the parallel performance results in Sect. 7.2.3 neglects local sparsities (cf. Sect. 3.2.3), i.e. the KKT systems are solved employing the standard implementation of the tree-sparse KKT solution algorithm, which uses dense storage formats for the node subblocks (cf. Sect. 6.4.4). Subsequently, a second implementation exploiting the local sparsity patterns of the node subblocks (7.23) is tested on the test set PT2. First, the state node subblocks G_j of the dynamics (7.22b) are of diagonal form and stored as vectors. Second, the control node subblocks E_j of the dynamics (7.22b) are not stored since they only consist of identities. Third, the node subblock F_j of the range

Table 7.11.: PT2 – Number of nonzero entries in the KKT matrices without and with local sparsity exploitation

			Problem and Tree Sizes			Nnz KKT Matrix		
No.	d	n_a	Nodes	Scenarios	Variables	Node Dense	Node Sparse	Saved
1	12	3	$2.24 \cdot 10^7$	$1.68 \cdot 10^7$	$2.01 \cdot 10^8$	$2.01 \cdot 10^9$	$2.18 \cdot 10^8$	89 %
2	10	4	$1.22 \cdot 10^7$	$9.77 \cdot 10^6$	$1.46 \cdot 10^8$	$1.93 \cdot 10^9$	$1.86 \cdot 10^8$	90 %
3	8	7	$1.92 \cdot 10^7$	$1.68 \cdot 10^7$	$4.03 \cdot 10^8$	$8.59 \cdot 10^9$	$7.21 \cdot 10^8$	92 %
4	7	9	$1.11 \cdot 10^7$	$1.00 \cdot 10^7$	$3.00 \cdot 10^8$	$8.10 \cdot 10^9$	$6.40 \cdot 10^8$	92 %
5	6	12	$5.23 \cdot 10^6$	$4.83 \cdot 10^6$	$1.88 \cdot 10^8$	$6.62 \cdot 10^9$	$4.97 \cdot 10^8$	93 %
6	5	19	$3.37 \cdot 10^6$	$3.20 \cdot 10^6$	$1.92 \cdot 10^8$	$1.04 \cdot 10^{10}$	$7.33 \cdot 10^8$	93 %
7	4	33	$1.38 \cdot 10^6$	$1.34 \cdot 10^6$	$1.36 \cdot 10^8$	$1.26 \cdot 10^{10}$	$8.39 \cdot 10^8$	93 %

Table 7.12.: PT2 – Number of iterations, complete runtimes of the IPM and the computing times per iteration for both implementations of the portfolio selection problem

	Dense Node Blocks			Sparse Node Blocks		
No.	Iterations	Runtime (s)	Time/Iter.	Iterations	Runtime (s)	Time/Iter.
1	13 + 1	90.149	6.439	8 + 1	40.995	3.727
2	14 + 1	77.203	5.147	10 + 1	41.483	3.771
3	10 + 1	149.649	13.604	8 + 1	88.269	9.808
4	9 + 1	124.185	12.419	3 + 1	28.795	7.199
5	7 + 1	74.817	9.352	3 + 1	19.419	4.855
6	8 + 1	123.283	13.698	4 + 1	37.206	7.441
7	8 + 1	170.944	18.994	4 + 1	44.536	8.907

constraints (7.22c) is the same for each node and is stored only once (for each process). This way, the numbers of stored nonzero entries in the KKT matrices are reduced by 89 % to 93 % as shown in Table 7.11.

In the following, the performance of the second implementation with sparse node subblocks is compared to the performance of the standard implementation with dense node subblocks. For this, the tests in PT2 are run using 64 processes and measuring the complete runtime (wall-clock) of the IPM algorithm. This complete runtime also includes the initialization stage requiring approximately the same computing time as one iteration of the algorithm. The performance results for both implementations are presented in Table 7.12. This table lists for each test in PT2 the number of required IPM iterations including the initialization stage (+1) as well as the complete runtime of the IPM and the computing time per IPM iteration. Exploiting local sparsities means requiring less floating point operations and avoiding unnecessary rounding errors for computations involving the KKT matrix. The results in Table 7.12 show that this has great impact on the performance of the algorithm. First, the number of required IPM iterations is significantly less when using the second implementation with sparse node subblocks and, second, the computing time per iteration is reduced by 26 % for PT2.2 to 53 % for PT2.7. Altogether, the complete runtime of the IPM algorithm is reduced by 41 % for PT2.3 to 73 % for PT2.7.

Chapter 8

Conclusions and Outlook

In this thesis, a structure-exploiting and distributed algorithmic framework for tree-structured nonlinear optimization problems is presented. The developed algorithms and concepts are realized in a flexible software framework and their performance is demonstrated by numerical results.

Motivated by stochastic optimization, two formulations of nonlinear tree-sparse problems are introduced that express the dynamic nature of multistage stochastic problems and it is discussed how these problems can be solved efficiently in the generic framework of an interior-point solver. Existing recursive KKT solution algorithms are stated in complete forms for both formulations and then extended to deal with rank-deficiencies and nonconvexities of nonlinear problems. For this, a problem-specific inertia correction strategy is developed enabling local convexifications to avoid refactorizations of the KKT matrix. Also, to deal with problems that do not provide second-order derivatives, a structured quasi-Newton approach is discussed. Tree-sparse Hessian update strategies based on partially separable functions are designed that approximate the tree-sparse Hessians of the Lagrangian in an efficient and structure-preserving way. Numerical results are presented that demonstrate the performance of the developed algorithms. Only in the example of the magnetic levitation vehicle the local convexifications of the inertia correction are successfully used. However, the tree-sparse Hessian update strategies combined with standard convexifications are successfully applied to solve tree-sparse problems in the robust model predictive control of a perturbed nonlinear bioreactor modeled by ordinary differential equations with stochastic disturbances. Moreover, in the control of the nonlinear double integrator, the structured quasi-Newton approach turns out to be a competitive alternative even if explicit second-order evaluations are available.

Also, this thesis presents a complete concept of distribution to solve the tree-sparse problems in parallel on distributed memory systems. Maintaining a consistent node-wise presentation of problems and algorithms, a distributed programming model is developed that is built on a static

depth-first distribution of the tree nodes. After introducing the concept of depth-first distributed trees, theoretical results are shown that are then used to develop prototypes for distributed tree algorithms with few idle times and communication overhead. Distributed versions of the tree-sparse KKT algorithms are provided and it is demonstrated how the theoretical results of the distributed trees can be used to save communication in the parallel performance of the algorithms using the post-distribution communication reduction. It is discussed how the distribution of the tree-sparse problems fits into the generic framework of an interior-point method and, finally, the programming model of a completely distributed interior-point solver is presented. Very good performance results that are comparable with others reported in the literature not only confirm the practicability of the concept of distribution but also show the efficiency of this approach. Moreover, numerical results are presented showing that applying the post-distribution communication reduction to trees in the context of the portfolio optimization problems is efficient and saves a lot of communication.

The infrastructure of the Tree-Sparse Library together with the modular design of the interior-point solver Clean::IPM provide a flexible and efficient solution framework for tree-sparse problems with many algorithmic options and configurations, and it is believed that it could be of great use to solve tree-structured optimization problems. Thus, for future directions this framework should prove its value by using it in further applications. For example, it is thinkable to solve huge-scale nonlinear portfolio optimization problems including skewness or logarithmic utility functions. In robust model predictive control, it is possible to regulate further dynamic processes such as the Chylla-Haase benchmark reactor including larger dynamic models and more uncertainties. Investigating further examples may also help to improve the solution approaches and, additionally, may provide more conclusions about the practicability of local convexifications in the proposed inertia correction strategy.

The realization of further applications could be motivated by facilitating the implementation of new optimization problems. Supporting the formulation of tree-sparse problems by a specifically designed modeling language, cumbersome chores of implementation could be automated using code generating techniques. Since already incorporated into the design of the Tree-Sparse Library, code generation could also be used to improve computational efficiencies and memory requirements of the solution algorithms by writing problem-specific source code exploiting local sparsities. Finally, extensions and modifications of the algorithmic approaches are also thinkable. Alternatively to the structured quasi-Newton approach, algorithmic differentiation could be used to generate first- and second-order derivatives. Also, the performance of the interior-point method could be improved by incorporating problem-tailored initial value strategies, e.g. warmstart strategies for tree-structured problems that operate on a reduced tree.

Bibliography

[1] OOPS – Object Oriented Parallel Solver. `http://www.maths.ed.ac.uk/~gondzio/parallel/solver.html`, 2003-2016.

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. `http://www.netlib.org/lapack`.

[4] J. Asprion, O. Chinellato, and L. Guzzella. Efficient solution of the diesel-engine optimal control problem by time-domain decomposition. *Control Engineering Practice*, 30(0):34 – 44, 2014.

[5] G. Bader and P. Deuflhard. A semi-implicit midpoint rule for stiff systems of ordinary differential equations. *Numer. Math.*, 41:373–398, 1983.

[6] J. J. Barton and L. R. Nackman. *Scientific and Engineering*. 1994.

[7] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numer. Math.*, 4:238–252, 1962.

[8] J. R. Birge and F. Louveaux. *Introduction to Stochastic Programming*. Springer, Berlin, 1997.

[9] J. Blomvall. *Optimization of Financial Decisions Using a New Stochastic Programming Method*. PhD thesis, Linköpings universitet, Linköping, Sweden, 2001.

[10] J. Blomvall. A multistage stochastic programming algorithm suitable for parallel computing. *Parallel Computing*, 29(4):431 – 445, 2003. Parallel computing in numerical optimization.

[11] J. Blomvall and P. O. Lindberg. A riccati-based primal interior point solver for multistage stochastic programming. *European J. Oper. Res.*, 143(2):452–461, 2002.

[12] J. Blomvall and P. O. Lindberg. A riccati-based primal interior point solver for multistage stochastic programming-extensions. *Optimization methods and software*, 17(3):383–407, 2002.

[13] J. Blomvall and P. O. Lindberg. Back-testing the performance of an actively managed option portfolio at the Swedish stock market, 1990–1999. *Journal of Economic Dynamics and Control*, 27(6):1099–1112, 2003.

[14] H. G. Bock. Recent advances in parameteridentification techniques for O.D.E. In P. Deuflhard and E. Hairer, editors, *Numerical Treatment of Inverse Problems in Differential and Integral Equations*, volume 2 of *Progress in Scientific Computing*, pages 102–125. Birkhäuser Verlag, Basel, Switzerland, 1983.

[15] Boost. Generic Programming Techniques. `http://www.boost.org/community/generic_programming.html`.

[16] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, 2004.

[17] R. H. Byrd, J. Nocedal, and R. A. Waltz. KNITRO: An integrated package for nonlinear optimization. In *Large Scale Nonlinear Optimization, 35–59, 2006*, pages 35–59. Springer Verlag, 2006.

[18] J. O. Coplien. Curiously Recurring Template Patterns. *C++ Report*, 7(2):24–27, 1995.

[19] T. B. Culver and C. A. Shoemaker. Optimal control for groundwater remediation by differential dynamic programming with quasi-Newton approximations. *Water Resources Research*, 29(4):823–831, 1993.

[20] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations research*, 8(1):101–111, 1960.

[21] J. E. Dennis and J. J. Moré. Quasi-Newton methods, motivation and theory. *SIAM Rev.*, 19(1):46–89, 1977.

[22] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, third edition, 2005.

[23] I. S. Duff. MA28—a set of FORTRAN subroutines for sparse unsymmetric linear equations. Report AERE R8730, HMSO, London, 1977.

[24] R. Fletcher. An optimal positive definite update for sparse Hessian matrices. *SIAM Journal on Optimization*, 5(1):192–218, 1995.

[25] R. Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[26] A. Forsgren, P. E. Gill, and M. H. Wright. Interior methods for nonlinear optimization. *SIAM Review*, 44(4):525–597, 2002.

[27] K. Frauendorfer. The stochastic programming extension of the Markowitz approach. *Int. J. Neural Mass-Parallel Comput. Inform. Syst.*, 5(4):449–460, 1995.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[29] I. Garrido Hernández and M. C. Steinbach. A multistage stochastic programming approach in real-time process control. In Grötschel et al. [42], pages 479–498.

[30] E. M. Gertz, P. E. Gill, and J. Muethering. Users guide for SnadiOpt: a package adding automatic differentiation to SNOPT. Technical Memorandum ANL/MCS-TM-245, Argonne National Labs, Jan. 2001.

[31] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. Sparse matrix methods in optimization. *SIAM Journal on Scientific and Statistical Computing*, 5(3):562–589, 1984.

[32] P. E. Gill, W. Murray, and M. S. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM J. Optim.*, 12(4):979–1006, 2002.

[33] J. Gondzio and A. Grothey. Solving nonlinear portfolio optimization problems with the primal-dual interior point method. Technical Report MS-04-001, Department of Mathematics and Statistics, University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ, UK, Mar.31 2004.

[34] J. Gondzio and A. Grothey. Direct solution of linear systems of size 10^9 arising in optimization with interior point methods. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Waśniewski, editors, *Parallel Processing and Applied Mathematics*, volume 3911 of *Lecture Notes in Computer Science*, pages 513–525. Springer Berlin Heidelberg, 2006.

[35] J. Gondzio and A. Grothey. Parallel interior-point solver for structured quadratic programs: Application to financial planning problems. *Annals of Operations Research*, 152(1):319–339, 2007.

[36] J. Gondzio and A. Grothey. Solving non-linear portfolio optimization problems with the primal-dual interior point method. *European Journal Operational Research*, 181(3):1019–1029, 2007.

[37] A. Griewank and P. L. Toint. On the unconstrained optimization of partially separable functions. *Nonlinear Optimization*, 1982:247–265, 1981.

[38] A. Griewank and P. L. Toint. Local convergence analysis for partitioned quasi-Newton updates. *Numerische Mathematik*, 39(3):429–448, 1982.

[39] A. Griewank and P. L. Toint. Partitioned variable metric updates for large structured optimization problems. *Numerische Mathematik*, 39(1):119–137, 1982.

[40] A. Griewank and P. L. Toint. Numerical experiments with partially separable optimization problems. In *Numerical analysis*, pages 203–220. Springer, 1984.

[41] A. Grothey. Financial applications: Parallel portfolio optimization. In *Parallel Computing*, pages 435–469. Springer, 2009.

[42] M. Grötschel, S. O. Krumke, and J. Rambau, editors. *Online Optimization of Large Scale Systems*. Springer, Berlin, 2001.

[43] L. Grüne and J. Pannek. *Nonlinear model predictive control*. Springer, 2011.

[44] R. Henrion, P. Li, A. Möller, M. C. Steinbach, M. Wendt, and G. Wozny. Stochastic optimization for operating chemical processes under uncertainty. In Grötschel et al. [42], pages 457–478.

[45] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Siam, 1996.

[46] N. Hofmann. Parallele KKT-Löser für stochastische Optimierungsprobleme. Diploma thesis, Leibniz Universität Hannover, 2009.

[47] The HSL Mathematical Software Library. *HSL MA27 Package Specification*, Mar. 2003.

[48] J. Hübner, M. Schmidt, and M. C. Steinbach. A Distributed Interior-Point KKT Solver for Multistage Stochastic Problems. In preparation, 2015.

[49] A. Huţanu. Code generator for sparse linear algebra in stochastic optimization. Diploma thesis, "Politecnica" University of Bucharest, 2002.

[50] J. Hübner. Parallelisierung rekursiver Algorithmen zur Optimierung mit Baumtopologie. Diploma thesis, Leibniz Universität Hannover, 2010.

[51] R. B. S. John E. Dennis. *Numerical methods for unconstrained optimization and nonlinear equations*, volume 16. Siam, 1996.

[52] O. C. Jonas Asprion and L. Guzzella. Partitioned quasi-Newton approximation for direct collocation methods and its application to the fuel-optimal control of a Diesel engine. *Journal of Applied Mathematics*, 2014, 2014.

[53] R. Jordan. *The Eye of the World*, volume 1 of *Wheel of Time*. 1990.

[54] H. S. Karl Frauendorfer. Mean-variance analysis in a multiperiod setting. 1997.

[55] C. Lawson, R. Hanson, D. Kincaid, and F. Krough. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5:308–325, 1979.

[56] M. Lazar, D. M. De La Peña, W. Heemels, and T. Alamo. On input-to-state stability of min–max nonlinear model predictive control. *Systems & Control Letters*, 57(1):39–48, 2008.

[57] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Program.*, 45(3):503–528, 1989.

[58] A. Lucia. An explicit quasi-Newton update for sparse optimization calculations. *MATHEMATICS of computation*, 40(161):317–322, 1983.

[59] S. Lucia and S. Engell. Multi-stage and two-stage robust nonlinear model predictive control. In *Nonlinear Model Predictive Control*, volume 4, pages 181–186, 2012.

[60] S. Lucia, T. Finkler, D. Basak, and S. Engell. A new robust NMPC scheme and its application to a semi-batch reactor example. In *Proc. of the International Symposium on Advanced Control of Chemical Processes*, pages 69–74, 2012.

[61] J. Macki and A. Strauss. *Introduction to optimal control theory*. Springer, 1982.

[62] V. Malmedy and P. L. Toint. Approximating hessians in unconstrained optimization arising from discretized problems. *Computational Optimization and Applications*, 50(1):1–22, 2011.

[63] H. M. Markowitz. Portfolio selection. *J. Finance*, 7(1):77–91, 1952.

[64] H. M. Markowitz. The utility of wealth. *J. Polit. Econ.*, 60(2):151–158, 1952.

[65] S. Mehrotra. On the implementation of a primal-dual interior point method. *SIAM J. Optim.*, 2(4):575–601, 1992.

[66] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, Berlin, 2nd edition, 2006.

[67] E. Polak. *Optimization: Algorithms and Consistent Approximations*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[68] M. Powell and P. L. Toint. The Shanno-Toint procedure for updating sparse symmetric matrices. *IMA Journal of Numerical Analysis*, 1(4):403–413, 1981.

[69] M. J. D. Powell. The convergence of variable metric methods for nonlinearly constrained optimization calculations. In O. L. Mangasarian, R. R. Meyer, and S. M. Robinson, editors, *Nonlinear Programming 3*, pages 27–61. Academic Press, 1978.

[70] T. Rauber and G. Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business, 2013.

[71] J.-C. G. Richard H.Byrd and J. Nocedal. A trust region method based on interior point techniques for nonlinear programming. *Mathematical Programming*, 89(1):149–185, 2000.

[72] M. Schmidt. *A Generic Interior-Point Framework for Nonsmooth and Complementarity Constrained Nonlinear Optimization*. PhD thesis, Gottfried Wilhelm Leibniz Universität Hannover, 2013.

[73] M. Schmidt. An Interior-Point Method for Nonlinear Optimization Problems with Locatable and Separable Nonsmoothness. Technical report, Friedrich-Alexander-Universität Erlangen-Nürnberg, Department Mathematik, April 2014. `http://www.optimization-online.org/DB_HTML/2014/04/4330.html`.

[74] W. F. Sharpe. *Portfolio Theory and Capital Market*. McGraw-Hill, New York, 1970.

[75] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. C++ In-Depth Series. Addison-Wesley, 2001.

[76] M. C. Steinbach. Numerische Berechnung optimaler Steuerungen für Industrieroboter. Diploma thesis, Universität Bonn, 1987.

[77] M. C. Steinbach. *Fast Recursive SQP Methods for Large-Scale Optimal Control Problems*. Ph. D. dissertation, Universität Heidelberg, 1995.

[78] M. C. Steinbach. Recursive direct algorithms for multistage stochastic programs in financial engineering. ZIB Preprint SC-98-23, Zuse Institute Berlin, 1998.

[79] M. C. Steinbach. Recursive direct optimization and successive refinement in multistage stochastic programs. ZIB Preprint SC-98-27, Zuse Institute Berlin, 1998.

[80] M. C. Steinbach. Hierarchical sparsity in multistage convex stochastic programs. ZIB Report ZR-00-15, Zuse Institute Berlin, May 2000. Appeared in [101].

[81] M. C. Steinbach. Recursive direct optimization and successive refinement in multistage stochastic programs. *Ann. Oper. Res.*, 2000. Accepted for publication; LATER REJECTED!!???.

[82] M. C. Steinbach. Markowitz revisited: Mean-variance models in financial portfolio analysis. *SIAM Rev.*, 43(1):31–85, 2001.

[83] M. C. Steinbach. Tree-sparse convex programs. *Math. Methods Oper. Res.*, 56(3):347–376, 2002.

[84] M. C. Steinbach. Robust process control by dynamic stochastic programming. *Proc. Appl. Math. Mech.*, 4(1):11–14, 2004.

[85] M. C. Steinbach. Optimierung bei Differentialgleichungen. Lecture Notes, 2009. Hannover.

[86] M. C. Steinbach, H. G. Bock, G. V. Kostin, and R. W. Longman. Mathematical optimization in robotics: Towards automated high speed motion planning. *Surv. Math. Ind.*, 7(4):303–340, 1998.

[87] M. C. Steinbach, H. G. Bock, and R. W. Longman. Time optimal control of SCARA robots. In *Proc. 1990 AIAA Guid., Nav., and Control Conf.*, pages 707–716, Portland, OR.

[88] M. C. Steinbach, H. G. Bock, and R. W. Longman. Time-optimal extension and retraction of robots: Numerical analysis of the switching structure. *J. Optim. Theory Appl.*, 84(3):589–616, 1995.

[89] M. C. Steinbach and H.-J. Vollbrecht. Efficient stochastic programming techniques for electricity swing options. In J. Kallrath, P. M. Pardalos, S. Rebennack, and M. Scheidt, editors, *Optimization in the Energy Industry*, chapter 21, pages 485–506. Springer, Berlin, 2009.

[90] A. Stepanov and M. Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1994.

[91] P. L. Toint. A note about sparsity exploiting quasi-Newton updates. *Mathematical Programming*, 21(1):172–181, 1981.

[92] M. Ulbrich, S. Ulbrich, and L. N. Vicente. A globally convergent primal-dual interior-point filter method for nonlinear programming. *Mathematical Programming*, 100(2):379–410, 2004.

[93] L. H. Ungar. A bioreactor benchmark for adaptive network-based process control. *Neural Networks for Control*, page 387, 1995.

[94] Boost C++ Libraries. `http://www.boost.org/`, 1998-2016.

[95] ISO C++ Standard. `https://isocpp.org/std`.

[96] MPI Standard. `http://www.mpi-forum.org/`.

[97] Open MPI: Open Source High Performance Computing. `http://www.open-mpi.org/`, 2004-2016.

[98] Parallel Boost Graph Library. `http://www.boost.org/doc/libs/1_57_0/libs/graph_parallel/doc/html/index.html`.

[99] `http://en.wikipedia.org/wiki/Standard_Template_Library`.

[100] Unified Modeling Language. `www.uml.org`.

[101] S. P. Uryasev and P. M. Pardalos, editors. *Stochastic Optimization: Algorithms and Applications*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.

[102] R. J. Vanderbei. *LOQO User's Manual – Version 4.05*. Princeton University, School of Engineering and Applied Science, Department of Operations Research and Financial Engineering, Princeton, New Jersey, Sept. 2006.

[103] R. J. Vanderbei and D. F. Shanno. An interior-point algorithm for nonconvex nonlinear programming. *Comput. Optim. Appl.*, 13:231–252, 1997.

[104] A. Wächter and L. T. Biegler. Line search filter methods for nonlinear programming: Motivation and global convergence. *SIAM J. on Optimization*, 16(1):1–31, May 2005.

[105] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57, 2006.

[106] A. Wächter and C. Laird. *Introduction to IPOPT: A Tutorial for Downloading, Installing and Using IPOPT*, Nov. 2006. Revision 799.

[107] S. J. Wright. *Primal-Dual Interior-Point Methods*. SIAM, Philadelphia, PA, 1997.

[108] S. A. Zenios. *Financial Optimization.* Cambridge University Press, Cambridge, UK, 1992.

[109] S. A. Zenios. High-performance computing in finance: The last 10 years and the next. *Parallel Computing*, 25(13):2149–2175, 1999.

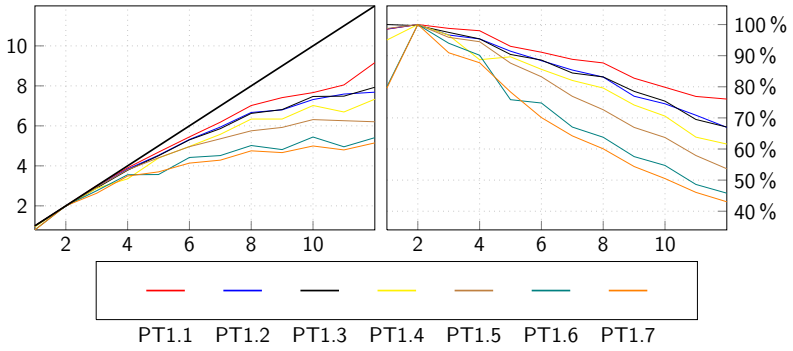
[110] W. T. Ziemba and J. M. Mulvey, editors. *Worldwide Asset and Liability Modeling.* Cambridge University Press, Cambridge, UK, 1998.

Chapter A

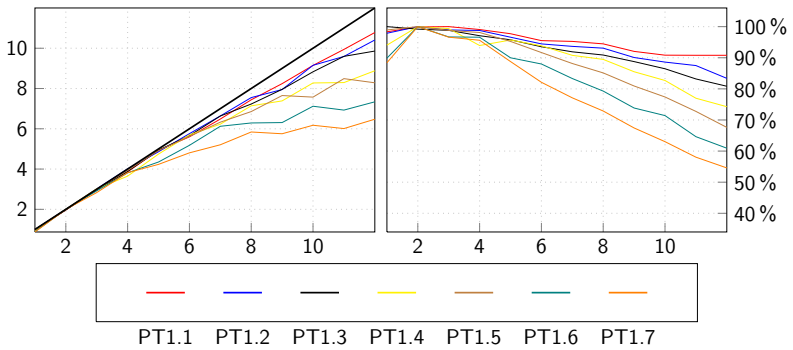
Parallel Performance Results

A.1. Supplemented Results for PT1 and PT2 Using Mean-Preserving Approximations

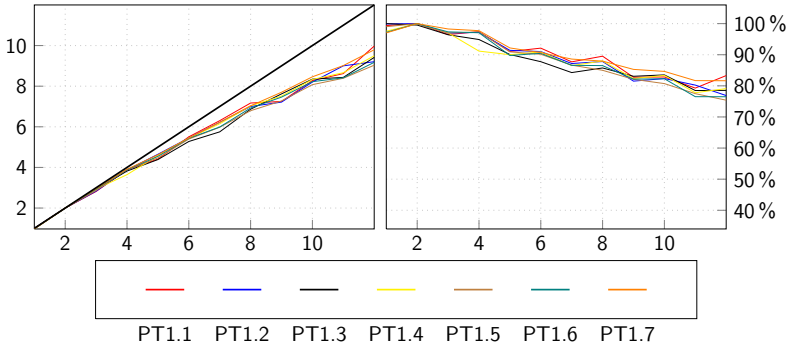
The following graphics supplements the parallel performance results in Sect. 7.2.3 for the portfolio test collections PT1 and PT2 using only mean-preserving approximations of the random variables.



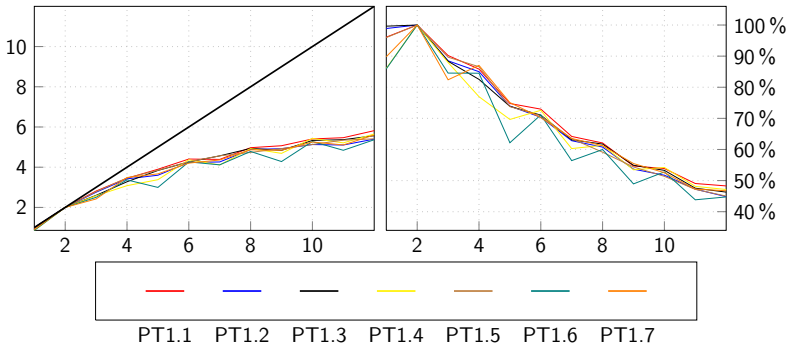
PT1 on one compute node — Parallel performance of the matrix-vector product



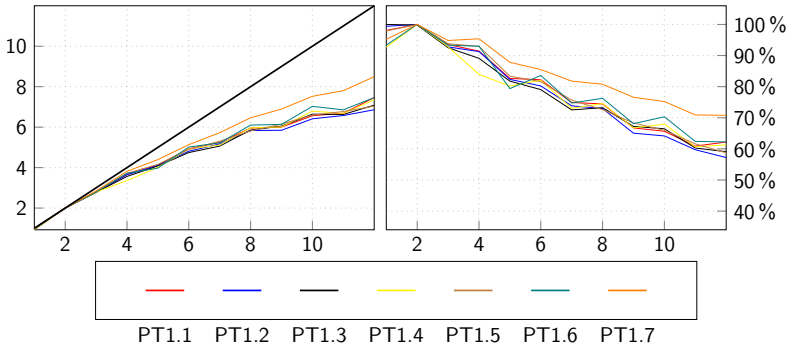
PT1 on one compute node – Parallel performance of the outward substitution



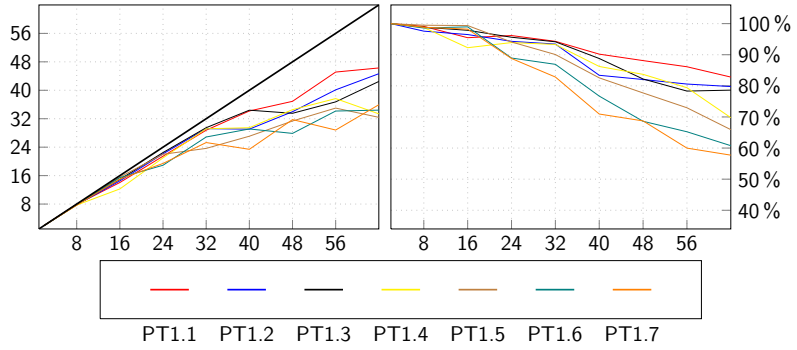
PT1 on one compute node – Parallel performance of the KKT solver



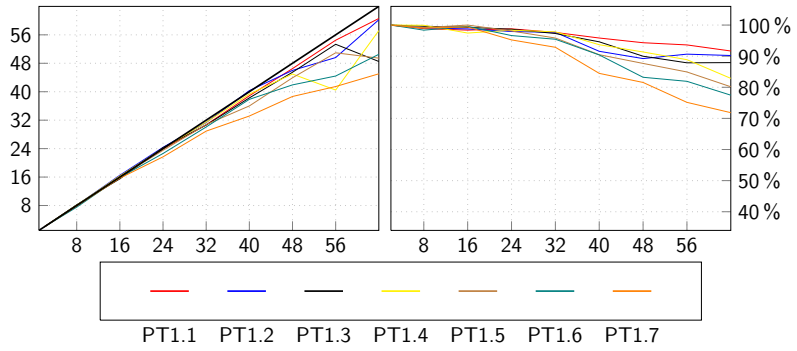
PT1 on one compute node – Parallel performance of the IPM iteration



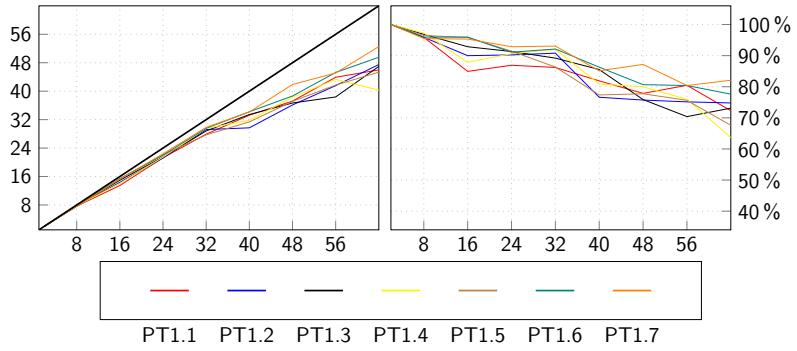
PT1 on one compute node – Parallel performance of the complete IPM algorithm



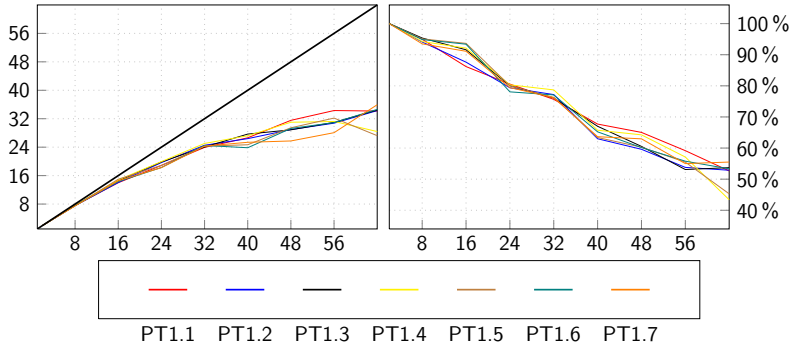
PT1 on eight compute nodes – Parallel performance of the matrix-vector product



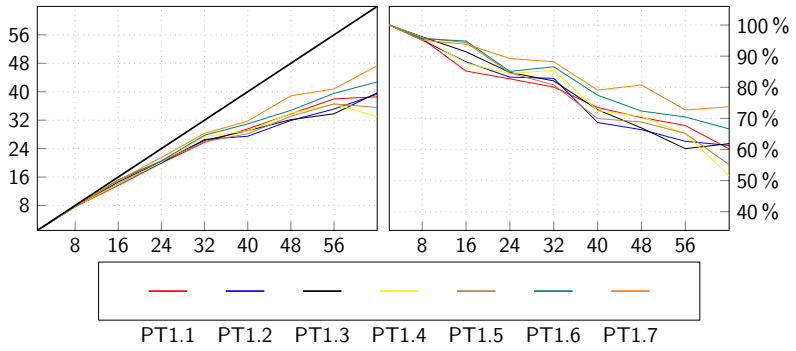
PT1 on eight compute nodes – Parallel performance of the outward substitution



PT1 on eight compute nodes – Parallel performance of the KKT solver



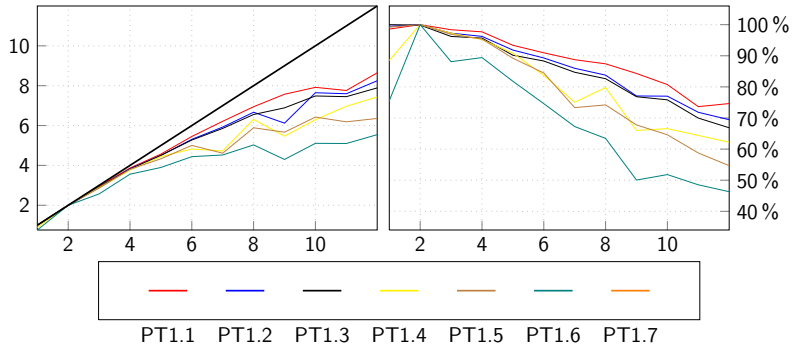
PT1 on eight compute nodes – Parallel performance of the IPM iteration



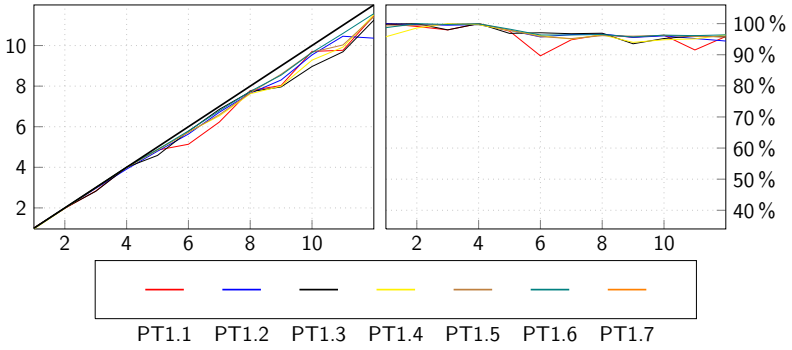
PT1 on eight compute nodes – Parallel performance of the complete IPM algorithm

A.2. Results for PT1 Using Second-Moment-Conforming Approximations

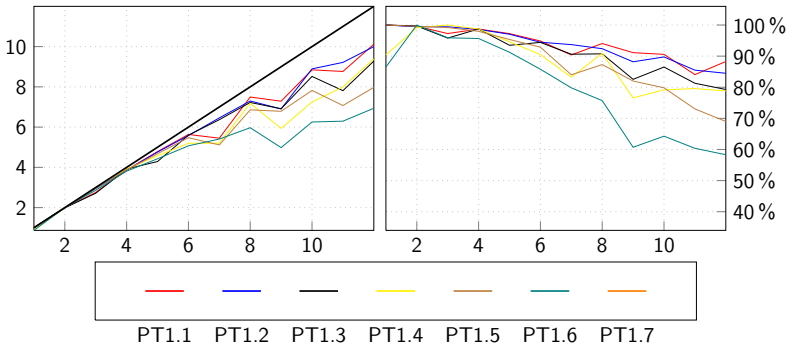
The following graphics demonstrate the parallel performance results for some of the test cases in the portfolio test collection PT1 (see Table 7.5) using approximations of the random variables that satisfy both conditions for the first and second moments (7.21), respectively.



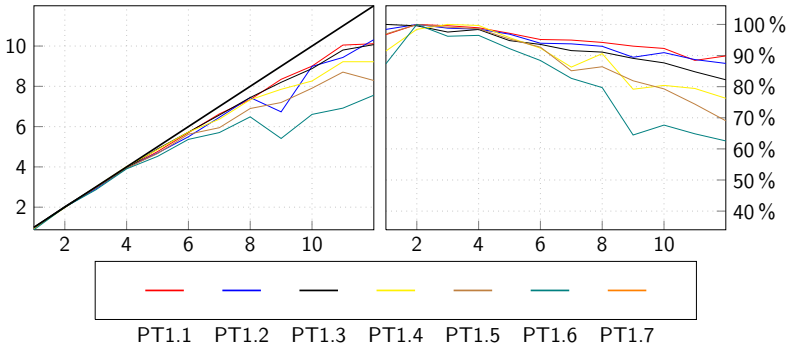
PT1 – Parallel performance of the matrix-vector product



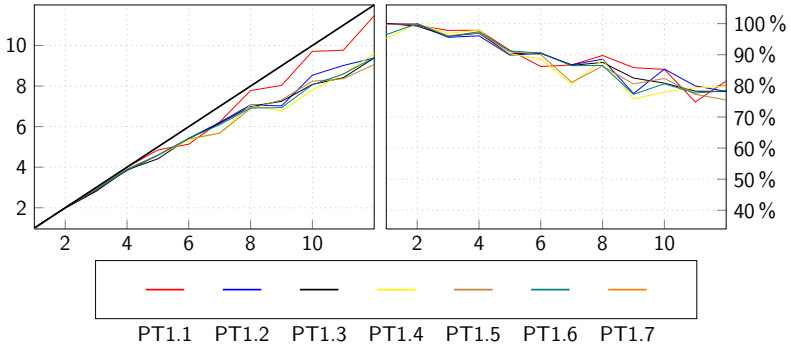
PT1 – Parallel performance of the factorization



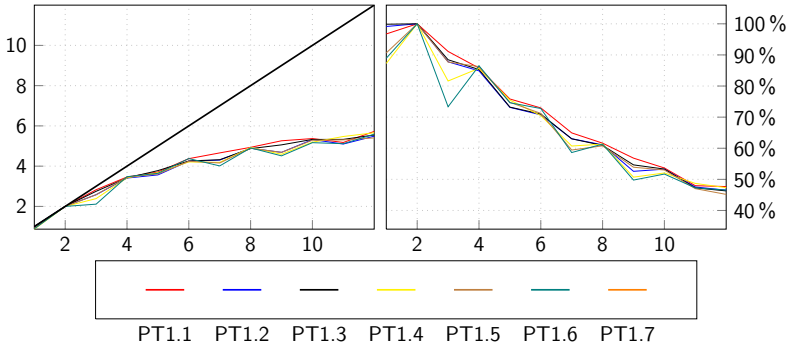
PT1 – Parallel performance of the inward substitution



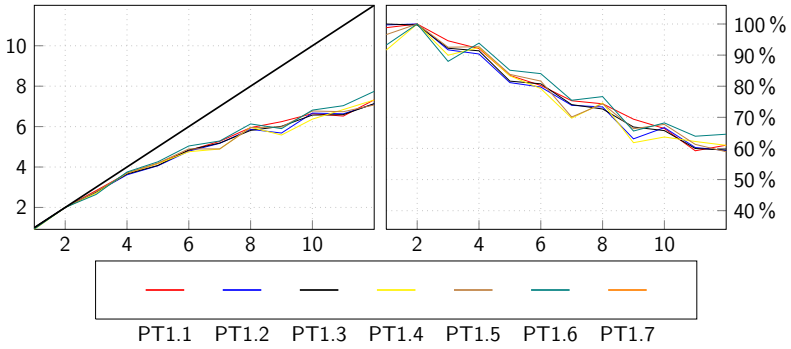
PT1 – Parallel performance of the outward substitution



PT1 – Parallel performance of the complete KKT solver



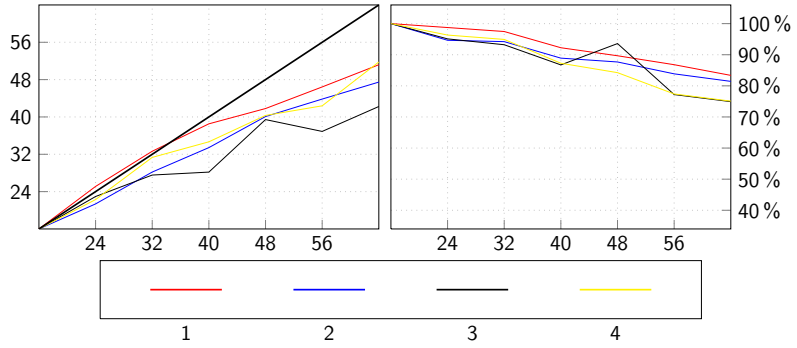
PT1 – Parallel performance of iterative IPM stage



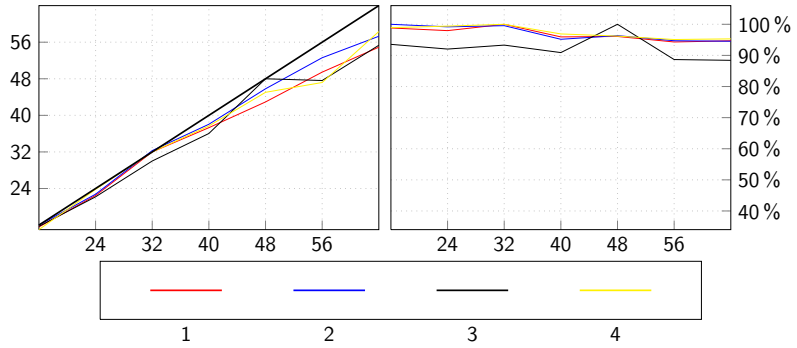
PT1 – Parallel performance of the complete IPM algorithm

A.3. Results for PT2 Using Second-Moment-Conforming Approximations

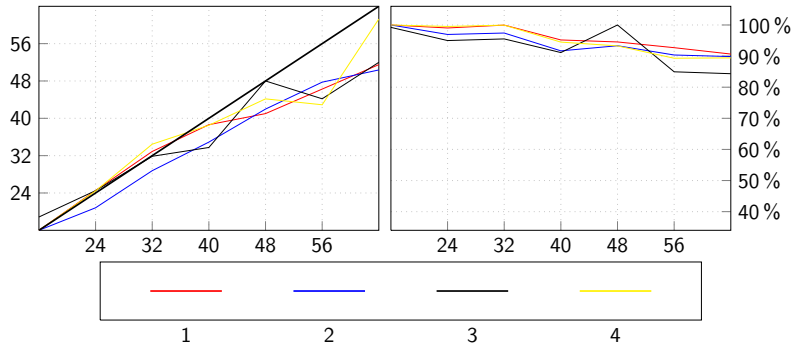
The following graphics demonstrate the parallel performance results for some of the test cases in the portfolio test collection PT2 (see Table 7.5) using approximations of the random variables that satisfy both conditions for the first and second moments (7.21), respectively.



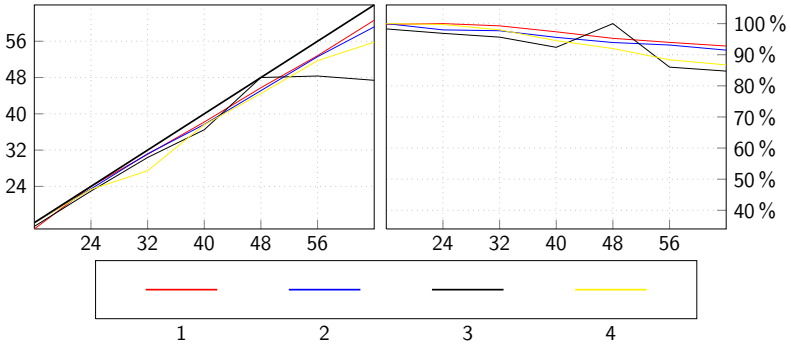
PT2 – Parallel performance of the matrix-vector product



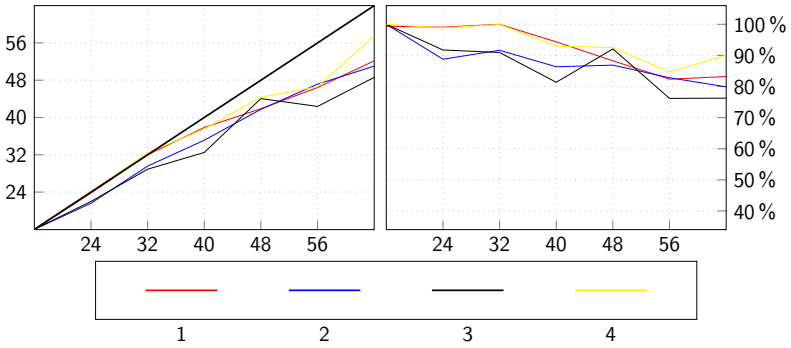
PT2 – Parallel performance of the factorization



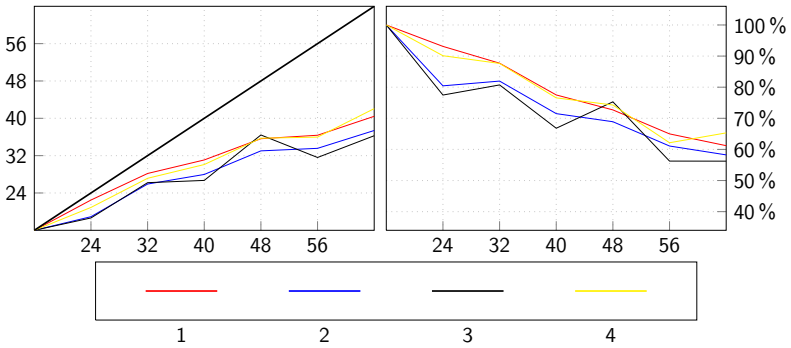
PT2 – Parallel performance of the inward substitution



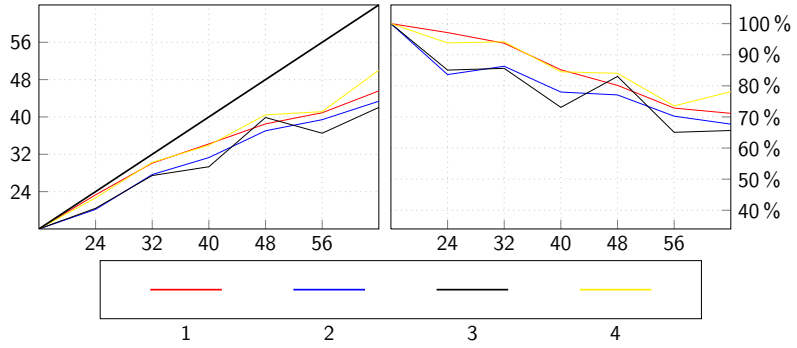
PT2 – Parallel performance of the outward substitution



PT2 – Parallel performance of the complete KKT solver



PT2 – Parallel performance of iterative IPM stage



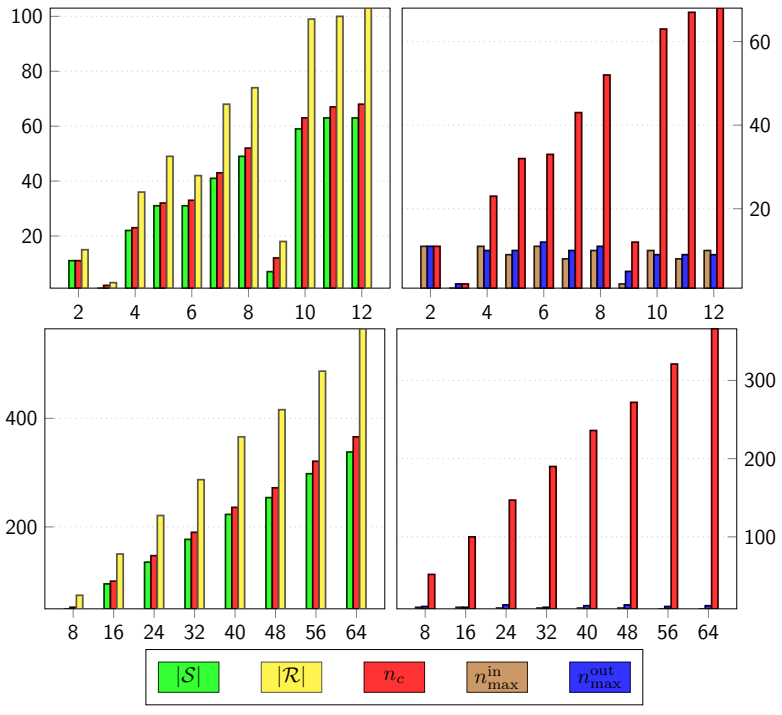
PT2 – Parallel performance of the complete IPM algorithm

Chapter B

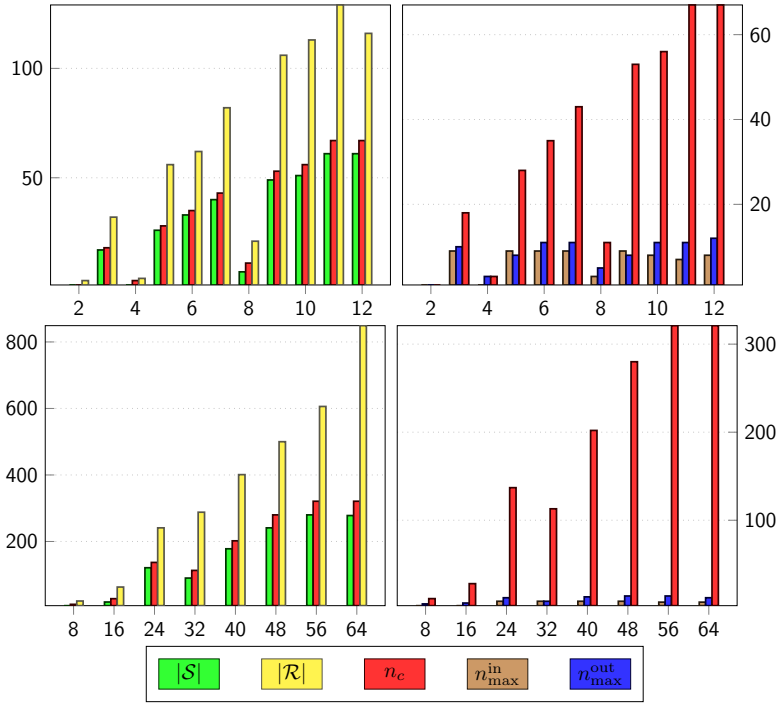
Results of the Post-Distribution Communication Reduction

B.1. Effect of the PDCR for Tree Collection 1

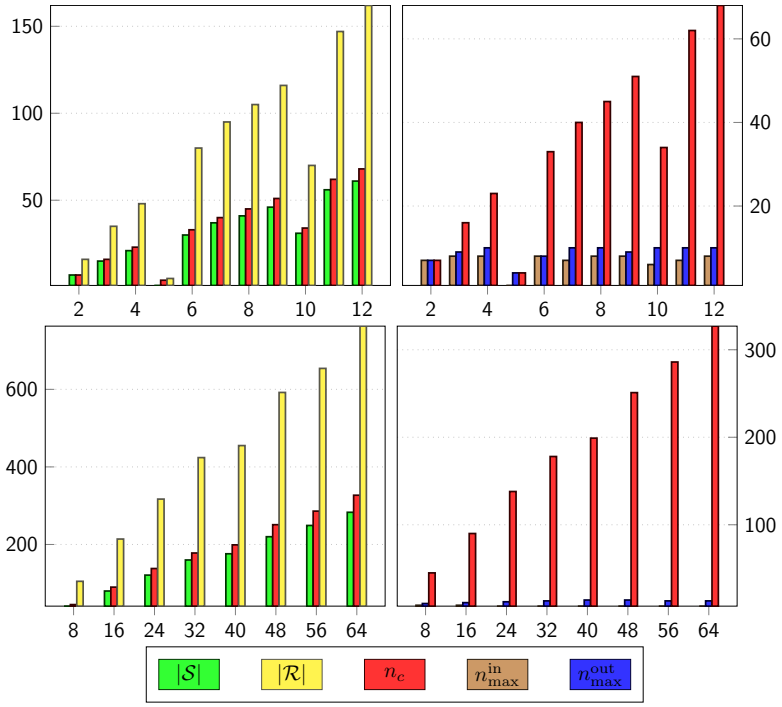
The following graphics show the effect of the PDCR for the remaining trees in the tree collection TC1 listed in Table 7.10. Those trees coincide with the trees in the portfolio test collection PT1 (see Table 7.5).



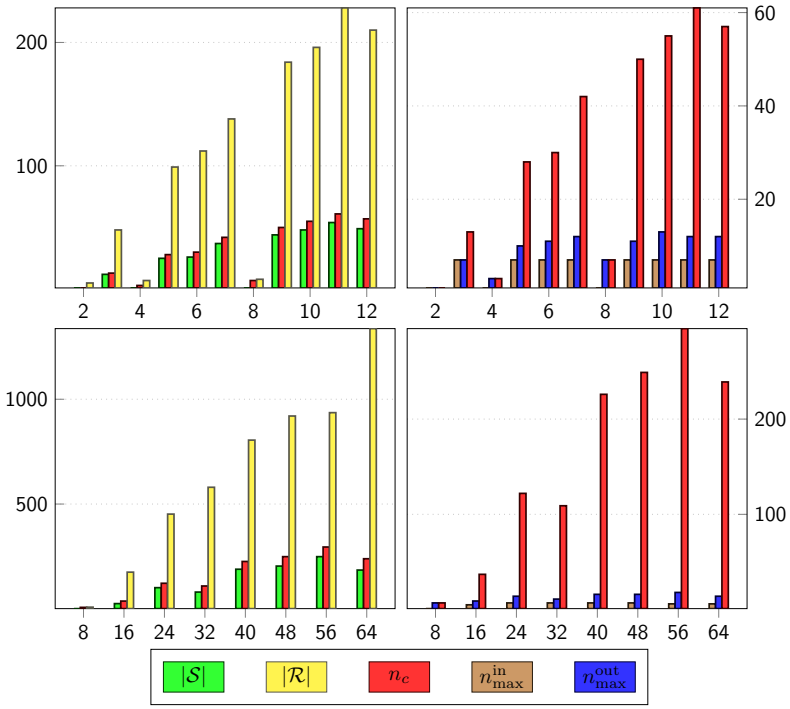
TC1.1 – Effect of the PDCR



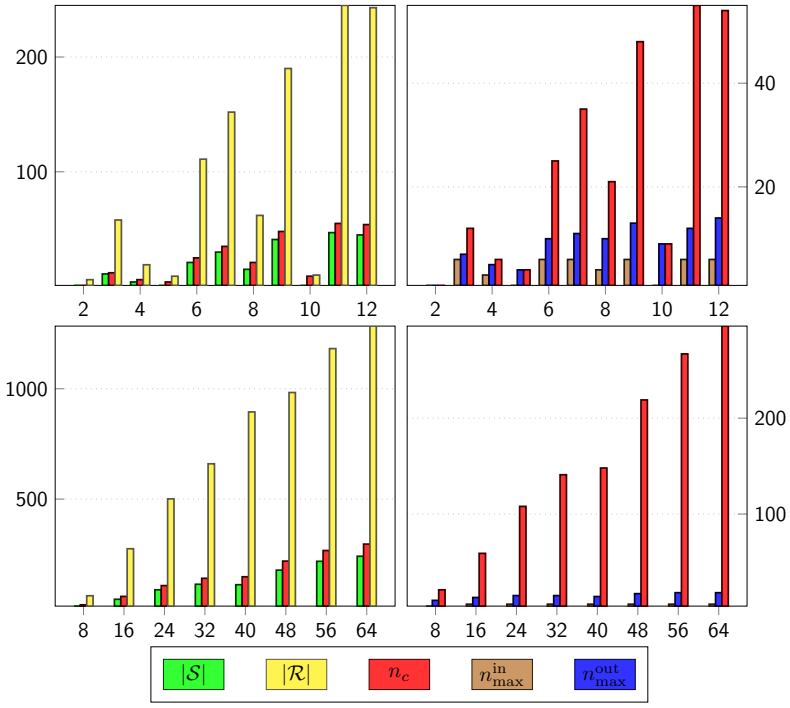
TC1.2 – Effect of the PDCR



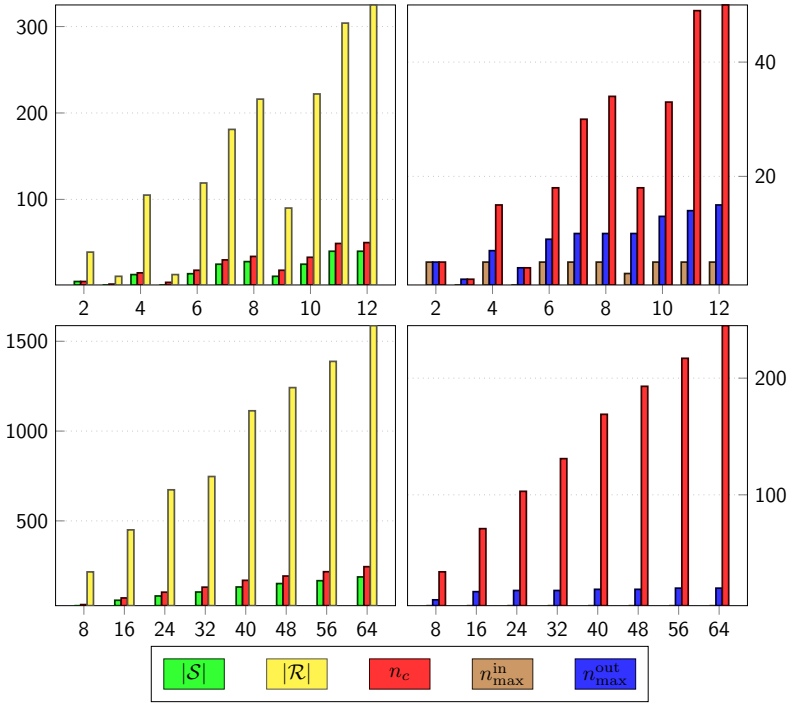
TC1.3 – Effect of the PDCR



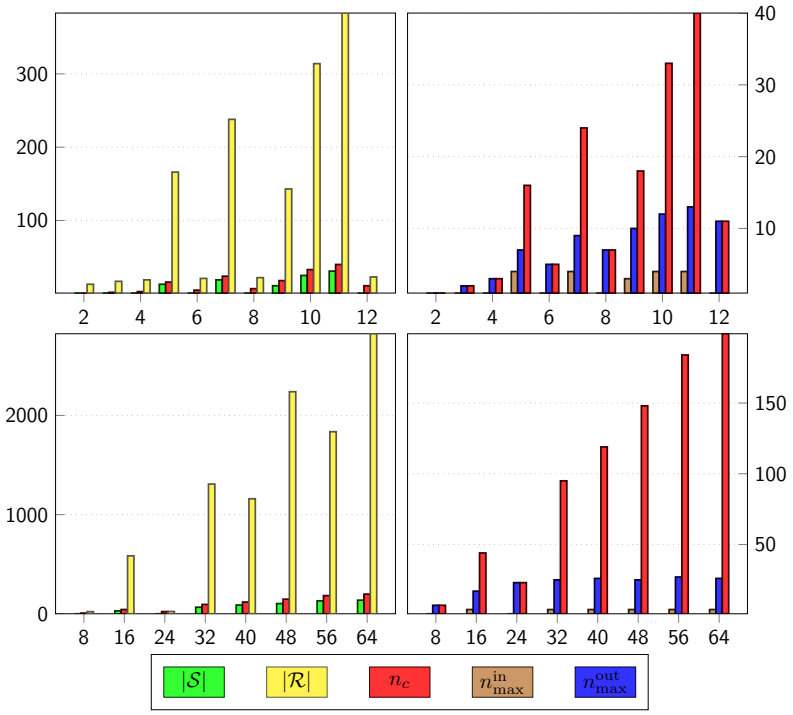
TC1.4 – Effect of the PDCR



TC1.5 – Effect of the PDCR



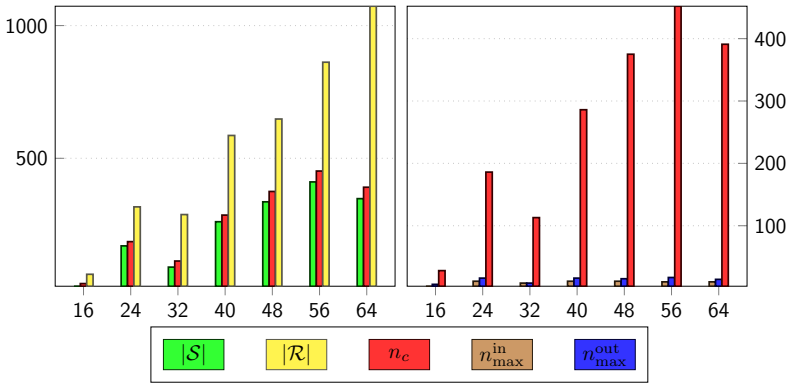
TC1.6 – Effect of the PDCR



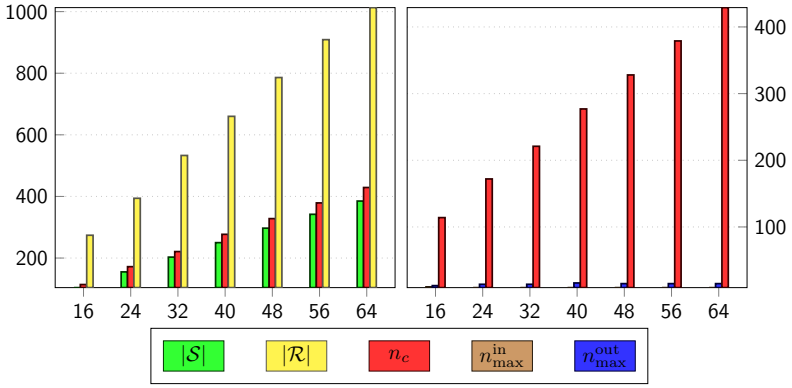
TC1.7 – Effect of the PDCR

B.2. Effect of the PDCR for Tree Collection 2

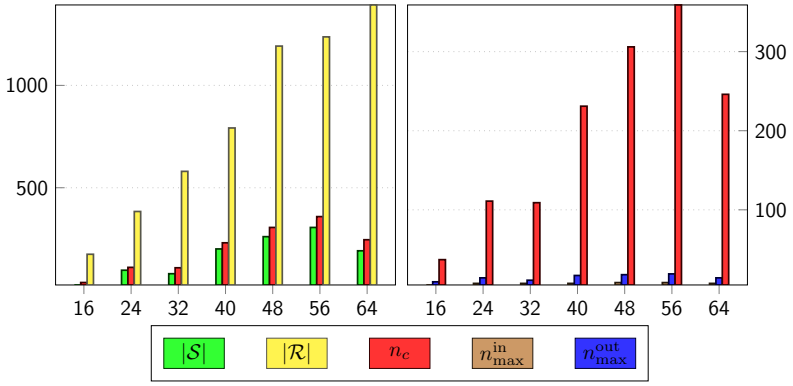
The following graphics show the effect of the PDCR of the remaining trees in the tree collection TC2 listed in Table 7.10. Those trees coincide with the trees in the portfolio test collection PT2 (see Table 7.5).



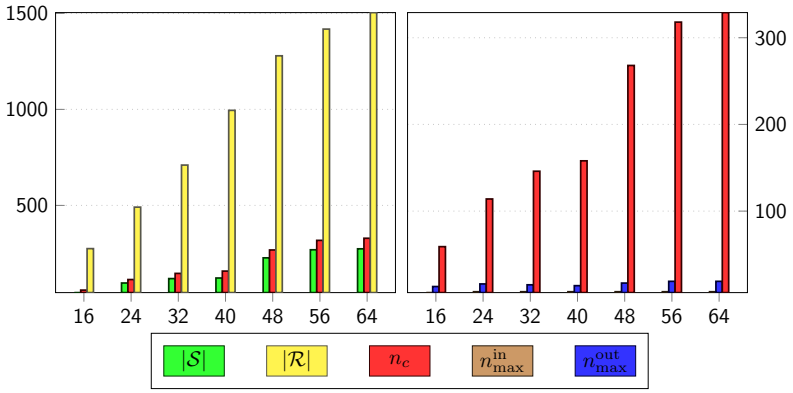
TC2.1 – Effect of the PDCR



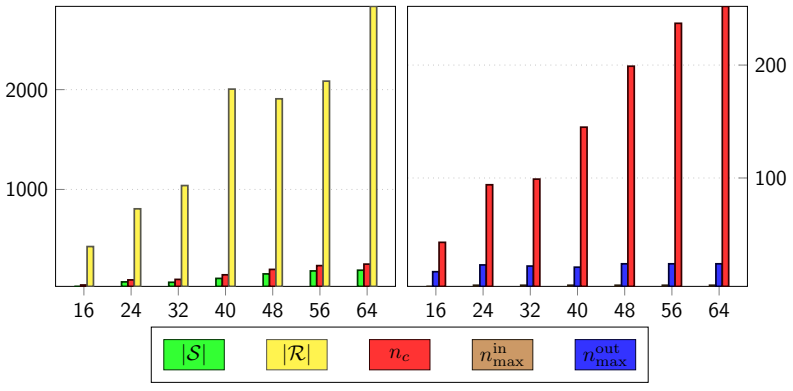
TC2.2 – Effect of the PDCR



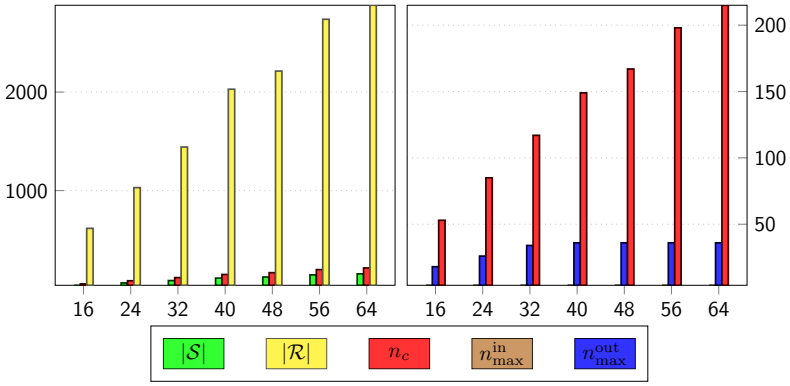
TC2.3 – Effect of the PDCR



TC2.4 – Effect of the PDCR



TC2.6 – Effect of the PDCR



TC2.7 – Effect of the PDCR

Lebenslauf – Dipl.-Math. Jens Hübner

Angaben zur Person

Geburtsdatum:	23.10.1984
Geburtsort:	Bremen
Adresse:	Steigertahlstraße 2, 30451 Hannover
E-Mail:	huebner@ifam.uni-hannover.de

Bildungsweg

seit 09/2010	Promotionsstudent an der Leibniz Universität Hannover
09/2010	Diplom im Studiengang Mathematik an der Leibniz Universität Hannover
09/2010	Diplomarbeit: *Parallelisierung rekursiver Algorithmen zur Optimierung mit Baumtopologie* an der Fakultät für Mathematik und Physik, Institut für Angewandte Mathematik, Prof. Dr. Steinbach
10/2005 - 09/2010	Studium des Diplom-Studiengangs Mathematik mit Nebenfach Physik an der Leibniz Universität Hannover
06/2004	Abitur am Gymnasium Großburgwedel
1997 - 2004	Gymnasium Großburgwedel
1995 - 1997	Orientierungsstufe Großburgwedel
1991 - 1995	Grundschule Großburgwedel

Beruflicher Werdegang

seit 09/2010	Wissenschaftlicher Mitarbeiter am Institut für Angewandte Mathematik, AG Algorithmische Optimierung (Prof. Dr. Steinbach), der Leibniz Universität Hannover
04/2009 - 09/2010	Wissenschaftliche Hilfskraft bei Prof. Dr. Steinbach am Institut für Angewandte Mathematik an der Leibniz Universität Hannover
10/2007 - 09/2010	Wissenschaftliche Hilfskraft in der Lehre am Institut für Angewandte Mathematik an der Leibniz Universität Hannover
10/2004 - 03/2005	Grundwehrdienst: Stabsdienstsoldat im Divisionsstab Hannover
07/2004 - 09/2004	Grundwehrdienst: Allgemeine Grundausbildung in Wesendorf