Fast exact algorithms for optimization problems in resource allocation and switched linear systems

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Zeyang Wu

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Prof. Qie He, Advisor

June, 2019

Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in graduate school.

Firstly, I would like to express my sincere gratitude to my advisor, Prof. Qie He, for his continuous and invaluable support during the last five years not only in academic research but also other aspects of life. I have been learning from his broad knowledge, innovative thinking, and pursuit of mathematical rigor and integrity. This dissertation would not have been possible without him. It has been an honor to be his first student.

Special thanks go to the rest of my thesis committee members: Prof. Ravi Janardan, Prof. Jean-Philippe Richard, Prof. Shuzhong Zhang. I also want to thank Prof. Kevin Leder for serving on my thesis proposal committee. I am grateful for their precious time and insightful comments on my research proposal and dissertation.

My friends consist of a great part of my wonderful and exciting life at the University of Minnesota. Special thanks go to Xiang Gao, Tianyi Chen, Ruizhi Shi, Zhiyuan Xu, Guiyun Feng, Yuanchen Su, Xiaoye Su, Chenglong Ye, Kaiyu Wang, Wenbo Dong, Zhenhuan Zhang, Xiaochen Zhang, Kameng Nip, Changning Wei, Zhuoxian Liang.

Finally, I would like to thank my parents and my sister for their endless love. This dissertation is dedicated to them.

Zeyang Wu

Minneapolis, May 2019

Dedication

To those who held me up over the years

Abstract

Discrete optimization is a branch of mathematical optimization where some of the decision variables are restricted to real values in a discrete set. The use of discrete decision variables greatly expands the scope and capacity of mathematical optimization models. In the era of big data, efficiency and scalability are increasingly important in evaluating the performance of an algorithm. However, discrete optimization problems usually are challenging to solve. In this thesis, we develop new fast exact algorithms for discrete optimization problems arising in the field of resource allocation and switched linear systems.

The first problem is the discrete resource allocation problem with nested bound constraints. It is a fundamental problem with a wide variety of applications in search theory, economics, inventory systems, etc. Given B units of resource and n activities, each of which associated with a convex allocation cost $f_i(\cdot)$, we aim to find an allocation of resources to the n activities, denoted by $\boldsymbol{x} \in \mathbb{Z}^n$, to minimize the total allocation cost $\sum_{i=1}^{n} f_i(x_i)$ subject to the total amount of resource constraint as well as lower and upper bound constraints on total resource allocated to subsets of activities. We develop a $\Theta(n^2 \log \frac{B}{n})$-time algorithm for it. It is an infeasibility-guided divide-and-conquer algorithm and the worst-case complexity is usually not achieved. Numerical experiments demonstrate that our algorithm significantly outperforms a state-of-the-art optimization solver and the performance of our algorithm is competitive compared to the algorithm with the best worst-case complexity for this problem in the literature.

The second problem is the minimum convex cost network flow problem on the dynamic lot size network. In the dynamic lot size network, there are one source node and n sink nodes with demand $d_i, i = 1, \ldots, n$. Let $B = \sum_{i=0}^{n} d_i$ be the total demand. We aim to find a flow $\boldsymbol{x}$ to minimize the total arc cost and satisfy all the flow balance and capacity constraints. Many optimization models in the literature can be seen as special cases of this problem, including dynamic lot-sizing problem and speed optimization. It is also a generalization of the first problem. We develop the Scaled Flow-improving Algorithm. For the continuous problem, our algorithm finds a solution that is at most ϵ away from an optimal solution in terms of the infinity norm in $O(n^2 \log \frac{B}{n\epsilon})$ time. For

the integer problem, our algorithm terminates in $O(n^2 \log \frac{B}{n})$ time. Our algorithm has the best worst-case complexity in the literature. In particular, it solves the discrete resource allocation problem with nested bound constraints in $O(n \log n \log \frac{B}{n})$ time and it also achieves the best worst-case complexity for that problem. We conduct extensive numerical experiments on instances with a variety of convex objectives. The numerical result demonstrates the efficiency of our algorithm in solving large-sized instances.

The last problem is the optimal control problem in switched linear systems. We consider the following dynamical system that consists of several linear subsystems: K matrices, each chosen from the given set of matrices, to maximize a convex function over the product of the K matrices and the given vector. This simple problem has many applications in operations research and control, yet a moderate-sized instance is challenging to solve to optimality for state-of-the-art optimization software. We prove the problem is NP-hard. We propose a simple exact algorithm for this problem. Our algorithm runs in polynomial time when the given set of matrices has the oligo-vertex property, a concept we introduce for a set of matrices. We derive several easy-to-verify sufficient conditions for a set of matrices to have the oligo-vertex property. In particular, we show that a pair of binary matrices has the oligo-vertex property. Numerical results demonstrate the clear advantage of our algorithm in solving large-sized instances of the problem over one state-of-the-art global solver. We also pose several open questions on the oligo-vertex property and discuss its potential connection with the finiteness property of a set of matrices, which may be of independent interest.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Discrete optimization

This thesis is dedicated to developing fast exact algorithms for several nonlinear discrete optimization problems. Discrete optimization is a branch of mathematical optimization where some of the decision variables are restricted to real values in a discrete set. When the discrete set is a set of integers, it is known as integer programming. When the discrete set is a set of combinatorial structures, such as permutations, combinations, or paths and trees in a graph, it is known as combinatorial optimization. Integer programming and combinatorial optimization are closely related: combinatorial optimization problems can be modeled as integer programs and conversely, many integer programming problems have meaningful combinatorial interpretations. In its generic form, discrete optimization can be formulated as follows:

$$\begin{aligned} &\min f(\boldsymbol{x}) \\ &\text{s.t.} \quad \boldsymbol{x} \in S, \end{aligned}$$

where $f(\boldsymbol{x})$ is the objective function and S is the feasible set.

The use of discrete decision variables greatly expands the scope and capacity of mathematical optimization models. Discrete optimization models are ubiquitous. Perhaps the most well-known example is the traveling salesman problem (TSP), which asks for the shortest tour to visit each city in a list exactly once [1, 2, 3]. This problem attracted a lot of attention after the RAND Corporation offered prizes for solving the problem in the 1950s. Since then, TSP and its variants have been studied extensively and the related algorithms have been applied to solve these problems in many areas such as logistics, transportation, and the semiconductor industry [4, 5, 6, 7]. Another example is the shortest path problem (SPP), which asks for a path of shortest length from one node to another node in a given network. Classic algorithms of SPP have been widely implemented by companies such as Google, Uber, and Lyft to find the shortest path between two locations. SPP also appears as a subproblem in many algorithms for other problems, including vehicle routing problems and network flow problems. The third example is the knapsack problem, which asks for the most profitable way of packing items into a knapsack. Since its first appearance in [8], it has been studied as a classic decision-making problem for more than one century with a variety of applications [9].

1.2 Nonlinear discrete optimization

In the simplest form of discrete optimization, the objective $f(\boldsymbol{x})$ is often a linear function. However, a nonlinear objective function arises naturally in some applications as it can provide a more accurate description of the relationship between the objective and the decision variables. One example is the speed optimization problem over a path [10, 11, 12], the goal of which is to find the optimal speed between two consecutive nodes over a fixed path to minimize the total cost, including fuel and emission cost over the arcs and possible customer inconvenience cost over the nodes. In maritime transportation, the fuel cost can be modeled as a cubic function of the speed and the customer inconvenience is usually modeled as a convex quadratic function of the arrival time. In aircraft transportation, the fuel cost is a polynomial function of order three [13, 14]. Another example is the resource allocation problem, a prototype problem that asks to find an optimal allocation of a fixed amount of resources to activities so as to minimize the cost incurred by the allocation [15]. The allocation cost is assumed to be nonlinear in most of the literature.

Algorithm design is one of the central themes in nonlinear discrete optimization. Nonlinear discrete optimization problems are very challenging to solve. The difficulty mainly comes from two aspects: combinatorial explosion and nonlinearity. There are two main directions in attacking such a problem. One direction is to formulate the problem as a mixed-integer nonlinear program (MINLP) and then solve it with off-the-shelf optimization software, such as Gurobi [16] and Cplex [17]. The software employs numerous algorithms and techniques developed for MINLP, including branch and bound, the cutting planes, column generation, and different decomposition approaches [18, 19]. The other direction is to exploit the problem structure and develop specialized algorithms. Several general approaches such as dynamic programming and the greedy strategy can be applied to assist the design of specialized algorithms. Examples in this direction include many classic algorithms such as Dijkstra's algorithm for SPP with nonnegative arc costs [20], and Kruskal's and Prim's Algorithms for minimum spanning tree [21].

1.3 Computational complexity of an algorithm

Intuitively, algorithms are step-by-step strategies for solving a particular problem. In 1937, Turing introduced the first model of computation, the Turing machine, that defines an algorithm in a precise way [22]. Thereafter, analyzing the efficiency of algorithms attracted significant attention. One of the most popular measures of efficiency is the asymptotic worst-case complexity. Based on this criterion, the Cobham-Edmonds thesis, named after Alan Cobham [23] and Jack Edmonds [24], asserts that a problem is tractable if it has a polynomial-time algorithm.

Definition 1. *[25] A **polynomial-time** algorithm is a Turing machine such that if n is the number of symbols on the input tape (the size of the input), then the machine is guaranteed to halt after a number of steps that is at most $cn^k + d$, for some constants k, c, and d.*

Big-oh notation is adapted to distinguish the asymptotic performance of two algorithms:

Definition 2. *Given two scalar functions f and g defined on some subset of real numbers, we write $f(x) = O(g(x))$ as $x \to \infty$, if there exist α and $x_0 \in \mathbb{R}$ such that $|f(x)| \leq \alpha|g(x)|$ for all $x \geq x_0$.*

Under the asymptotic performance criterion, an $O(n^2)$-time algorithm is preferable to an $O(n^3)$-time algorithm regardless the constants hidden in the big-oh notation. The notion of polynomial-time algorithm provides researchers a well-defined qualitative approach when attacking new problems. The race to obtain the best big-oh guarantees has given rise to many of the fundamental algorithmic techniques. This notion is amazingly successful in theory as well as practice.

1.3.1 Class P and class NP

Perhaps the most fundamental open question under this theme is the infamous P = NP conjecture. Despite great efforts that have been dedicated to the conjecture, a positive or negative answer is unfortunately absent. Class P and class NP are sets of decision problems. A decision problem is a problem that has yes or no answers. For example,

the TSP decision problem is stated as follows: given a list of n cities and a given value v, is there a TSP tour with a cost less than v? The short notion P denotes those decision problems that have polynomial-time algorithms. In 1971, Cook [26] studied class NP, a possibly more general class of decision problems for which a certificate to the yes instance can be verified in polynomial time. The shorthand NP comes from the view of non-deterministic Turing machines. For example, the TSP decision problem is in this class: to verify the a "yes" instance, we can compute the cost of the given certificate with $O(n)$ additions. The P = NP conjecture essentially asks whether an efficient polynomial-time algorithm exists for any decision problem in which each "yes" instance has a certificate that can be verified in polynomial time. An important concept in exploring the P = NP conjecture is NP-completeness. A problem is NP-complete if it is a problem in class NP and any problem in class NP can be transformed into it in polynomial time. The existence of a polynomial-time algorithm for any NP-complete problem leads to a positive answer to the P = NP conjecture. In [26], Cook proved that the boolean satisfiability problem is NP-complete. One year later in [27], Karp proved the NP-completeness of a set of 21 problems. The list of NP-complete problems has grown longer ever since, with thousands of computational problems in a variety of disciplines. The TSP decision problem is also NP-complete. Finding an efficient algorithm for any TSP instance seems hopeless unless P = NP.

The notion of polynomial-time relies on a specific model of computation. Class NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a non-deterministic Turing machine, while class P is class NP's counterpart on a deterministic Turing machine. Indeed, the "Turing machine" in the previous definition of class P and NP can be replaced by some other machines, e.g., random-access machine. The meaning of P and NP will stay unchanged as long as the basic operations of the new machine can be simulated by a Turing machine with a polynomial number of operations. To avoid the over dependence of the low-level details such as moving tapes or encoding, we may favor high-level basic operations such as arithmetic operations. The worst-case running time of an algorithm, i.e., computational complexity is the maximum number of basic operations the algorithm performs as a function of its input length [28]. The basic operations vary among different models of computation. In a deterministic Turing machine, the legal basic operations include

the arithmetic operations (e.g., addition, multiplication, division) of rational numbers represented by finite binary strings. In a BSS machine [29], the legal basic operations are arithmetic operations of real numbers. Under two models of computation, the complexity of the same algorithm may be different, e.g., the polynomial-time solvable class of the BSS machine is different from P. To be compatible with most of the literature, complexity analysis in this thesis is always based on a variant of the deterministic Turing machine: *the oracle Turing machine*. Oracle Turing machine is the standard Turing machine with an extra oracle tape that can magically solve some decision problem in one basic operation [28]. By its definition, the oracle could be too powerful to be practical in the real world, e.g., solve TSP in $O(1)$ time. We hereby restrict the oracle to function oracles. In the presence of convex functions, a Turing machine with function oracles is desired: a convex function is assumed to be an oracle through which we can query its value at any point in constant time ($O(1)$ time). This assumption is reasonable from a practical viewpoint. Most of the functions in practical application are polynomials, exponentials, and logarithms that can be evaluated with $O(1)$ arithmetic operations. In some scenarios where the explicit form of the cost function is not known, the value can usually be estimated through some fixed procedure with $O(1)$ arithmetic operations. With such an oracle Turing machine, we can measure the complexity of an algorithm as the maximum number of basic operations (including arithmetic operations and function oracle queries) the algorithm performs as a function of its input length. We will give a more detailed discussion on models of computation and complexity classes in Section 1.6.1.

1.3.2 Exact, approximation, and meta-heuristic algorithms

A natural question for problems like TSP is: if an efficient algorithm seeking for the optimal solution is unlikely to exist, can we efficiently find a solution that is "close" to the optimal solution? Besides the theoretical complexity, people are also interested in the practical performance of an algorithm. There have been extensive studies revolving around this question from both theoretical and practical perspectives. For a particular discrete optimization problem, its algorithms can be roughly classified into three categories:

(a) Exact algorithms that always solve the optimization problem to optimality.

(b) Approximation algorithms that find a solution with provable performance guarantees compared to the optimal one.

(c) Heuristic algorithms and meta-heuristic algorithms that find a solution without any theoretical guarantee on its performance.

We illustrate these ideas with the example of a special case of TSP called symmetric metric TSP, which we denote it by mTSP. In mTSP, the distance function is symmetric and satisfies the triangle inequality.

At first glance, mTSP seems easy to solve by enumeration since the solution set is finite. However, exhaustive search is not a tractable approach because of the combinatorial explosion: the cardinality of the solution set grows exponentially with respect to the size of the problem. The solution set of mTSP can be viewed as a set of permutations of n cities, the number of which is as many as $\frac{(n-1)!}{2}$. When $n = 30$, the number of possible solutions is around 1.32×10^{32}. Even with the fastest supercomputer in the world, it would take thousands of years to enumerate all possible solutions. Such an enumeration algorithm is impractical even for a list of only 20 cities. A better exact algorithm, the Bellman-Held-Karp algorithm [30, 31] uses the idea of dynamic programming, has a complexity of $O(2^n n^2)$ and it can solve instances with 30-40 cities. Currently, the best exact algorithms are implementations of the branch-and-bound algorithm and problem-specific cutting-planes [32, 7], which can solve instances with $85,900$ cities.

For approximation algorithms, instances of much larger size can be solved with some sacrifice of quality of the solution. An α-approximation algorithm is an algorithm that always outputs a solution whose cost is at most α-times the cost of the optimal solution [21]. A simple 2-approximation $O(n^2)$-time algorithm based on minimum spanning tree can solve any TSP instance of millions of nodes (approximately) in seconds. The algorithm with the best approximation ratio so far, Christofides' Algorithm [33], achieves a $\frac{3}{2}$-approximation with an $O(n^3)$ running time.

Many other heuristic and meta-heuristic algorithms also have been applied to mTSP, e.g., LinKernighan heuristics [34, 35], simulated annealing [36], tabu search [37], evolution algorithms [38], etc. Modern heuristic methods can find solutions for extremely

large problems (millions of cities) within a reasonable amount of time whose value is 2-3% higher than the optimal solution with high probability [39]. Even though there is no theoretical guarantee in general on how good the solution is, heuristic algorithms provide a practical way to solve a hard problem and they do produce ans optimal solution in many cases.

In general, we can loosely state that algorithm design is a pursuit of both efficiency and solution quality. For some problems, the exact algorithms are too slow to be practical and we then turn to the approximation algorithms, heuristic algorithms, or a combination of them. For some problems, efficiency and quality of the solution can be achieved simultaneously by well-designed algorithms.

For a nonlinear discrete optimization problem, we are interested in the following questions:

- Which complexity class does the problem belong to?
- What algorithm can solve the problem exactly or approximately?
- What is the complexity of the algorithm?
- What is the performance of the algorithm for instances from real world applications?

In this thesis, we pursue fast exact algorithms for three particular nonlinear discrete optimization problems. We introduce the problems and summarize our contribution in the next section.

1.4 Three nonlinear discrete optimization problems

1.4.1 Discrete resource allocation problem with nested bound constraints

The first problem is a resource allocation problem (RAP) that asks to find an optimal allocation of a fixed amount of resources to activities so as to minimize the cost incurred by the allocation [15]. It has a wide variety of applications ranging from portfolio selection, production planning, to video-on-demand batching and telecommunications [15, 40, 41]. In its simplest form, RAP aims to minimize a separable convex

function under a single constraint concerning the total amount of resources to be allocated. In several applications, however, such single global resource bound is not sufficient to model partial budget or investment limits, release dates and deadlines, or inventory or workforce limitations. In these situations, additional constraints over the cumulative resource allocated are desired. We are interested in the following discrete optimization problem (P1):

> Given B units of resource and n activities, each of which associated with a convex allocation cost $f_i(\cdot)$, find an allocation of resource to the n activities, denoted by $\boldsymbol{x} \in \mathbb{Z}^n$, to minimize the total allocation cost $\sum_{i=1}^{n} f_i(x_i)$ subject to the total amount of resource constraint as well as lower and upper bound constraints on total resource allocated to subsets of activities.

Our interest of studying (P1) stems from the scheduling problem over a fixed route [12], where the resource allocated to each activity is the time spent between two consecutive customers over the route and the nested constraints correspond to the time-window constraints imposed on the customers.

Our contributions. We develop a $\Theta(n^2 \log \frac{B}{n})$-time algorithm for (P1). Our algorithm essentially combines the divide-and-conquer framework in [42] with the scaling-based algorithm in [43]. It is an infeasibility-guided divide-and-conquer algorithm. The worst-case time complexity of our algorithm is worse than the best time complexity in the literature, $O(n \log n \log B)$. z‘ For a specific instance, however, the number of subproblems solved by our algorithm may be significantly fewer than the current best algorithm, a pure divided-and-conquer algorithm that always divides the problems into subproblems of equal size. The worst-case complexity is usually not achieved. Finally, to the best of our knowledge, there are no numerical experiments on (P1) in the literature. We evaluate the performance of our algorithm on a set of test instances with linear and quadratic functions, and then compare the results with the performance of a state-of-the-art mixed-integer linear and quadratic optimization solver–Gurobi [16]. Our algorithm significantly outperforms Gurobi, solving instances of much larger sizes with less computational time. Moreover, we perform a comprehensive numerical test on

test instances with three classes of convex objectives and compare the results with the performance of another divide-and-conquer based algorithm, MDA, which has the best worst-case time complexity for (P1) in the literature [44]. Our algorithm solves substantially fewer subproblems then MDA and the performance is competitive compared to MDA on the benchmark instances in the literature.

1.4.2 Minimum convex cost flow problem over the dynamic lot size network

The second problem is a generalization of (P1). It is a minimum convex cost network flow problem on the dynamic lot size network $G = (N, A)$:

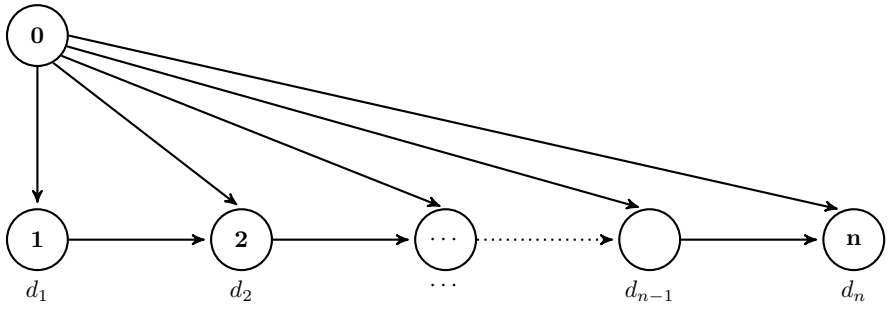


Figure 1.1: The dynamic lot size network.

The dynamic lot size network is defined as follows: there are one source node 0 and n sink nodes $1, 2, \cdots, n$. Each arc $e \in A$ is associated with a lower bound 0, an upper bound u_e, and a convex cost function $f_e(\cdot)$. We are interested in the following optimization problem (P2)

> Given a dynamic lot size network $G = (N, A)$ described in Figure 1.1, find a flow $\boldsymbol{x}$ on G to minimize the total arc cost $\sum_{e \in A} f_e(x_e)$ and satisfy all the flow balance and capacity constraints.

We study the continuous version of (P2) as well as its restriction to integer flows. Many optimization models in the literature can be seen as special cases of (P2). We give three examples here. The first example is the dynamic lot-sizing model with linear

or convex production cost and holding cost. The second example is (P1), the resource allocation problem with nested bound constraints on resourced consumed by consecutive activities, which is the underlying model in many engineering applications [15, 41]. The third example is the speed optimization problem over a path [12], which appears as a subproblem in many transportation and logistics applications. The goal is to find the optimal speed $v_{i,i+1}$ between two consecutive nodes over a fixed route while subjected to a prescribed time window to minimize the total cost (including fuel and emission cost over the arcs, and possible customer inconvenience cost over the nodes). The speed optimization problem can be transformed into a minimum convex cost problem over the dynamic lot size network.

Our contributions. The best result known for the integer version of (P2) is $O(n^2 \log n \log \frac{B}{n})$, based on the capacity scaling algorithm for the minimum convex cost network flow problem [45]. We design the Scaled Flow-improving Algorithm (SFA), a new scaling-based algorithm. The running time of our algorithm is $O(n^2 \log \frac{B}{n\epsilon})$ for the continuous problem for finding a solution that is at most ϵ away from an optimal solution in terms of the infinity norm, and $O(n^2 \log \frac{B}{n})$ for the integer problem where $B = \sum_{i=0}^{n} d_i$. The $\log n$ factor improvement over the current best algorithm is based on two ingredients: (1) a new scaling framework that is not based on LP duality as existing capacity scaling algorithms; (2) a stronger proximity result between the optimal solutions of the original problem and the scaled problem than the existing results on the minimum convex cost flow problem.

In particular, for the resource allocation problem with nested constraints (P1), our algorithm terminates in $O(n \log n \log \frac{B}{n\epsilon})$ time for the continuous problem and $O(n \log n \log \frac{B}{n})$ time for the integer problem with $B = \sum_{i=0}^{n} d_i$. The speed up is obtained by using the data structure segment tree and red-black tree for two key operations in our algorithm. The complexity matches the best result in the literature [44].

Finally, we conduct extensive numerical experiments on instances with a variety of convex objectives. The numerical result demonstrates the efficiency of our algorithm in solving large size instances.

1.4.3 Optimal switch sequence for switched linear systems

The third problem is the optimal control problem of discrete-time switched linear systems. Many real-world systems exhibit significantly different dynamics under various modes or conditions, for example a manual transmission car operating at different gears, a chemical reactor under different temperatures and flow rates of reactants, and a group of cancer cells responding to different drugs. Such phenomena can be modeled under a unified framework of switched systems. A switched system is a dynamical system that consists of several subsystems and a rule that specifies the switching among the subsystems. Finding a switching rule to optimize the dynamics of a switched system under certain criteria has found numerous applications in power system operations, chemical process control, air traffic management, and medical treatment design [46, 47, 48, 49]. We study the following discrete-time switched linear system:

$$x(k+1) = T_k x(k), \qquad T_k \in \Sigma,\ k = 0, 1, \ldots, \tag{1.1}$$

where $x(k)$ is an n-dimensional real vector that captures the system state at period k, the set Σ contains m given $n \times n$ real matrices, each of which describes the dynamics of a linear subsystem, and the initial vector $x(0)$ is a given n-dimensional real vector a. Such a system with switching only at fixed time instants appear in many practical applications, and is also employed to approximate the more complex dynamics of a continuous-time hybrid system with switching rules defined over the real line [46, 48].

We are interested in the following optimization problem (P3) related to the system in 1.1:

> Given a switched linear system described in (1.1), a positive integer K, and a convex function $f : \mathbb{R}^n \to \mathbb{R}$, find a sequence of K matrices $T_0, T_1, \ldots, T_{K-1} \in \Sigma$ to maximize $f(x(K))$.

One type of such convex functions are the ℓ_p norms.

Example 1. *Consider a switched linear system consisting of two subsystems with system matrices* $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ *and* $B = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, *an initial vector* $a = (2, 1)^\top$, *and* $K = 8$.

Figure 1.2 illustrates the trajectory of $x(k)$ under three switching sequences, with the final state $x(8)$ being $(53, 23)^\top$, $(58, 41)^\top$, and $(71, 41)^\top$, respectively.

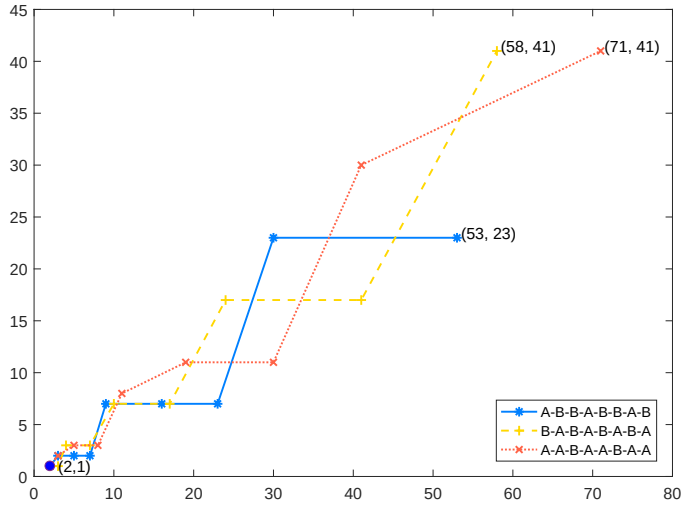


Figure 1.2: The trajectory of $x(k)$ under different matrix sequences

(P3) has many practical applications and is closely connected to several fundamental problems in control and optimization. We give three examples here. Firstly, it can model the process of mitigating antibiotic resistance, in which each component of $x(k)$ represents the percentage of a genotype of an enzyme produced by bacteria, each matrix represents the mutation rates between different genotypes under certain antibiotic, and the goal is to maximize the population of the wild type after K periods. Secondly, it generalizes the matrix K-mortality problem which asks whether the zero matrix can be expressed as a product of K matrices in Σ (duplication allowed). Thirdly, it is closely related to computing the joint spectral radius of a set of matrices which generalizes the spectral radius of one matrix and has found wide applications in a variety of seemingly unrelated fields, such as wavelet functions, switched systems, constrained coding, and network security management.

Our contributions. We show that (P3) is NP-hard for a pair of stochastic matrices or binary matrices. We propose a polynomial-time exact algorithm for the problem when all input data are rational and the given set of matrices Σ has the oligo-vertex property, a new concept we introduce below. Let $P_k(\Sigma, a)$ be the convex hull of all

possible values of $x(k)$, i.e.,

$$P_k(\Sigma, a) := \text{conv}(\{x(k) \mid x(k) = T_{k-1} \cdots T_0 a, T_j \in \Sigma, j = 0, \ldots, k-1\}).$$

Let $N_k(\Sigma, a)$ be the number of extreme point of $P_k(\Sigma, a)$ and $N_k(\Sigma) = \sup_{a \in \mathbb{R}^n} \{N_k(\Sigma, a)\}$. A set of matrices Σ is said to has the **oligo-vertex property** if there exists some constant d such that $N_k(\Sigma) = O(k^d)$. The oligo-vertex property indicates that the number of extreme points of $P_k(\Sigma, a)$ grows at most polynomially in k regardless of the initial vector a, although the number of possible values of $x(k)$ grows exponentially with k in general.

We derive a set of sufficient and easy to verify conditions for a set of matrices to have the oligo-vertex property: (1) A finite set of matrices that commute; (2) A finite set of matrices containing at most one matrix with rank higher than one; (3) A pair of 2×2 matrices sharing at least one common eigenvector; (4) A pair of 2×2 binary matrices. We also show that the oligo-vertex property is invariant under a similarity transformation. In particular, we show that $N_k(\Sigma) = O(k^4)$ when Σ is a pair of 2×2 binary matrices. Finally, we conjecture that any pair of 2×2 real matrices has the oligo-vertex property.

1.5 Organization of the thesis

The remainder of this thesis is organized as follows. We first complete Chapter 1 with a section of preliminary knowledge. Then we present fast exact algorithms for the three nonlinear discrete optimization problem in Chapter 2, Chapter 3, and Chapter 4, respectively.

1.6 Preliminary

In this section, we provide preliminary material for solving the three nonlinear discrete optimization problems.

1.6.1 Model of computation

The model of computation forms the base of computational complexity theory. In this section, we elaborate the mathematical definitions of problems, Turing machines, and complexity classes. Most of the notations and definitions are adapted from [28].

Problems

In computational complexity theory, a problem refers to an abstract question to be solved. In contrast, an instance of this problem is a concrete realization of the abstract question. An instance serves as input of an algorithm and hence must provide the necessary information (data) to answer the question. For example, an instance of TSP includes a list of n cities and n^2 pairwise distances between the them, which are given by n^2 rational numbers. Usually, the data of an instance is encoded as a string under some pre-specified rules. The length of the string is called the size of the instance. We use *semi-group* to describe such relation.

Definition 3. *(**Semi-group**) A semigroup is a set S with a closed binary operation $\cdot$ on S that satisfies the associative property, i.e., $\forall a, b, c \in S$, we have*

$$a \cdot b \in S, \quad a \cdot (b \cdot c) = (a \cdot b) \cdot c.$$

For example, the set of finite length binary strings is a semi-group with string concatenation as the associated binary operation. Let $\{0,1\}^*$ denote the semigroup of finite binary strings. For an instance (or a member) $x \in \{0,1\}^*$, let $|x|$ denote the length of instance x. Since the solution of an instance can also be encoded in string, we hereby use a function $f : \{0,1\}^* \to \{0,1\}^*$ to describe the mapping from the instances to their solutions. Among these functions, of particular interest in computational theory are the ones with binary output, which is also known as *decision problems.*

Definition 4. *(**Decision problem**) A decision problem $f : \{0,1\}^* \to \{0,1\}$ is a problem has yes or no answers.*

Given a decision problem $f : \{0,1\}^* \to \{0,1\}$, let

$$\Sigma := \{x \mid x \in \{0,1\}^*, f(x) = 1\}$$

denote all the yes-instances, i.e., the instances with a positive answer. In this perspective, the decision problem can be viewed as a membership problem that asks whether a given instance x belongs to the set Σ or not. Σ is called a *language*. The term language and decision problem are used interchangeably in the literature. Here are a few examples of decision problem:

- **The decision TSP problem:** Given a list of n cities and a given value v, is there a TSP tour whose cost is less than v?
- **The decision version of the shortest path problem:** Given a graph $G = (V, E)$ with associated cost, two nodes s and t, and a given value v, is there a path from node s to node t such that the cost is less than v?
- **The decision version of linear programming:** Given a linear program $\max\{c^\top x : Ax \leq b\}$ and a given value v, is there a feasible solution such that the objective value is greater than v?

Yet there are some computational problems not easily expressed as decision problems, the framework of decision problems turns out to be surprisingly expressive [28]. In particular, the three problems above are examples of casting the original optimization problem as a decision problem. By imposing a bound on the objective value, we can transform the original optimization problem into a series of decision problems. The number of decision problems is a logarithm of the bound if binary search is employed.

Big-oh notation

In the above definition, the data of an instance is assumed to be a finite-length binary string. Furthermore, the encoding rules, are not specified. The length of the same instance, however, may vary if different encoders are employed. To avoid such lower-level detail, we introduce the big-oh notation:

Definition 5. ***(big-oh notation)*** *Given two scalar functions f and g defined on some subset of real numbers, we write $f(x) = O(g(x))$ as $x \to \infty$, if there exist α and $x_0 \in \mathbb{R}$ such that $|f(x)| \leq \alpha |g(x)|$ for all $x \geq x_0$.*

With big-oh notation, the length of the encoding is indifferent with respect to the alphabet allowed, i.e., the $\{0, 1\}$ can be replaced by other finite sets. For example, to

encode n positive integer numbers that are less than a thousand, we need a binary string of length $8n = O(n)$. If we use $\{0, 1, \cdots, 9\}$, the decimal encoding, the length of the string should be at least $4n$, which is also $O(n)$.

Turing machine

The input and output of an instance are defined as strings in a semi-group. In computational complexity, an algorithm is a machine that computes the output based on the input string fed to it. A Turing machine is a mathematical model of computation that defines such abstract machine.

Definition 6. *[28] (**deterministic Turing machine**) A deterministic Turing machine M is formally described by a tuple (Γ, Q, δ) containing:*

- *A set Γ of the symbols that Ms tapes can contain. We assume that Γ contains a designated blank symbol, denoted $\square$, a designated start symbol, denoted $\triangleright$ and the numbers 0 and 1. We call Γ the alphabet of M.*
- *A set Q of possible states Ms register can be in. We assume that Q contains a designated start state, denoted q_{start} and a designated halting state, denoted q_{halt}.*
- *A function $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, S, R\}$ describing the rule M uses in performing each step. L is the movement to the left, S is no movement, and R is the movement to the right This function is called the transition function of M.*

With a deterministic Turing machine, we can then formalize the notion of running time.

Definition 7. *(**Running time of a deterministic Turing machine**) Let $f : \{0,1\}^* \to \{0,1\}^*$ and let $T : N \to N$ be some functions, and let M be a deterministic Turing machine. We say that M computes f in $O(T(n))$-time if for every $x \in \{0,1\}^*$, if M is initialized to the start configuration on input x, then after at most $O(T(|x|))$ steps it halts with $f(x)$ written on its output tape. We say that M computes f if it computes f in $O(T(n))$ time for some function $T : N \to N$.*

Class P and Class NP

Complexity classes are sets of problems of related resource-based complexity. Class P and class NP are time-complexity classes that classify the problems by bounding the time used by the algorithm. The class P is considered to be the class of decision problems that are efficiently solvable.

Definition 8. *(**Class P**) A decision problem $f : \{0,1\}^* \to \{0,1\}$ is in class P if and only if there exist a deterministic Turing machine M, a polynomial function $T(n) = n^k$ for some constant k, such that M computes f in $O(T(n))$ time.*

The shorthand notion P stands for polynomial-time solvable. Such algorithms in Definition 8 are also known as *polynomial-time algorithms*. One of the other most well-known complexity classes is class NP. Class NP is a possibly more general class of decision problems for which a certificate to the yes instance can be verified in polynomial time.

Definition 9. *[28] (**Class NP**) A decision problem $f : \{0,1\}^* \to \{0,1\}$ is in class NP if and only if there exist a deterministic Turing machine M, two polynomial function $T(n) = n^{k_1}, p(n) = n^{k_2}$ with constants k_1, k_2, such that*

$$f(x) = 1 \Leftrightarrow \exists u \in \{0,1\}^{O(p(|x|))}, M(x,u) = 1.$$

and M computes $M(x,u)$ in $O(T(n))$ time. If such u exists, we call u a certificate of the problem f. Here $\{0,1\}^{O(p(|x|))}$ denotes the set of binary strings of length less than $O(p(|x|))$.

A problem is in class NP if for every yes instance x, there exists a certificate, u, whose size is polynomial in $|x|$, and an algorithm, M, that can verify this certificate u in polynomial time. The infamous P = NP conjecture essentially asks whether the existence of efficient algorithm that verifies a certificate to the yes instance implies the existence of efficient algorithm that solves the problem. Equivalent to the above verifier-based definition, class NP can be defined similarly as class P in the view of non-deterministic Turing machine. Actually, the notation NP stands for non-deterministic polynomial-time solvable.

Definition 10. *[28] (**non-deterministic Turing machine**) A non-deterministic Turing machine M is formally described by a tuple $(\Gamma, Q, \delta_0, \delta_1)$ containing:*

- *A set Γ of the symbols that Ms tapes can contain. We assume that Γ contains a designated blank symbol, denoted $\square$, a designated start symbol, denoted $\triangleright$ and the numbers 0 and 1. We call Γ the alphabet of M.*
- *A set Q of possible states Ms register can be in. We assume that Q contains a designated start state, denoted q_{start}, a designated halting state, denoted q_{halt}, and a designated accept state, denoted q_{accept}.*
- *Two function $\delta_0, \delta_1 : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, S, R\}$ describing the rule M uses in performing each step. L is the movement to the left, S is no movement, and R is the movement to the right. These functions are called the transition function of M.*

A non-deterministic Turing machine is distinct from a deterministic Turing machine as it have one more transition function. At each step of execution, the non-deterministic Turing machine M makes an arbitrary choice on which of its two transition functions to apply. It outputs 1 on a given input x if there is some sequence of these choices with which M reaches q_{accept} on input x. Otherwise, if every sequence of choices makes M halt without reaching q_{accept}, then M outputs 0. The running time of a non-deterministic Turing machine is the maximum operation required under such sequences.

Definition 11. *[28] (**Running time of a non-deterministic Turing machine**) Let $f : \{0,1\}^* \to \{0,1\}$ and let $T : N \to N$ be some functions, and let M be a non-deterministic Turing machine. We say that M computes f in $O(T(|x|))$-time if for every $x \in \{0,1\}^*$, if M is initialized to the start configuration on input x, then after at most $O(T(|x|))$ steps it either halts or reaches q_{accept} with $f(x)$ written on its output tape. We say that M computes f if it computes f in $O(T(|x|))$ time for some function $T : N \to N$.*

The sequence of choices made by an accepting computation of a non-deterministic Turing machine can be viewed as a certificate of the input x and vice versa. This property characterizes class NP as the set of non-deterministic polynomial-time solvable problems.

Definition 12. *(**Class NP**) A decision problem $f : \{0,1\}^* \to \{0,1\}$ is in class P if and only if there exist a non-deterministic Turing machine M, a function $T(n) = cn^k + d$ for some constants k, c, and d, such that M computes f in $T(n)$ time.*

Definition 12 and Definition 9 are equivalent. Class P and class NP are sets of polynomial-time solvable decision problems under different machines. However, the similarity in definition does not imply P = NP. Indeed, the algorithms defined by non-deterministic Turing machine is different from those defined by deterministic Turing machine. A non-deterministic Turing machine seems to be too powerful to be practical. On the other hand, deterministic Turing machine, though still very abstract, turns out to be expressive and the definition of computational complexity is flexible. Deterministic Turing machine can simulate any algorithms written in modern machine languages and in turn, any programming languages such as C, Java [28].

It is cumbersome to analyze the running time an algorithm with low-level operations such as moving a tape. Instead, it is more convenient to measure the efficiency in terms of high-level primitive operations e.g., arithmetic operations, insertion to a list. Therefore, a machine that is similar to programming languages is more favorable. Fortunately, unlike non-deterministic Turing machine, such machines will not affect the meaning of polynomial-time solvable class (class P). Let's elaborate on it here.

Oracle Turing machine

Recall that the computational complexity of an algorithm is defined as the maximum number of basic operations the algorithm performs as a function of its input length. Here, the basic operations, i.e., primitive operations, are the operations which have unit cost in a model of computation. In the deterministic Turing machine, the basic operations are very weak-the machine can read and write symbols, the head moves to the left or to the right.

Consider a special model of computation $\mathcal{U}$ with a set Δ of primitive operations and a polynomial function $T(n)$. For each primitive operation $p \in \Delta$, there exists a deterministic Turing machine that can stimulate p in $O(T(n))$ time. In this context, a deterministic Turing machine stimulate operation p means p can be replaced by a sequence of basic operations of deterministic Turing machine. This replacement always takes the same input and produces the same output. For any machine U of $\mathcal{U}$

whose running time is $O(G(n))$, it can be simulated by a Turing machine M whose running $O(G(n)T(n))$ time. Since the function $O(G(n)T(n))$ is polynomial if and only if $O(G(n))$ is polynomial, an algorithm on $\mathcal{U}$ is polynomial-time if it is polynomial-time on a deterministic Turing machine. Therefore, class P also characterizes the complexity class of decision problems that can be solved on machine $\mathcal{U}$ in polynomial time.

To formalize the above intuition, let's consider a variant of Turing machine: the oracle Turing machine.

Definition 13. *[28] (**oracle Turing machine**) An oracle Turing machine is a Turing machine M that has a special read/write tape we call Ms oracle tape and three special states q_{query}, q_{yes}, q_{no}. To execute M, we specify in addition to the input a language $O \subset \{0,1\}^*$ that is used as the oracle for M. Whenever during the execution M enters the state q_{query}, the machine moves into the state q_{yes} if $O(q) = 1$ and q_{no} if $O(q) = 0$, where q denotes the contents of the special oracle tape. Note that, regardless of the choice of O, a membership query to O counts only as a single computational step. If M is an oracle machine, $O \subset \{0,1\}^*$ a language, and $x \in \{0,1\}^*$, then we denote the output of M on input x and with oracle O by $M^O(x)$.*

Oracle Turing machine is the standard Turing machine with an extra oracle tape that can magically solve some decision problem in one basic operation [28]. Decision problems might be too limited. In a broader context, the oracle is a "black box" that is able to produce a solution for any instance of a given computational problem. We omit the discussion as in this thesis we restrict the oracles to be rather simple ones. The non-deterministic oracle Turing machine is defined similarly. Then we can define the polynomial-time solvable classes on the oracle Turing machine.

Definition 14. *[28] (**Class P^O and class NP^O**) For every $O \subset \{0,1\}^*$, P^O is the set of languages decided by a polynomial-time deterministic TM with oracle access to O and NP^O is the set of languages decided by a polynomial-time nondeterministic TM with oracle access to O.*

The oracle in the oracle Turing machine could be too powerful to be practical in the real world, e.g., solve TSP in $O(1)$ time. In general, $\mathrm{P}^O \neq \mathrm{P}$. Nevertheless, if we restrict O to be decision problems in P,

Claim 1. *[28, ?] If $O \in P$, then $P^O = P$.*

Examples of oracles in P includes arithmetic operations of finite length binary strings. Moreover, we are also interested in function oracles. In presence of convex functions, a Turing machine with function oracles is desired: a convex function is assumed to be an oracle through which we can query its value at any point in constant time ($O(1)$ time). This assumption is reasonable from a practical viewpoint. Most of the functions in practical application are polynomials, exponentials, and logarithms that can be evaluated with $O(1)$ arithmetic operations. In some scenarios when the explicit form of the cost function is not known, the value can usually be estimated through some fixed procedure with $O(1)$ arithmetic operations.

Model of computation in this thesis

In this thesis, our complexity analysis is based on a special oracle Turing machine. In this model, we assume that

1. The basic arithmetic operations (addition, subtraction, multiplication, division, and comparison) take a unit time step to perform, regardless of the sizes of the operands.

2. A function oracle can evaluate the value of the function at any point x in a unit time.

These assumption are widely used in measuring the complexity of optimization algorithm. Strictly speaking, the time-complexity of addition or subtraction is $O(\log B)$ where B the maximum bit-size of the operands. ing They are polynomial in the input size.

In summary, with such an oracle Turing machine, we measure the computational complexity of an algorithm as the maximum number of basic operations (including arithmetic operations, function oracle queries) the algorithm performs as a function of its input length. The complexity analysis in this thesis will follow this criterion.

BSS machine

BSS machine is a model of computation intend to describe computation over real numbers. Turing machine only works with finite strings and discrete symbols. However, real numbers may not be represented by finite-length strings since the set $\mathbb{R}$ is uncountable. Blum, Shub and Smale [29] introduced generalized Turing Machines (which refer to BSS machine) that compute over $\mathbb{R}$. In a standard Turing machine, the domain of the function f is the semi-group $\{0,1\}^*$, i.e., each cell of the string hold a binary symbol. In the BSS machine, the domain of the function f is the semigroup $\mathbb{R}^*$, i.e., each cell of the string hold a real number. Similar to P^{O} and NP^{O} on oracle Turing machines, the polynomial-time solvable classes on BBS machine, $\mathrm{P}_{\mathbb{R}}$ and $\mathrm{NP}_{\mathbb{R}}$, are defined respectively. It can be shown that $\mathrm{P} \subset \mathrm{P}_{\mathbb{R}}$.

Important concepts

Finally, we complete this section by a brief introduction to some important concepts in computational complexity theory: NP-completeness, NP-completeness, pseudo-polynomial-time algorithm, and strongly polynomial-time algorithm.

NP-completeness and NP-hardness

NP-completeness and NP-hardness characterize the most difficult problems in class NP.

Definition 15. *[28] (**Reductions, NP-hardness and NP-completeness**) We say that a language $A \subset \{0,1\}^*$ is polynomial-time Karp reducible to a language $B \subset \{0,1\}^*$ (sometimes shortened to just "polynomial-time reducible") denoted by $A <_p B$ if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*, x \in A$ if and only if $f(x) \in B$.*

We say that B is NP-hard if $A <_p B$ for every A in class NP. We say that B is NP-complete if B is NP-hard and B is in class NP.

The concept of polynomial-time reduction enable us to show a problem B is at least as hard as problem A. We say that $A <_p B$ if there exists some procedure f such that:

1. f terminates in polynomial time. ing

2. f transform any instance of problem A to an instance of problem B, i.e., f is a mapping that embeds A into B.

3. The answer to an instance x of A is yes if and only if the answer of $f(x)$, an instance of B is yes.

In other words, to solve problem A, we can transform it to an instance of problem B in polynomial time and solve the new instance independently. In this way, any algorithm solving problem B can be adapted to solve problem A with an extra polynomial-time preprocessing step.

The most challenging problems in class NP is the NP-complete problems since every problem in class NP is polynomial-tine reducible to them. Indeed, an efficient (polynomial-time) algorithm for any NP-complete problems will imply a positive answer to the P = NP conjecture.

Pseudo-polynomial-time

Definition 16. ***(Pseudo-polynomial-time)*** *An algorithm runs in pseudo-polynomial time if its running time is a polynomial in the numeric value of the input (the largest integer present in the input).*

A pseudo-polynomial time is not necessary a polynomial-time algorithm since it is only polynomial in the numeric value. The numeric value of the input is exponential in the input length. For example, an $O(n)$-time algorithm of a input integer no larger than n is pseudo-polynomial time but not polynomial-time.

Strongly and weakly polynomial-time algorithm

In some context where the input contains only integer values, we further differentiate the concept of polynomial-time between strongly and weakly polynomial-time.

Definition 17. *[50, 51]****(Strongly polynomial-time)*** *The algorithm runs in strongly polynomial time if (1) the number of operations in the arithmetic model of computation is bounded by a polynomial in the number of integers in the input instance; and (2) the space used by the algorithm is bounded by a polynomial in the size of the input.*

An algorithm is a weakly polynomial-time algorithm if it runs in polynomial time but not strongly polynomial time.

1.6.2 Minimum cost network flow problems

The *minimum cost network flow problem* (MCNFP) and the *minimum convex cost network flow problem* (MCCNFP) are classic combinatorial optimization problems in the field of operation research. In this section, we review the algorithms for two problems in the literature.

Problem formulation

MCNFP minimizes the sum of linear cost over the arcs in a directed network. Suppose $G = (N, A)$ is a directed graph with $n = |N|$ nodes and $m = |A|$ arcs, the minimum cost flow problem can be formulated as follows,

$$\min \quad \sum_{(i,j)\in A} f_{ij}(x_{ij}) \tag{1.2a}$$

$$\text{s.t.} \quad \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i), \qquad \forall i \in N, \tag{1.2b}$$

$$0 \le x_{ij} \le u_{ij}, \qquad \forall (i,j) \in A. \tag{1.2c}$$

where $f_{ij}(x_{ij}) = c_{ij}x_{ij}, \forall (i,j) \in A$, and $b(i)$ is the supply of node i if $b(i)$ is positive, and demand of node i otherwise. A natural generalization of MCNFP is MCCNFP. In MCCNFP, the functions $f_{ij}(\cdot), (i,j) \in A$ in (1.2) are general convex functions. Let U be the maximum capacity of an arc.

Algorithms for MCNFP

MCNFP is a linear program and is polynomial-time solvable. Most of the polynomial-time algorithms for MCNFP are developed with the help of the scaling techniques. One stream of studies scales the capacity. In [52], Edmonds and Karp introduced the successive shortest-path algorithm, the first weakly polynomial-time algorithm for MCNFP. In 1988, Orlin introduced a variant of Edmonds and Karps algorithm, *capacity-scaling algorithm*, which is an improved version of the successive shortest-path algorithm with

a scaling technique. Another stream of studies relies on the cost-scaling suggested by Rock [53] and, independently, Bland and Jensen [54]. The best implementation is due to Goldberg and Tarjan [55], which has a $O(nm \log \frac{n^2}{m} \log nC)$ where C is maximum unit cost of an arc. The two scaling techniques can be combined. The double scaling algorithm due to Ahuja et al. [56] achieved a complexity of $O(nm \log \log U \log nC)$. There are also other algorithms based on nonlinear optimization, e.g., Bertsekas et al. [57] developed a class of relaxation methods which iteratively improves the dual cost. The first strongly polynomial-time algorithms for MCNFPis proposed by Tardos [58]. Currently, the fastest implementation is given by the enhanced capacity-scaling algorithm due to Orlin [59] with a complexity of $O((m \log n)(m + n \log n))$.

Algorithms for MCCNFP

Unlike MCNFP, MCCNFP does not admit any strongly polynomial-time algorithm since the convex cost function may contain as many as $O(U)$ linear pieces (See an example in [43]). Strongly polynomial-time algorithms are only available for special classes, e.g., convex quadratic objectives [60]. Moreover, we do not distinguish the continuous MCNFP and the integer MCNFP for because of the integrality property of the linear constraints. An optimal solution is always integral as long as the data are integers. For MCCNFP, however, the integer restriction on the optimal solution does make a difference. Continuous MCCNFP and integer MCCNFP need to be handled more carefully.

For continuous MCCNFP, there are in general three approaches.

One possibility for dealing with the convex cost cases is to use differentiable smooth optimization methods. For example, the Lagrangian dual of continuous MCCNFP is an unconstrained convex optimization problem. If the cost function is differentiable and strictly convex, there are many convex optimization methods available, e.g, coordinate descent, conjugate gradient, Newton's method, etc.

The second way is to reduce the problem to a linear cost problem by piece-wise linearization of the convex arc cost functions, see [61]. As a special case, when the cost function is piece-wise linear with integer break-points, the continuous MCCNFP is equivalent to a MCNFP over a network that allows multiple arcs between two nodes.

The third and more general alternative, which applies to general convex cost function, is to extend the methods for MCNFP. For instance, Bertsekas et al. [62] extended the ϵ-relaxation methods to solve MCCNFP in $O(nm \log n \log \frac{\epsilon^0}{\epsilon})$ time where ϵ^0 is the initial accuracy and ϵ is the target accuracy. Scaling methods can also be extended to MCCNFP by solving an integer MCCNFP first. This is done by scaling the data by $\frac{1}{\epsilon}$, find the integer solution to the transformed problem, and then scales the integer solution to a continuous solution to the original problem [45]. In this way, we can generalize any algorithm for integer MCCNFP to solve continuous MCCNFP. We will discuss this approach in the next section.

Now we first review the algorithms for integer MCCNFP. Many polynomial-time algorithms of MCNFP can be extended to integer MCCNFP. In [63] and [64], Minoux extended capacity-scaling algorithm to find an integer optimal solution for minimum convex cost flow problems. The complexity of capacity-scaling algorithm remains the same for MCNFP and MCCNFP, which is $O(m \log U(m+n \log n))$. Goldberg and Tarjan [65] generalized their cost-scaling algorithm to obtain an integer optimal solution of MCCNFP given that all $f_{ij}(\cdot)$ are integers at integer points in $O(nm \log \frac{n^2}{m} \log(nC))$. Another algorithm by Kanzanan and McCormick [66] has complexity $O(mn \log n \log (nc))$ where c is related to the slope of the function near the bounds. Other scaling techniques are also possible. In 1990, Hochbaum and Shanthikumar [67] proposed a scaling-based framework for mixed-integer convex optimization problem with linear constraints and a separable objective. Their framework can be applied to MCCNFP . A direct implementation of the algorithm yields a complexity of $O(\log \frac{U}{4m} MCNF(\text{multigraph with 4nm arcs})))$, where $MCNF(\text{multigraph with 4nm arcs})$ denote the time of solve a MCNFP with $4nm$ arcs. However, this complexity is worse than the algorithms above.

The algorithms for integer MCCNFP can obtain an ϵ-accurate continuous solution via scaling the data. The complexity of solving continuous MCCNFP the complexity of solving integer version with an extra $\frac{1}{\epsilon}$ term on the data. For example, the complexity of capacity scaling is $O(m \log \frac{U}{\epsilon}(m + n \log n))$.

1.6.3 Convex hull algorithms

A convex hull algorithm is an algorithm that finds the convex hull for a given set of points. In this thesis, we focus on the convex hull of a finite set S, which is denoted by

conv(S). Intuitively, the convex hull is what you get by driving a nail into the plane at each point and then wrapping a piece of string around the nails. There are several mathematically equivalent definition of convex hull.

- Given a finite set $S \subset \mathbb{R}^n$, conv(S) is the smallest convex set is the smallest convex set that contains S.
- Given a finite set $S \subset \mathbb{R}^n$, conv(S) is the set of all convex combinations of points in S, i.e.,

$$\text{conv}(S) = \left\{ \sum_{i=1}^{k} a_i x_i \mid k \in \mathbb{N}; \sum_{i=1}^{k} a_i = 1; a_i \geq 0, x_i \in S, i = 1, 2, \cdots, k \right\}.$$

- Given a finite set $S \subset \mathbb{R}^n$, conv(S) is the set of weighted average of all points in the set S:

$$\text{conv}(S) = \left\{ \sum_{i=1}^{|S|} a_i x_i \mid \sum_{i=1}^{|S|} a_i = 1; a_i \geq 0, x_i \in S, i = 1, 2, \cdots, |S| \right\}.$$

In the last definition, we can always reduce the size of S by eliminating the points in S which is a convex combination of other points in S. Therefore, there exists a minimum set $ext(S) \subset S$ such that conv(S) = conv($ext(S)$). We call the points in $ext(S)$ the *extreme points* or *vertices* of S. Geometrically speaking, the convex hull of S is the convex polytope with vertices $ext(S)$.

V-representation and H-representation

By "computing" the convex hull, we mean to find a representation of it. Two popular representation of a convex hull are the vertex representation (V-representation) and half-space representation (H-representation). In a V-representation, the convex hull is described by its extreme points. Finding a V-representation is called the vertex-enumeration problem. In an H-representation, the convex hull is described by a set of half spaces in $\mathbb{R}^n$. Such definition is called a half-space representation. A closed half space in $\mathbb{R}^n$ can be written as a linear inequality. Hence, the H-representation can be written concisely as matrix inequality: $Ax \leq b$ with a $n \times n$ matrix A and $n \times 1$ vector

b. There exist infinitely many H-representations of a convex polytope. However, for a full-dimensional convex polytope, the minimal H-description is in fact unique and is given by the set of the facet-defining halfspaces. Finding a H-representation is called the facet-enumeration problem. There are various algorithms deals with both of the vertex-enumeration problem and the facet-enumeration problem. We mainly focus on vertex enumeration problem and summarize the most well-known algorithms below.

Convex hull algorithm in $\mathbb{R}^2$

For a finite set $S \subset \mathbb{R}^2$, there are many efficient algorithms to find conv(S). Most of them solve both the vertex-enumeration problem and the facet-enumeration problem at the same time. Let n be the cardinality of the input set S and h be the cardinality of the output set $ext(S)$. Table 1.1, we summarize the convex hull algorithm with their running time and algorithmic ideas.

Algorithm	Complexity	Algorithmic idea
Jarviss march (Gift-wrapping)	$O(nh)$	Repeatedly find the vertices that goes in the next slot
Divide and conquer	$O(n^2)$	Divide-and-conquer
Grahams Scan	$O(n \log n)$	Sorts the points and find the vertices by scanning
Incremental Insertion	$O(n \log n)$	Sweeping line
Chans Algorithm (Shattering)	$O(n \log h)$	A combination of divide-and-conquer and gift-wrapping
QuickHull algorithm	$O(nh)$	Divide-and-conquer
Prune and Search	$O(n \log h)$	Improved QuickHull

Table 1.1: The worst-case complexity of convex hull algorithms in $\mathbb{R}^2$

Linear programming methods for constructing convex hull in $\mathbb{R}^n$

Some of those methods above can be extended to find the convex hull in three-dimensional space. However, in n-dimensional space with $n > 3$, the H-representation are much more complex. In $\mathbb{R}^2$, transformation between the two representations seems particular easy. In $\mathbb{R}^n$, , however, finding the corresponding H-representation with

a given V-representation is not a easy tasks. Hence, we have to discuss the vertex-enumeration problem and the facet-enumeration problem separately for $S \subset \mathbb{R}^n$.

Fortunately, the vertex-enumeration problem is still an easy problem in $\mathbb{R}^n$: Given a finite set $S = \{p^1, \ldots, p^l\} \subset \mathbb{R}^n$, checking if a point $p^j \in S$ is an extreme point of $\text{conv}(S)$ can be done by solving the linear program below.

$$v^* = \max_{z, z_0} \ (p^j)^\top z - z_0 \tag{1.3a}$$

$$\text{s.t.} \ (p^i)^\top z - z_0 \leq 0, \ i = 1, \ldots, l, i \neq j \tag{1.3b}$$

$$(p^j)^\top z - z_0 \leq 1. \tag{1.3c}$$

Problem (1.3) is always feasible and bounded. Suppose $((z^*)^\top, z_0^*)$ is an optimal solution and v^* is the corresponding objective value. If $v^* > 0$, then we find a hyperplane $(z^*)^\top x = z_0^*$ that separates p^j and the set $S \backslash \{p^j\}$, and p^j is an extreme point of $\text{conv}(S)$. Otherwise p^j is not an extreme point of $\text{conv}(S)$. Problem (1.3) can be solved by various interior point methods in polynomial time, for example, Karmarkar's algorithm [68]. Hence, we may solve $O(n)$ instances of problem (1.3) to find the set of extreme points $ext(S)$.

Chapter 2

Discrete resource allocation with nested constraints

In this chapter, we study a discrete resource allocation problem with nested constraints. Formally, it is the following discrete optimization problem (P1):

> Given B units of resources and n activities, each of which associated with a convex allocation cost $f_i(\cdot)$, find an allocation of resources to the n activities, denoted by $\boldsymbol{x} \in \mathbb{Z}^n$, to minimize the total allocation cost $\sum_{i=1}^{n} f_i(x_i)$ subject to the total amount of resources constraint as well as lower and upper bound constraints on resources allocated to subsets of activities.

This problem appears as a subproblem in many optimization models employed in production planning, transportation, and data analytics, and has to be solved many times in iterative algorithms for these models, so a fast algorithm to this problem is essential in reducing the overall solution time for those optimization models. We develop a $\Theta(n^2 \log \frac{B}{n})$-time algorithm for the integer version of this problem. Although the theoretical worst-case complexity of our algorithm is worse than the $O(n \log n \log B)$-time algorithm MDA[44], the best-known algorithm in the literature, the performance of our algorithm on all benchmark instances in the literature is competitive to that of MDA. Both our algorithm and MDA are recursive algorithms based on divide and conquer, but for all the benchmark instances the number of recursions our algorithm incurs is significantly fewer than that of MDA. Two distinct features of our algorithm are its simplicity and the adaptive use of infeasibility information of solutions found during the algorithm to guide how division is done.

2.1 Introduction

We refer to (P1) as DRAP-NC to emphasize its discrete nature and its origin from the resource allocation problem. The goal of this problem is to allocate B units of resources to n activities in order to minimize the total allocation cost. For any positive integers $i \leq j$, let $[i : j]$ denote the set $\{i, i+1, \ldots, j\}$. Without loss of generality, we require that the amount of resources allocated to activity i is an integer within the interval $[0, d_i]$ and the allocation cost for activity i is given by a convex function $f_i : [0, d_i] \to \mathbb{R}$, for $i \in [1 : n]$. We also require that the total amount of resources that can be allocated to the first i activities should be within the interval $[a_i, b_i]$, for $i \in [1 : n]$. We assume

that d_i is integral, a_i is integral, b_i is integral or ∞, for $i \in [1:n]$, and $a_n = b_n = B$. Then DRAP-NC can be formulated as the following mixed-integer convex program:

$$\min_{x} \quad \sum_{i=1}^{n} f_i(x_i) \tag{2.1a}$$

$$\text{s.t.} \quad a_i \le \sum_{k=1}^{i} x_k \le b_i, \qquad \forall i \in [1:n-1], \tag{2.1b}$$

$$\sum_{i=1}^{n} x_i = B, \tag{2.1c}$$

$$0 \le x_i \le d_i, x_i \in \mathbb{Z}, \qquad \forall i \in [1:n]. \tag{2.1d}$$

2.2 Literature review

DRAP-NC has its origins in the resource allocation problem (RAP), which is a basic and fundamental problem in engineering. RAP has a wide variety of applications ranging from portfolio selection, production planning, to video-on-demand batching and telecommunications [15, 40, 41]. With a separable convex allocation cost function, RAP falls within the realm of convex optimization, so it can be solved efficiently. If the resource allocated to each activity is required to be integral, then the problem, which we call the discrete resource allocation problem (DRAP), has to be formulated as a mixed-integer convex program, which is NP-hard in general. Due to the special problem structure, however, many variants of DRAP have been shown to be polynomially solvable in the literature. Our interest of studying DRAP-NC stems from the speed optimization problem over a fixed route [12], where the resources allocated to each activity is the time spent between two consecutive customers over the route and the nested constraints correspond to the time-window constraints imposed on serving the customers. Below we first survey results on RAP with continuous variables and then DRAP.

RAP has been studied extensively in the literature. It is a convex optimization problem of which the optimal solution may be irrational and cannot be represented by binary strings. Therefore, we aim to find an ϵ-optimal solution of RAP.

Definition 18. *(ϵ-**optimal solution**) Let $\epsilon > 0$, then x is an ϵ-optimal solution of a continuous optimization problem (P) if there exists some optimal solution $x^\star$ of (P) such that $\|x - x^\star\|_\infty \le \epsilon$.*

In [43], Hochbaum proposed an $O(n \log \frac{B}{\epsilon})$-time algorithm to obtain an ϵ-optimal solution of RAP. If there are additional upper bound constraints on resources that can be consumed by the first i activities for $i \in [1:n]$, then the problem is called the resource allocation problem with linear ascending constraints (RAP-A). Padakandla and Sundaresan [69] proposed a dual

method for the problem with time complexity $O(n^2F)$, where F is the time complexity of solving one RAP with n variables. The result was later improved by Wang [70] with time complexity $O(\max\{n^2 \log n, nF\})$ and Vidal et al. [71] with time complexity $O(n \log n \log \frac{B}{\epsilon})$. There have been growing interests on the resource allocation problems with nested constraints (RAP-NC), which is the continuous relaxation of DRAP-NC. In a recent paper [42], van der Klauw et al. proposed a divide-and-conquer algorithm for RAP-NC. The algorithm runs in $O(n^2)$ time for quadratic cost functions and $O(n^2 \log \frac{B}{\epsilon})$ for general convex cost functions. In [44], Vidal et al. proposed an $O(n \log n \log \frac{B}{\epsilon})$-time divide-and-conquer algorithm for RAP-NC. For the most up-to-date review on the applications and algorithms of RAP, we refer interested readers to [40, 41].

For DRAP, Gross [72] and later Fox [73] derived greedy algorithms to find an optimal solution in $O(B)$ time, which is pseudo-polynomial time. If the function f_i is linear or quadratic for $i \in [1:n]$, then DRAP can be solved in linear time [74, 15]. Hochbaum [43] proposed the first polynomial-time algorithm for DRAP with general cost functions—a scaling-based algorithm with a greedy algorithm as a subroutine; the algorithm has a running time $O(n \log \frac{B}{n})$. The discrete version of RAP-A, which we call DRAP-A, can also be solved by Hochbaum's algorithm in $O(n \log n \log \frac{B}{n})$ time. For a more comprehensive review on RAP-A and DRAP-A, we refer the readers to a recent survey [75]. There are very few results on DRAP-NC. van der Klauw et al. studied a problem more general than DRAP-NC by replacing the integrality constraints (2.1d) with constraints $x_i \in S_i$, where S_i is an arbitrary discrete set, for each $i \in [1:n]$. They proposed an $O(B \log n)$-time algorithm, which is pseudo-polynomial time. In the recent paper [44], Vidal et al. showed that the framework of their algorithm for RAP-NC can also be modified to work on DRAP-NC, and the corresponding algorithm runs in $O(n \log n \log B)$ time. However, no numerical results are reported on its performance on instances of DRAP-NC.

Our contribution

In this chapter, we develop a $\Theta(n^2 \log \frac{B}{n})$-time algorithm for DRAP-NC. Our algorithm essentially combines the divide-and-conquer framework for RAP-NC in [42] with the scaling-based algorithm for DRAP in [43]. It is an infeasibility-guided divide-and-conquer algorithm. The worst-case time complexity of our algorithm is worse than the best time complexity in the literature, $O(n \log n \log B)$ achieved by MDA [44]. For a specific instance, however, the number of subproblems solved by our algorithm may be significantly fewer than MDA, a pure divided-and-conquer algorithm that always divides the problems into subproblems of equal size. The worst-case complexity is usually not reached. Finally, to the best of our knowledge, there are no numerical experiments on DRAP-NC in the literature. We conduct extensive numerical experiments to evaluate the performance of our algorithm on test instances with five classes of convex

objectives. Our algorithm significantly outperforms a state-of-the-art mixed-integer linear and quadratic optimization solver–Gurobi [16]. Moreover, our algorithm solves substantially fewer subproblems then MDA and the performance is competitive compared to MDA on benchmark instances in the literature.

The remainder of this section is organized as follows. Section 2.3 describes our algorithm for DRAP-NC and its correctness is proven in Section 2.4. We present extensive numerical results on DRAP-NC instances with a variety of convex objectives in Section 2.5.

2.3 A recursive algorithm based on divide and conquer

In this section, we present an infeasibility-guided divide-and-conquer algorithm (DCA) to solve DRAP-NC. The main idea is to solve a relaxation of DRAP-NC by dropping the nested constraints and divide the problem into two subproblems according to the obtained optimal solution. For the ease of exposition, we introduce two dummy parameters a_0, b_0, and set $a_0 = b_0 = 0$. We define the problem DRAP-NC(s, e) below to be DRAP-NC with respect to activity $s, s+1, \ldots, e$ for $s, e \in [1 : n]$.

$$\min \quad \sum_{i=s}^{e} f_i(x_i) \tag{2.2a}$$

$$\text{s.t.} \quad a_i - a_{s-1} \le \sum_{k=s}^{i} x_k \le b_i - a_{s-1}, \qquad \forall i \in [s : e], \tag{2.2b}$$

$$\sum_{k=s}^{e} x_k = b_e - a_{s-1}, \tag{2.2c}$$

$$0 \le x_i \le d_i, x_i \in \mathbb{Z}, \qquad \forall i \in [s : e], \tag{2.2d}$$

where we assume that $a_{s-1} = b_{s-1}$ and $a_e = b_e$. Moreover, we define the problem DRAP(s, e) to be a relaxation of DRAP-NC(s, e), by dropping the nested constraints (2.2b).

$$\min \quad \sum_{i=s}^{e} f_i(x_i) \tag{2.3a}$$

$$\text{s.t.} \quad \sum_{k=s}^{e} x_k = b_e - a_{s-1}, \tag{2.3b}$$

$$0 \le x_i \le d_i, x_i \in \mathbb{Z}, \qquad \forall i \in [s : e]. \tag{2.3c}$$

To solve DRAP-NC is equivalent to solve DRAP-NC$(1, n)$. We sketch the idea of solving DRAP-NC(s, e) for any $s, e \in [1 : n]$ as follows: We first solve the relaxation DRAP(s, e) to

optimality. If the resulting optimal solution satisfies the nested constraints (2.2b), then we stop and output it as the optimal solution of DRAP-NC(s, e). Otherwise, we find the index K of the most violated nested constraint and fix the total resources consumed by the first K activities to be a_K or b_K, depending on which side of the nested constraints is violated. In particular, let $\bar{x}$ be an optimal solution of DRAP(s, e). If $\sum_{k=s}^{K} \bar{x}_k > b_e - a_{s-1}$, then we set $\sum_{k=s}^{K} x_k = b_e - a_{s-1}$; otherwise we have $\sum_{k=s}^{K} \bar{x}_k < a_e - a_{s-1}$, and we set $\sum_{k=s}^{K} x_k = a_e - a_{s-1}$. With the above adjustment, we divide DRAP-NC(s, e) into two subproblems DRAP-NC(s, K) and DRAP-NC$(K+1, e)$, and solve each subproblem. Finally, we combine the optimal solutions of DRAP-NC(s, K) and DRAP-NC$(K+1, e)$ to obtain an optimal solution of DRAP-NC(s, e). Note that each of the subproblems DRAP-NC(s, K) and DRAP-NC$(K+1, e)$ is solved to optimality following the above procedure recursively. The details of the algorithm is described in Algorithm 1.

Algorithm 1 DCA: A recursive algorithm for DRAP-NC(s, e)

Input: Nested bounds a_i and b_i for $i \in [s-1 : e]$ with $a_{s-1} = b_{s-1}$ and $a_e = b_e$; variable upper bound d_i and function oracle f_i, for $i \in [s : e]$.
Output: An optimal solution $(\hat{x}_s, \ldots, \hat{x}_e)$ of DRAP-NC(s, e).
function DRAP-NC(s, e)
    if $e = s$ **then return** $x_s = a_s - a_{s-1}$
    end if
    Solve DRAP(s, e) and obtain the optimal solution: $(\bar{x}_s, \ldots, \bar{x}_e)$
    if $(\bar{x}_s, \ldots, \bar{x}_e)$ satisfies the nested bound constraints **then return** $(\bar{x}_s, \ldots, \bar{x}_e)$
    end if
    Find index K of the most violated nested constraint. Break ties by choosing the maximum index among all indices of most violated nested constraints.
    if $\sum_{k=s}^{K} \bar{x}_k > b_K - a_{s-1}$ **then** $a_K \leftarrow b_K$ ▷ Update bounds for subproblems
    else $b_K \leftarrow a_K$
    end if
    $(\hat{x}_s, \ldots, \hat{x}_K) \leftarrow$ DRAP-NC(s, K)
    $(\hat{x}_{K+1}, \ldots, \hat{x}_e) \leftarrow$ DRAP-NC$(K+1, e)$
    return $(\hat{x}_s, \hat{x}_{s+1}, \ldots, \hat{x}_{e-1}, \hat{x}_e)$
end function

We state our main result below.

Theorem 1. *Algorithm 1 solves DRAP-NC correctly in* $\Theta(n^2 \log \frac{B}{n})$ *time.*

The correctness of Algorithm 1 follows directly from the Proposition 1 below.

Proposition 1. *Suppose* f_i *is convex for* $i \in [s:e]$ *with* $s, e \in [1:n]$. *Let* $(\bar{x}_s, \bar{x}_{s+1}, \cdots, \bar{x}_e)$ *be an optimal solution of DRAP*(s,e). *Suppose* $(\bar{x}_s, \bar{x}_{s+1}, \cdots, \bar{x}_e)$ *violates at least one nested constraint of DRAP-NC*(s,e) *and the index of the most violated nested constraint is* K. *Then there exists an optimal solution* $(\hat{x}_s, \hat{x}_{s+1}, \cdots, \hat{x}_e)$ *of DRAP-NC*(s,e) *such that* $\sum_{k=s}^{K} \hat{x}_k = a_K - a_{s-1}$ *if* $\sum_{i=s}^{K} \bar{x}_i < a_K - a_{s-1}$ *or* $\sum_{k=s}^{K} \hat{x}_k = b_K - a_{s-1}$ *if* $\sum_{i=s}^{K} \bar{x}_i > b_K - a_{s-1}$.

We postpone the proof of Proposition 1 to Section 2.4, and explain first the detail of Step 6 in Algorithm 1: how to solve the relaxation problem DRAP(s,e) to optimality.

2.3.1 DRAP(s,e) in Step 6 of Algorithm 1

We use Hochbaum's scaling-based algorithm [43] to solve DRAP(s,e). Hochbaum's algorithm makes multiple calls to a subroutine that returns a solution that uses as many resources as possible. The algorithm runs in $O(n \log \frac{B}{n})$ time for n activities and B units of total resources. For the completeness of our algorithm, we present Hochbaum's algorithm for DRAP with our notations in Algorithm 2 and Algorithm 3 below. Let $\boldsymbol{e}$ be a column vector of all one's, $\mathbf{0}$ be a column vector of all zeros, and $\boldsymbol{e}^i$ be a unit vector with the i-th row being one.

Algorithm 2 Algorithm for DRAP(s,e) [43]

Input: An instance of DRAP(s,e) with variable upper bound d_i and value oracle f_i for $i \in [s:e]$, and total amount of resources $B = b_e - a_{s-1}$.
Output: An optimal solution $\boldsymbol{x}^\star$ of DRAP(s,e).
function DRAP(s,e)
    Initialization: $\delta \leftarrow \lceil \frac{B}{2n} \rceil$, $\boldsymbol{x} \leftarrow \mathbf{0}$.
    while $\delta > 1$ **do**
        $\boldsymbol{x} \leftarrow$ Greedy$(\delta, \boldsymbol{x}, B)$
        $\boldsymbol{x} \leftarrow \max\{\boldsymbol{x} - \delta\boldsymbol{e}, \mathbf{0}\}$, $\delta \leftarrow \lceil \frac{\delta}{2} \rceil$
    end while
    $\boldsymbol{x}^\star \leftarrow$ Greedy$(1, \boldsymbol{x}, B)$
    return $\boldsymbol{x}^\star$
end function

Algorithm 3 The greedy subroutine Greedy($\delta, \boldsymbol{x}, B$) [43]

Input: Integer step size $\delta \geq 1$; a vector $\boldsymbol{y}$ that satisfies the variable bound constraints of DRAP(s, e), i.e., $y_i \in [0, d_i]$ for $i \in [s : e]$; the total amount of available resources $B = b_e - a_{s-1}$; the value oracle f_i for $i \in [s : e]$.
Output: A solution $\boldsymbol{x}$ with the property that $x_i \in [0, d_i]$ for $i \in [s : e]$ and $\sum_{i=s}^{e} x_i \geq \sum_{i=s}^{e} y_i$.
function GREEDY($\delta, \boldsymbol{y}, B$)
    Initialization: $\boldsymbol{x} \leftarrow \boldsymbol{y}$, $R \leftarrow B - \boldsymbol{y}^\top \boldsymbol{e}$, $E \leftarrow \{s, s+1, \cdots, e\}$.
    while $R > 0$ and $E \neq \emptyset$, **do**
        Find i such that $\Delta_i(x_i) = \min_{j \in E}\{\Delta_j(x_j)\}$ where $\Delta_j(x_j) = f_j(x_j + 1) - f_j(x_j)$. ▷ Find out the greedy step
        if $x_i + 1 < d_i$ **then** $E \leftarrow E \setminus \{i\}$ ▷ Check if the variable bound constraint is violated
        else if $x_i + \delta > d_i$ or $\delta > R$ **then**
            $\delta' = \min\{R, d_i - x_i\}$, $x_i \leftarrow x_i + \delta'$, $R \leftarrow R - \delta'$
            $E \leftarrow E \setminus \{i\}$
        else $x_i \leftarrow x_i + \delta$, $R \leftarrow R - \delta$
        end if
    end while
    if $\delta > 1$ or $R = 0$ **then return** $\boldsymbol{x}$
    else
        return The instance DRAP(s, e) is infeasible.
    end if
end function

2.3.2 Time complexity of Algorithm 1

The computational complexity of Algorithm 1 is given by th.

Proposition 2. *The time complexity of Algorithm 1 is* $\Theta(n^2 \log \frac{B}{n})$.

Proof. By Proposition 1, Algorithm 1 correctly solves DRAP-NC.

For each subproblem DRAP-NC(s, e), it takes $O((e - s + 1) \log \frac{B}{e-s+1})$ time to solve the relaxation DRAP(s, e) by Hochbaum's algorithm. In addition, it takes $O(e - s + 1)$ operations to find the most violated nested constraint given an optimal solution to DRAP(s, e). Since each

recursion of Algorithm 1 fixes one nested constraint at equality, we have at most $n-1$ nested constraints to fix at equality. The depth of the recursion tree is bounded by n. At the same level of the recursion tree, the total size of the subproblems is n. Therefore, the complexity of Algorithm 1 is $O(n^2 \log \frac{B}{n})$.

Next, we show that by a constructed example that the time complexity $O(n^2 \log \frac{B}{n})$ is tight. The depth of the recursion tree could be as deep as n. The example is proposed by Thibaut Vidal [76]. Consider the following instance of DRAP-NC:

$$\begin{aligned}
\min \quad & \sum_{k=1}^{n} x_k^2 && && \text{(2.4a)}\\
\text{s.t.} \quad & \sum_{k=1}^{i} x_k = (-1)^i i, && \forall i \in [1:n-1], && \text{(2.4b)}\\
& \sum_{k=1}^{n} x_k = (-1)^n n, && && \text{(2.4c)}\\
& -2n \le x_i \le 2n, x_i \in \mathbb{Z}, && \forall i \in [1:n], && \text{(2.4d)}
\end{aligned}$$

In this example, the nested upper and lower bounds of the same index are the same. The instance (2.4) can be transformed into the Formulation (2.1) with $B = 2n^2 + (-1)^n n$ by variable substitution $x_i = x_i' - 2n, i \in [1:n]$. It has a unique optimal solution $\boldsymbol{x} = (-1, 3, -5, \cdots, (-1)^n(2n-1))^\top$.

Consider applying Algorithm 1 to instance (2.4). In the first iteration, the first $n-1$ nested constraints (2.4b) are relaxed and we obtain DRAP$(1, n)$:

$$\begin{aligned}
\min \quad & \sum_{k=1}^{n} x_k^2 && && \text{(2.5a)}\\
\text{s.t.} \quad & \sum_{k=1}^{n} x_k = (-1)^n n, && && \text{(2.5b)}\\
& -2n \le x_i \le 2n, x_i \in \mathbb{Z}, && \forall i \in [1:n], && \text{(2.5c)}
\end{aligned}$$

The optimal solution to DRAP$(1, n)$ is $\boldsymbol{x} = ((-1)^n, (-1)^n, \cdots, (-1)^n)^\top$. The index of the most violated nested constraint is $n-1$. Therefore, we divide the problem into DRAP-NC$(1, n-1)$ and DRAP-NC(n, n).

1. DRAP-NC(n, n) is as follows

$$\begin{aligned} \min \quad & x_n^2 && (2.6a)\\ \text{s.t.} \quad & x_n = (-1)^n(2n-1), && (2.6b)\\ & -2n \le x_n \le 2n, x_n \in \mathbb{Z}. && (2.6c) \end{aligned}$$

   The optimal solution is $x_n = (-1)^n(2n-1)$.

2. DRAP-NC$(1, n-1)$ is as follows

$$\begin{aligned} \min \quad & \sum_{i=1}^{n-1} x_i^2 && && (2.7a)\\ \text{s.t.} \quad & \sum_{k=1}^{i} x_k = (-1)^i i, && \forall i \in [1:n-2], && (2.7b)\\ & \sum_{k=1}^{n-1} x_k = (-1)^{n-1}(n-1), && && (2.7c)\\ & -2n \le x_i \le 2n, x_i \in \mathbb{Z}, && \forall i \in [1:n-1]. && (2.7d) \end{aligned}$$

   It has the same structure as DRAP-NC$(1, n)$ with one less variable. It is solved recursively. By a similar discussion as above, the algorithm will divide DRAP-NC$(1, n-1)$ into DRAP-NC$(1, n-2)$ and DRAP-NC$(n-1, n-1)$.

Hence, we can fix only one variable at each level of the recursion tree. The depth of the recursion tree is n. For the instance (2.4), the running time is $\Omega(n^2 \log \frac{B}{n})$ with $B = 2n^2 + (-1)^n n$. The complexity of Algorithm 1 is $\Theta(n^2 \log \frac{B}{n})$. □

2.4 Proof of Proposition 1

In this section, we provide a complete proof of Proposition 1. The proof is inspired by the proof of the correctness of the algorithm for RAP-NC in [42]. In [42], the authors proved a proposition very similar to Proposition 1 for the continuous counterpart of DRAP-NC. In particular, they showed that Proposition 1 holds if the problems DRAP(s, e) and DRAP-NC(s, e) are replaced with their continuous relaxations RAP(s, e) and RAP-NC(s, e), respectively.

We first introduce an auxiliary function that connects the discrete and continuous versions of the resource allocation problems. Given a convex function $f(x)$, we define a piecewise linear

function as follows:

$$f^{PL}(x) = f(\lfloor x \rfloor) + (x - \lfloor x \rfloor) \times (f(\lceil x \rceil) - f(\lfloor x \rfloor)). \tag{2.8}$$

It can be easily verified that $f^{PL}(x)$ has the same values as $f(x)$ at integer points. Consider any instance of DRAP-NC(s, e). We define DRAP-NC$^{PL}(s, e)$ to be a problem by replacing the function f_i in DRAP-NC(s, e) with function f_i^{PL} for $i \in [s : e]$. Similarly, we define RAP-NC$^{PL}(s, e)$, DRAP$^{PL}(s, e)$, and RAP$^{PL}(s, e)$ to be the continuous relaxation of DRAP-NC$^{PL}(s, e)$, DRAP-NC$^{PL}(s, e)$ without nested constraints, the continuous relaxation of DRAP-NC$^{PL}(s, e)$ without nested constraints, respectively.

The proof of Proposition 1 can be outlined as follows: (To simplify the notation, we omit the parameters (s, e) in all related problems below in the rest of this section.)

1. We prove that an optimal solution of DRAP-NC is an optimal solution of DRAP-NC^{PL} and vice versa, and an optimal solution of DRAP is an optimal solution of DRAP^{PL} and vice versa.

2. We prove that an optimal solution of DRAP-NC^{PL} is also an optimal solution of RAP-NC^{PL}, and as a result, an optimal solution of DRAP^{PL} is also an optimal solution of RAP^{PL}.

3. We prove that Proposition 1 holds, with an idea inspired by Lemma 2 in [42] for the continuous counterparts RAP-NC and RAP.

The first step is easy to show. Since $f_i(x_i)$ and $f_i^{PL}(x_i)$ coincide in the integer domain for each i, each feasible solution of DRAP-NC is a feasible solution of DRAP-NC^{PL} with the same objective value, and vice versa. Therefore, any optimal solution of DRAP-NC is an optimal solution of DRAP-NC^{PL} and vice versa. The second statement of the first step follows directly from the fact that RAP^{PL} and DRAP^{PL} are special cases of RAP-NC^{PL} and DRAP-NC^{PL} respectively by setting $a_i = -\infty$ and $b_i = \infty$ for each i.

The second step follows from Theorem 4 in [44] and is not difficult to prove. We state the result below.

Proposition 3. *[44, Theorem 4] Any optimal solution $\boldsymbol{x}^\star$ of DRAP-NC$^{PL}(s, e)$ is also an optimal solution of the corresponding RAP-NC$^{PL}(s, e)$ without integrality constraints.*

Finally, we give a proof of Proposition 1. The proof idea is inspired by Lemma 2 in [42], which makes use of the KKT conditions for continuous problems RAP-NC and RAP.

Proof of Proposition 1. We will prove that there exists an optimal solution $(\hat{x}_s, \ldots, \hat{x}_e)$ of DRAP-NC(s, e) satisfying $\sum_{k=s}^{K} \hat{x}_k = b_K - a_{s-1}$ if $\sum_{k=s}^{K} \bar{x}_k > b_K - a_{s-1}$. The result for

the other case in which $\sum_{k=s}^{K} \bar{x}_k < a_K - a_{s-1}$ can be proven analogously.

We prove the result by contradiction. Suppose that there does not exist any optimal solution of DRAP-NC(s,e) such that the constraint $\sum_{k=s}^{K} x_k \le b_K - a_{s-1}$ is satisfied at equality. Since any instance of DRAP-NC(s,e) has only a finite number of optimal solutions, we select the optimal solution that maximizes $\sum_{k=s}^{K} x_k$ among all optimal solutions. Call this solution $\hat{\boldsymbol{x}} = (\hat{x}_s, \ldots, \hat{x}_e)$. Assume that $\sum_{k=s}^{K} \hat{x}_k < b_K - a_{s-1}$. Let $I = \arg\max\{i \mid \sum_{k=s}^{i} \hat{x}_k = b_i - a_{s-1}, s - 1 \le i < K\}$ and $J = \arg\min\{i \mid \sum_{k=s}^{i} \hat{x}_k = b_i - a_{s-1}, K < i \le e\}$.

We claim that there exist integer $p \in [I+1:K]$ and $q \in [K+1:J]$ such that $\bar{x}_p > \hat{x}_p$ and $\bar{x}_q < \hat{x}_q$. To see this, since the index of the most violated nested constraint is K, $\sum_{k=s}^{K} \bar{x}_k - b_K \ge \sum_{k=s}^{I} \bar{x}_k - b_I$. Then $\sum_{k=s}^{K} \bar{x}_k - \sum_{k=s}^{I} \bar{x}_k \ge b_K - b_I$. Meanwhile, $\sum_{k=s}^{K} \hat{x}_k < b_K - a_{s-1}$ and $\sum_{k=s}^{I} \hat{x}_k = b_I - a_{s-1}$. Thus $\sum_{k=s}^{K} \bar{x}_k - \sum_{k=s}^{I} \bar{x}_k \ge b_K - b_I > (\sum_{k=s}^{K} \hat{x}_k + a_{s-1}) - b_I = \sum_{k=s}^{K} \hat{x}_k - \sum_{k=s}^{I} \hat{x}_k$. Therefore $\sum_{k=I+1}^{K} \bar{x}_k > \sum_{k=I+1}^{K} \hat{x}_k$. Then there must exist some $p \in [I+1:K]$ such that $\bar{x}_p > \hat{x}_p$. Similarly, we can find some $q \in [K+1:J]$ such that $\bar{x}_q < \hat{x}_q$.

The solution $\bar{\boldsymbol{x}}$ is an optimal solution of DRAP(s,e), so it is also an optimal solution of RAP$^{PL}(s,e)$. In addition, $\bar{x}_p > \hat{x}_p \ge 0$ and $\bar{x}_q < \hat{x}_q \le d_q$. Therefore, the solution $\bar{\boldsymbol{x}}$ should satisfy the KKT conditions for RAP$^{PL}(s,e)$ (see Chapter 28-30 of [77] for KKT conditions with subdifferentials). There must exist some dual variable λ such that

$$\inf \partial f_p^{PL}(\bar{x}_p) \le \lambda \le \sup \partial f_q^{PL}(\bar{x}_q),$$

where $\partial f_i^{PL}(x)$ is the subdifferential of $f_i^{PL}(x)$ for $i \in [s:e]$. Then by the monotonicity of the subdifferential of convex functions, we have

$$\sup \partial f_p^{PL}(\hat{x}_p) \le \inf \partial f_p^{PL}(\bar{x}_p) \le \sup \partial f_q^{PL}(\bar{x}_q) \le \inf \partial f_q^{PL}(\hat{x}_q). \tag{2.9}$$

We create an integer vector $\tilde{\boldsymbol{x}} = (\tilde{x}_s, \ldots, \tilde{x}_e)$ such that $\tilde{x}_p = \hat{x}_p + 1$, $\tilde{x}_q = \hat{x}_q - 1$, and $\tilde{x}_i = \hat{x}_i$ otherwise. We claim that $\tilde{\boldsymbol{x}}$ is also an optimal solution of DRAP-NC(s,e). To check the feasibility of $\tilde{\boldsymbol{x}}$, note that $\hat{x}_p < \bar{x}_p \le d_p$ and $\hat{x}_p$ and $\bar{x}_p$ are integers, so $\tilde{x}_p \le d_p$. Similarly, we can show that $\tilde{x}_q \ge 0$. To check that $\tilde{x}$ satisfies all nested constraints, note that by the choice of p and q, for each $j \in [p:q]$, $\sum_{k=s}^{j} \hat{x}_k < b_j - a_{s-1}$. Thus for each $j \in [p:q]$, $\sum_{k=s}^{j} \tilde{x}_k \le b_j - a_{s-1}$. To check the optimality of $\tilde{\boldsymbol{x}}$, note that f_p^{PL} and f_q^{PL} are both piecewise linear functions with break points only at integer points. Then from 2.9 we have $f_p^{PL}(\tilde{x}_p) + f_q^{PL}(\tilde{x}_q) \le f_p^{PL}(\hat{x}_p) + f_q^{PL}(\hat{x}_q)$. But $\sum_{k=s}^{K} \tilde{x}_k = \sum_{k=s}^{K} \hat{x}_k + 1$, contradicting to the choice of $\hat{\boldsymbol{x}}$ that it maximizes $\sum_{k=s}^{K} x_k$. $\square$

2.5 Numerical experiments

In this section, we evaluate the performance of DCA under three different settings. Firstly, we test the performance of DCA on DRAP-NC instances with linear and quadratic costs and compare that with the state-of-the-art solver Gurobi [16]. Note that our algorithm works for any convex function and only requires access to value oracles of the cost functions. We choose linear and quadratic cost functions only because Gurobi is handle mixed-integer linear and quadratic optimization problems. According to the test result, DCA is on average 6 times faster on DRAP-NC with linear objectives and five to six orders of magnitude faster on DRAP-NC with quadratic objectives than Gurobi. In the second and the third part, we evaluate the performance of DCA on instances with separable convex objectives with MDA, the algorithm with the best worst-case time complexity of $O(n \log n \log B)$ in the literature [44]. In the second part, the test instances are DRAP-NC instances with three classes of separable general convex objectives. Lastly, we evaluate the performance when DCA and MDA are used to solve the projection subproblems in an application of RAP-NC with non-separable objectives. According to the test results, the averaged solution time of DCA is at least 50% less than that of MDA.

The parameters of the test instances of DRAP-NC are generated in the following way.

- We introduce a parameter V_b to be the maximum range of any variable in DRAP-NC. The variable upper bound d_i's are drawn from a discrete uniform distribution over the set $\{1, 2, \cdots, V_b\}$.
- The bounds of the nested constraints a_i's and b_i's are generated following the procedure introduced in [44]. In particular, we first introduce the following two sequences of random numbers $\{v_i\}$ and $\{w_i\}$.

$$\begin{aligned} v_0 = w_0 = 0, \\ v_i = v_{i-1} + X_i^v, \\ w_i = w_{i-1} + X_i^w, \end{aligned}$$

  where X_i^w and X_i^v are drawn independently from a uniform distribution over the interval $[0, d_i]$. Finally, we set $a_i = \min\{v_i, w_i\}$ and $b_i = \max\{v_i, w_i\}$ for $i = 1, 2, \cdots, n$. This procedure guarantees the feasibility of each generated instance.

For each pair of (n, V_b) values, 10 instances are generated. In addition, when the solution time is less than 1 millisecond, we solve each instance 100 times and report the average running time. The time limit is set to be 1200s for each instance. All programs are coded in Java and the experiment is conducted on a desktop with i7-8700k CPU, 32G memory, an Ubuntu Linux system, and Gurobi 8.0.1. We also do a sanity check and compare the solutions obtained

by DCA, Gurobi, and MDA for all test instances that are solved within the time limit. The solutions turn out to be the same for every instance in the test set.

2.5.1 Computational experiment with Gurobi

Linear costs

In each instance with linear costs, the function $f_i(x)$ has the form $f_i(x) = p_i x$, where p_i is drawn from a uniform distribution over $[-1, 1]$. The parameter V_b is set to be 10 or 100. We test both our algorithm and Gurobi on 24 different pairs of (n, V_b) values. Note that it is not difficult to check that the constraint matrix of DRAP-NC is totally unimodular. Thus DRAP-NC with linear costs can be solved as a linear program by dropping the integrality constraints on the variables. To speed up the performance of Gurobi, we introduce the new variables $y_i = \sum_{k=1}^{i} x_i$ and reformulate the DRAP-NC problem as follows:

$$\min \quad \sum_{i=1}^{n} f_i(x_i) \tag{2.10a}$$

$$\text{s.t.} \quad y_1 = x_1 - a_1, \tag{2.10b}$$

$$y_i = y_{i-1} + x_i - (a_i - a_{i-1}), \qquad \forall i \in [2 : n-1], \tag{2.10c}$$

$$0 = x_{n-1} + x_n - (a_n - a_{n-1}), \tag{2.10d}$$

$$0 \le y_i \le b_i - a_i, \qquad \forall i \in [1 : n-1], \tag{2.10e}$$

$$0 \le x_i \le d_i, \qquad \forall i \in [1 : n]. \tag{2.10f}$$

With such equivalent formulation with a sparse coefficient matrix, Gurobi is able to solve the problem thirty to one hundred times faster than when it uses the dense formulation (2.1). We summarize the performance of two algorithms in Table 2.1 and Figure 2.1. The running time is averaged over 10 instances for each set of parameters.

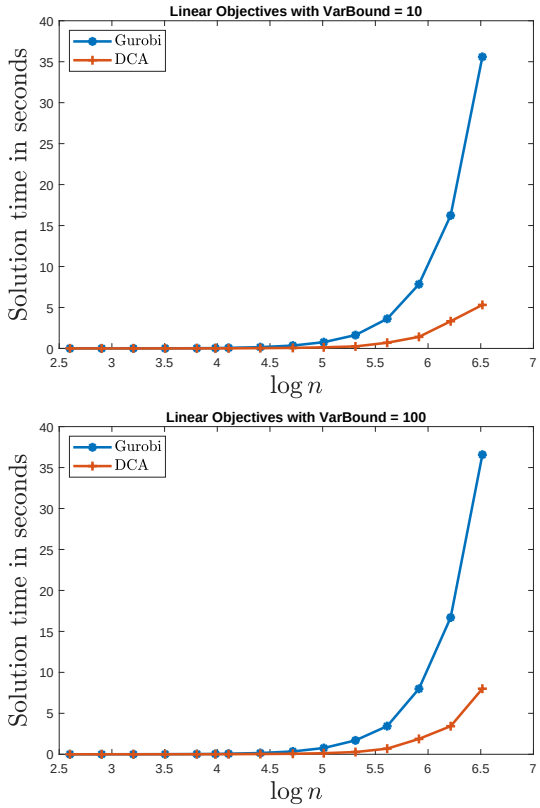


Figure 2.1: Solution time of DCA and Gurobi for instances with linear costs.

Parameters		DCA	Gurobi	Parameters		DCA	Gurobi
n	V_b	Time (s)	Time (s)	n	V_b	Time (s)	Time (s)
3200	10	0.003	0.012	3200	100	0.003	0.012
6400	10	0.006	0.027	6400	100	0.006	0.026
9600	10	0.008	0.040	9600	100	0.010	0.040
12800	10	0.013	0.069	12800	100	0.014	0.069
25600	10	0.034	0.157	25600	100	0.032	0.155
52100	10	0.065	0.350	52100	100	0.061	0.348
102400	10	0.136	0.762	102400	100	0.132	0.762
204800	10	0.245	1.629	204800	100	0.273	1.697
409600	10	0.693	3.619	409600	100	0.687	3.437
819200	10	1.411	7.834	819200	100	1.877	7.994
1638400	10	3.318	16.232	1638400	100	3.428	16.695
3276800	10	5.313	35.591	3276800	100	8.005	36.572

Table 2.1: Solution statistics of DCA and Gurobi for instance with linear costs.

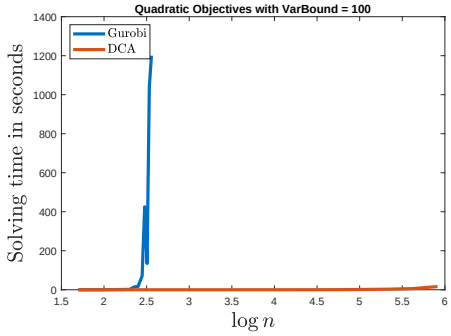
It can be seen that both algorithms can solve large-sized instances of DRAP-NC. The linear

programming solver of Gurobi is already very efficient in solving instances with more than three million variables, but DCA is on average 6 times faster.

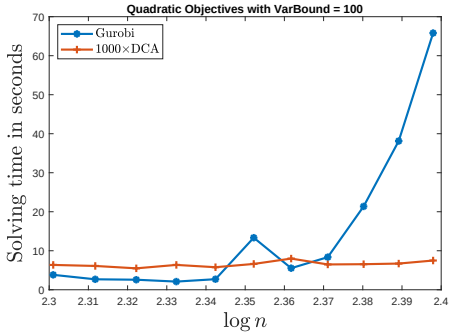
Quadratic costs

In each instance with quadratic costs, the function $f_i(x)$ has the form $f_i(x) = p_i x^2 + q_i x$, where p_i is drawn from a uniform distribution over $[0, 1]$ and q_i is drawn from a uniform distribution over $[-1, 1]$. The parameter V_b is set to be 10 or 100. We generate 20 pairs of (n, V_b) values and 10 instances for each pair of (n, V_b) value.

We summarize the performance of the two algorithms in Table 2.2 and Figure 2.2 below. All statistics are obtained by averaging over 10 instances. In Table 2.2, the columns `OptGap(%)`, `RootGap(%)`, and `HGap(%)` give the integrality gap when Gurobi reaches the time limit, the integrality gap at the root node of the branch-and-bound tree, and the integrality gap at the root node after Gurobi applies some heuristics, respectively. We set in Gurobi the tolerance for the optimality gap to be 10^{-4}, which means that Gurobi will report that an optimal solution is found if the relative integrality gap falls below 10^{-4}.



(a) Solution time of DCA and Gurobi for instances with quadratic costs



(b) Detailed solution time of DCA and Gurobi for instances with quadratic costs and 200 to 250 variables. (The solution time of DCA is multiplied by 1000.)

Figure 2.2: Solution time for DCA and Gurobi for instances with quadratic costs. The largest instance solved to optimality by DCA has 819,200 variables.

Parameters		DCA	Gurobi				
n	V_b	Time(s)	Time(s)	Explored Nodes	OptGap(%)	RootGap(%)	HGap(%)
50	10	0.000156	0.0166	31.1	0.0045	20.4994	0.075
100	10	0.000268	0.1163	2062.5	0.0053	20.626	0.0978
200	10	0.000601	10.7616	571590.4	0.0096	23.259	0.0992
230	10	0.000775	68.7861	3999867.2	0.0098	22.9281	2.7767
250	10	0.000928	227.8887	10598736.5	0.0112	25.1479	0.0989
280	10	0.000855	442.2539	20844989	0.013	25.424	0.0984
300	10	0.001113	634.9682	25889488	0.0157	27.349	0.1023
320	10	0.001165	1105.4133	44173760.3	0.0217	25.182	0.1132
340	10	0.001113	1004.0611	37761076.5	0.0241	27.94	0.1065
360	10	0.001132	> 1200	32546604.2	0.0166	28.378	0.0918
50	100	0.000123	0.0157	50.6	0.0018	21.0764	0.102
100	100	0.000248	0.0859	1429.2	0.0069	23.3402	0.0984
200	100	0.000540	6.1296	248931.3	0.0098	25.222	2.2209
230	100	0.000687	20.8305	1071874.9	0.01	25.4477	0.1045
250	100	0.000725	155.6103	8529738.3	0.0103	25.2047	0.0998
280	100	0.000869	254.595	11572698.8	0.0113	26.565	0.1005
300	100	0.000859	759.4357	34022833.6	0.0179	26.841	0.1032
320	100	0.000986	285.6427	12557311.3	0.0107	26.2289	0.0933
340	100	0.001104	1076.565	39186584.9	0.0277	26.6789	0.1086
360	100	0.001041	> 1200	39181356	0.039	25.96	0.1138

Table 2.2: Solution statistics for DCA and Gurobi with quadratic cost.

It can be seen that Gurobi is only able to solve instances with less than 360 variables within the 1200 second time limit. On the other hand, DCA can solve instances with $819,200$ variables in 30 seconds. The largest instance that can be solved by DCA is far beyond the size of instances that can be handled by Gurobi. The root gap of the mixed-integer convex programming formulation (2.1) is large (over 20%). Although Gurobi has very powerful heuristic methods, it still requires a lot of time to prove that the obtained solution is optimal. In addition, the running time of Gurobi increases significantly as the size of instance increases, accompanied by a steep increase in the number of nodes explored in the branch-and-bound tree. On the other hand, our algorithm DCA is five to six orders of magnitude faster than Gurobi, solving instances for instances with no more than 360 variables in around 10^{-3}s.

2.5.2 Computational experiment on convex objectives

In the next, we test the performance of DCA on DRAP-NC instances with three classes of convex objectives and compare it with MDA [44], the best algorithm in the literature. MDA

has a time complexity of $O(n \log n \log B)$, which is strictly better than the time complexity $\Theta(n^2 \log \frac{B}{n})$ of our algorithm. Nevertheless, the computational experiment indicates that the DCA is competitive when compared to MDA. The rest of this section is organized as follows: Firstly, we give a brief introduction to MDA and discuss possible advantages of DCA and MDA; Then we present examples on which DCA is superior to MDA as it solves much fewer subproblems; At last, we conduct a numerical experiment to test the performance of DCA and MDA on three classes of convex objectives. We report the average solution time and average number of subproblems solved. These statistics indicate that DCA has a better performance than MDA on the test instances.

Introdution to MDA

MDA is currently the most efficient algorithm for DRAP-NC in the literature as it achieves a worst-case time complexity of $O(n \log n \log B)$ [44]. It is a divide-and-conquer algorithm. It breaks down an instance of DRAP-NC with n variables into two subproblems of equal size, i.e, two subproblems with $\lceil \frac{n}{2} \rceil$ variables. The subproblems are solved recursively, with the subproblem with exactly one variable solved trivially. Based on the solutions returned by the two subproblems, MDA generates stronger variable bounds that dominates the original nested bound constraints. With these stronger variable bounds, the original problem is essentially reduced to four instances of DRAP and hence can be solved by any algorithm for DRAP (e.g., Hochbaum's algorithm). Since MDA always divides the problem evenly, the total number of calls to the DRAP subroutine in MDA is $\Theta(n)$.

On the other hand, the divide-and-conquer scheme used in DCA is fairly distinct. DCA is an infeasibility-guided divide-and-conquer algorithm. At each recursion, it first ignores the nested bound constraints and solves an RAP relaxation of the instance. Then it examines the optimal solution of the relaxation and divides the problem into subproblems if and only if a maximum violation of the nested bound constraints is found. If the optimal solution of RAP relaxation is feasible with respect to the nested bound constraints, then the subproblem is already solved. Hence, it is possible that the number of subproblems solved by DCA is substantially fewer than $\Theta(n)$, the number of subproblems solved by MDA. Even though MDA is guaranteed to have a better worst-case time complexity, DCA is may have a better performance under the criterion of average-case complexity.

In the following, we provide constructed examples to demonstrate the idea above.

An example on which DCA is better than MDA

Consider the following instance:

$$\min \quad -Mx_{n+1} + \sum_{k=1}^{n+1} x_k^2 \tag{2.11a}$$

$$\text{s.t.} \quad i \le \sum_{k=1}^{i} x_k \le i, \qquad \forall i \in [1:n] \tag{2.11b}$$

$$\sum_{k=1}^{n+1} x_k = n, \tag{2.11c}$$

$$0 \le x_i \le 2n, x_i \in \mathbb{Z}, \qquad \forall i \in [1:n+1]. \tag{2.11d}$$

where M is a sufficiently large number and the objective function $f_i(\cdot)$ is convex for each $i \in [1:n+1]$. Note that DRAP-NC$(1, n+1)$ has a unique optimal solution in which $x_{n+1} = 0$ and $x_i = 1$ otherwise, and its relaxation DRAP$(1, n+1)$ has a unique optimal solution in which $x_{n+1} = n$ and $x_i = 0$ for other i. The maximum violation of is achieved by the nested bound constraint with index n. Then we set $K = n$ and divide the problem into two subproblems DRAP-NC$(1, n)$ and DRAP-NC$(n+1, n+1)$.

1. DRAP-NC$(1, n)$:

$$\min \quad \sum_{k=1}^{n} x_k^2 \tag{2.12a}$$

$$\text{s.t.} \quad i \le \sum_{k=1}^{i} x_k \le i, \qquad \forall i \in [1:n-1] \tag{2.12b}$$

$$\sum_{k=1}^{n} x_k = n, \tag{2.12c}$$

$$0 \le x_i \le 2n, x_i \in \mathbb{Z}, \qquad \forall i \in [1:n], \tag{2.12d}$$

2. DRAP-NC$(n+1, n+1)$:

$$\min \quad x_{n+1}^2 - Mx_{n+1} \tag{2.13a}$$

$$\text{s.t.} \quad x_{n+1} = 0, \tag{2.13b}$$

$$0 \le x_i \le 2n, x_i \in \mathbb{Z}. \tag{2.13c}$$

Observe that the relaxed problem DRAP$(1, n)$ has a unique optimal solution $(1, \ldots, 1)$ and the the relaxed problem DRAP$(n+1, n+1)$ has a unique optimal solution 0. In addition, these

solutions do not violated any of the nested bounds. Therefore, we can obtain the optimal solution to the original problem DRAP-NC by solving exactly three subproblems. In comparison, MDA still needs to solve $\Theta(n)$ subproblems to obtain the final solution.

Benchmark convex functions

We test the performance of DCA and MDA on randomly generated DRAP-NC instances with three classes of benchmark convex objectives used in the paper [44]: [F], [CRASH], and [FUEL].

$$[\text{F}]: f_i(x) = \frac{x^4}{4} + p_i x, \tag{2.14}$$

$$[\text{CRASH}]: f_i(x) = k_i + \frac{p_i}{x}, \tag{2.15}$$

$$\text{and } [\text{FUEL}]: f_i(x) = p_i \times c_i \times (\frac{c_i}{x})^3. \tag{2.16}$$

Convex objective [F]

[F] is a convex cost function that has the form

$$f_i(x) = \frac{x^4}{4} + p_i x.$$

In each instance, the function $f_i(x)$ has the form $f_i(x) = \frac{x^4}{4} + p_i x$, where p_i is drawn from a uniform distribution over $[-1, 1]$. The parameter V_b is set to 100. We generate 14 different n values ranging from 800 to 6 millions and 10 instances for each value of n. The average running time as well as the average number of calls of solving RAP subproblems are reported in Table 2.3 and Figure 2.3.

Parameters	CPU time (s)		Average number of subproblems solved	
n	DCA	MDA	DCA	MDA
800	0.011	0.021	98.8	6396
1600	0.015	0.038	142.2	12796
3200	0.033	0.090	177.2	25596
6400	0.071	0.197	342.0	51196
12800	0.155	0.447	374.0	102396
25600	0.375	0.989	561.6	204796
51200	1.221	2.493	1310.2	409596
102400	2.199	4.945	1179.8	819196
204800	5.693	11.163	2598.2	1638396
409600	12.044	26.132	2230.4	3276796
819200	32.200	62.631	2762.6	6553596
1638400	80.897	149.035	4738.0	13107196
3276800	148.142	363.447	3731.2	26214396
6553600	431.889	868.560	9323.4	52428796

Table 2.3: Solution statistics of DCA and MDA for instances with convex cost objectives [F].

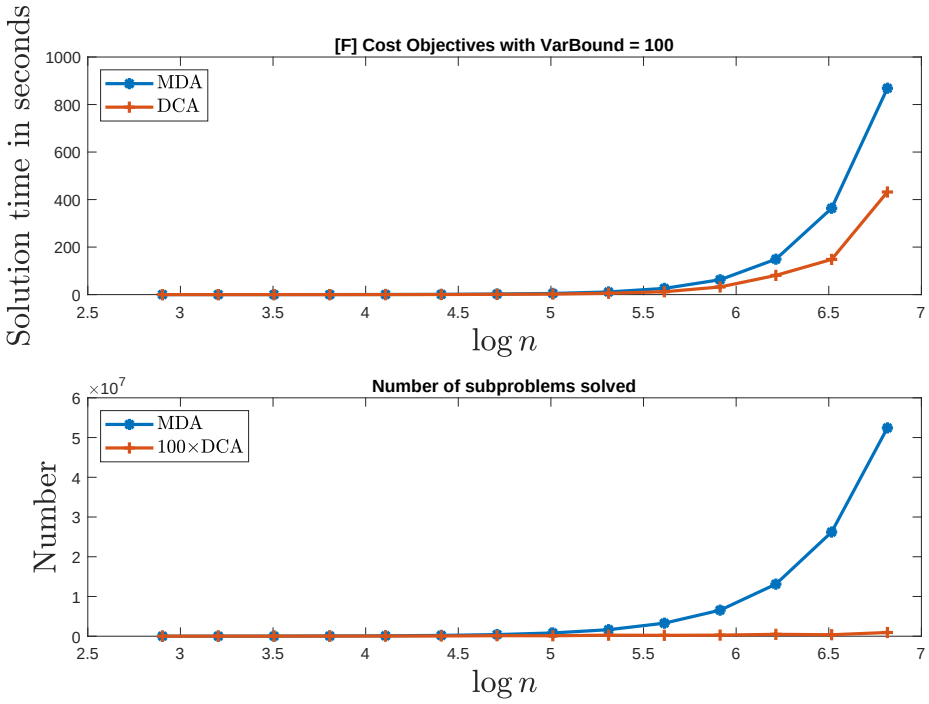


Figure 2.3: Solution time and number of subproblems solved statistics of DCA and MDA for instances with convex cost objectives [F].

Convex objective [CRASH]

[CRASH] is a convex cost function that has the form

$$f_i(x) = k_i + \frac{p_i}{x}.$$

In each instance, the function $f_i(x)$ has the form $f_i(x) = k_i + \frac{p_i}{x}$, where p_i and k_i are drawn from a uniform distribution over $[0, 1]$. The parameter V_b is set to 100. We generate 14 different n values ranging from 800 to 6 millions and 10 instances for each value of n. The average running time as well as the average number of calls of solving RAP subproblems are reported in Table 2.4 and Figure 2.4.

Parameters	CPU time (s)		Average number of subproblems solved	
n	DCA	MDA	DCA	MDA
800	0.011	0.021	110.2	6396
1600	0.016	0.040	130.8	12796
3200	0.039	0.089	256.6	25596
6400	0.097	0.202	361.4	51196
12800	0.229	0.446	654.4	102396
25600	0.454	1.077	636.6	204796
51200	1.165	2.266	1147.2	409596
102400	2.424	5.234	1010.8	819196
204800	6.704	12.115	1791.0	1638396
409600	16.761	27.381	2930.6	3276796
819200	39.872	62.899	3455.6	6553596
1638400	86.280	146.509	4758.8	13107196
3276800	222.655	346.476	9547.8	26214396
6553600	470.372	846.707	11013.2	52428796

Table 2.4: Solution statistics of DCA and MDA for instances with convex cost objectives [CRASH].

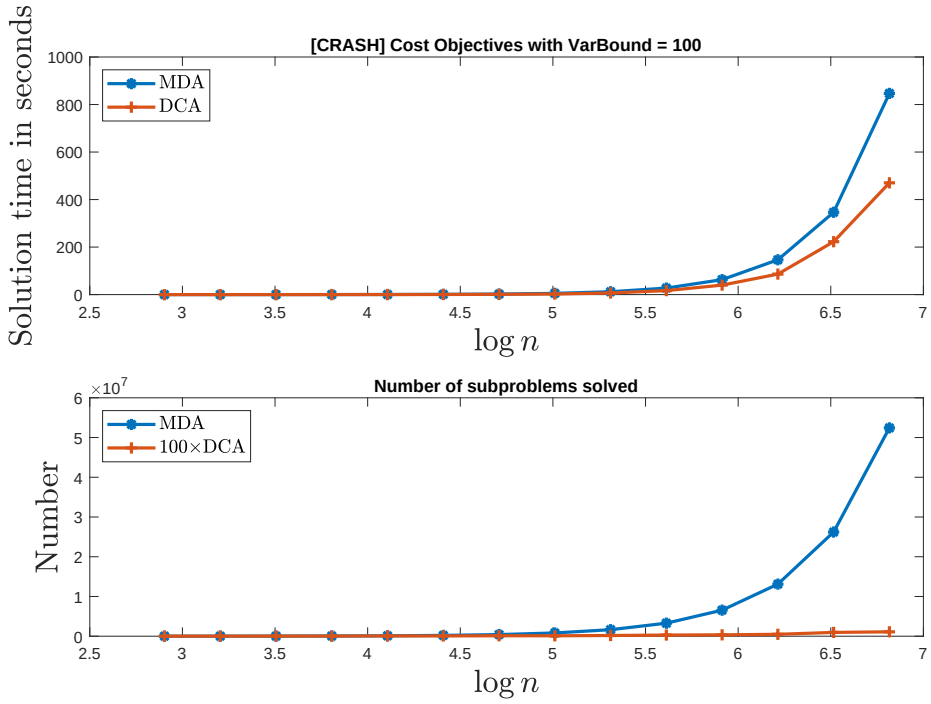


Figure 2.4: Solution time and number of subproblems solved statistics of DCA and MDA for instances with convex cost objectives [CRASH].

Convex objective [FUEL]

[FUEL] is a convex cost function that has the form

$$f_i(x) = p_i \times c_i \times (\frac{c_i}{x})^3.$$

It is a function used in speed optimization to measure fuel costs [78].

In each instance, the function $f_i(x)$ has the form $f_i(x) = p_i \times c_i \times (\frac{c_i}{x})^3$, where p_i and c_i are drawn from a uniform distribution over $[0, 1]$. The parameter V_b is set to 100. We generate 14 different n values ranging from 800 to 6 millions and 10 instances for each value of n. The average running time as well as the average number of calls of solving RAP subproblems are reported in Table 2.5 and Figure 2.5.

Parameters	CPU time (s)		Average number of subproblems solved	
n	DCA	MDA	DCA	MDA
800	0.013	0.024	135.8	6396
1600	0.020	0.044	151.6	12796
3200	0.041	0.102	220.8	25596
6400	0.099	0.225	421.8	51196
12800	0.234	0.506	502.2	102396
25600	0.539	1.131	514.4	204796
51200	1.221	2.493	1310.2	409596
102400	2.665	5.829	912.2	819196
204800	7.333	12.982	2225.2	1638396
409600	18.617	30.030	2916.2	3276796
819200	40.721	72.228	2771.2	6553596
1638400	108.050	167.963	6643.6	13107196
3276800	237.597	402.043	7161.0	26214396
6553600	586.074	936.209	14240.2	52428796

Table 2.5: Solution statistics of DCA and MDA for instances with convex cost objectives [FUEL].

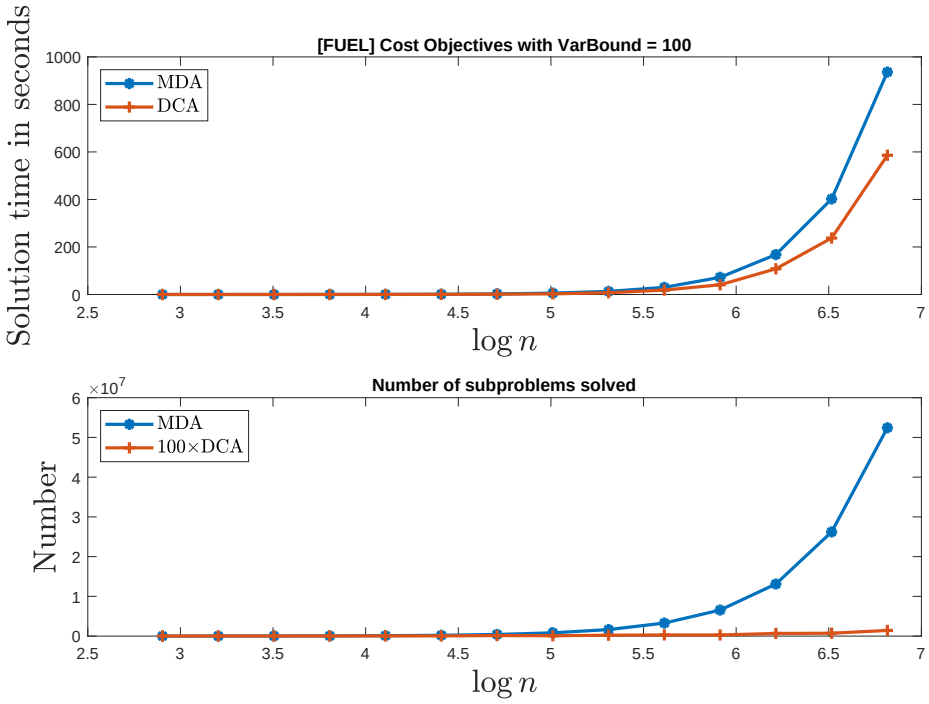


Figure 2.5: Solution time and number of subproblems solved statistics of DCA and MDA for instances with convex cost objectives [FUEL].

It can be seen that for all the test instances, DCA solved significantly fewer subproblems than MDA and the average running time of DCA is at least 50% less than that of MDA on the same instance. Though MDA has a strictly better worst-case complexity, on the benchmark instances in the literature, DCA is competitive compared to MDA.

2.5.3 Non-separable convex objective

Our last numerical experiment is on the Support Vector Ordinal Regression (SOVREX) model. SOVREX aims to optimize multiple thresholds to define parallel discriminant hyperplanes to classify samples with ordinal scales [79]. Vidal et al. [44] show that the dual problem of SOVREX in [79] is a non-separable convex optimization problem over a special case of the RAP-NC constraint polytope (2.17):

$$\max \quad \sum_{j=1}^{r}\sum_{i=1}^{n^j}(\alpha_i^j + \alpha_i^{j*}) \qquad + \qquad \sum_{j=1}^{r}\sum_{i=1}^{n^j}\sum_{j'=1}^{r}\sum_{i'=1}^{n^j}(\alpha_i^j - \alpha_i^{j*})(\alpha_{i'}^{j'} - \alpha_{i'}^{j'*})]\mathcal{K}(x_i^j, x_{i'}^{j'}) \tag{2.17a}$$

$$\text{s.t.} \quad 0 \le \alpha_i^j \le C, \qquad \forall i \in [1:n^j], j \in [1:r-1], \tag{2.17b}$$

$$0 \le \alpha_i^{j*} \le C, \qquad \forall i \in [1:n^j], j \in [2:r], \tag{2.17c}$$

$$\alpha_i^r = 0, \qquad \forall i \in [1:n^r], \tag{2.17d}$$

$$\alpha_i^{1*} = 0, \qquad \forall i \in [1:n^1], \tag{2.17e}$$

$$\sum_{k=1}^{j}(\sum_{i=1}^{n^j}\alpha_i^j - \sum_{i=1}^{n^{j+1}}\alpha_i^{j+1*}) \ge 0, \qquad j \in [1:r-1], \tag{2.17f}$$

where $\mathcal{K}(x_i^j, x_{i'}^{j'})$ is the kernel function and x_i^j is the feature of sample (i, j). It is not difficult to see that (2.17) can be transformed into RAP-NC with a variable substitution of $\boldsymbol{\alpha}^*$ and (2.17f) becomes the nested bound constraints. In [44], Vital et al. evaluate the performance of MDA with a projected gradient procedure for finding the optimal value of the dual of SOVREX. The procedure is a type of block coordinate ascent method that maintains a small-sized working set of variables to optimize while the other variables remain fixed. In particular, instances of RAP-NC are solved by MDA in the projection step. To ensure convergence, a minimal working set may need to contain the two variables which most violate the KKT conditions [79]. When the working set contains only two samples, the projection subproblem can be solved analytically. However, an analytic solution for the projection subproblem is not available if the working set contains more than two samples. Larger working set can reduce the total number of iterations while the cost of each projected gradient step is more expensive. Hence, a fast algorithm for the projection subproblem is critical.

The projected gradient procedure is summarized in Algorithm 4. It is adapted from [44]. We follow most of the general settings in [44] (data preparation, a step size of $\gamma = 0.2$, working set size $n_{ws} = \{2, 4, 6, 8, 10\}$, training set size, etc) and evaluate the performance on the same eight instances[1]. In the working set selection, we select the most-violated samples pairs for each hyperplane rather than the most-violated sample pairs as in [44]. We also use Gaussian kernel $K(x, x') = exp(-\frac{\kappa}{2}\sum_{s=1}^{d}(x_s - x'_s)^2)$. More detailed guidelines can be found in [79] and [44].

Algorithm 4 A projected gradient ascent approach to solve SVOREX (Adapted from [44, Algorithm 4])

1: **Settings:** Gaussian kernel with parameter κ, penalty parameter C, working set size $n_{ws} \in \{2, 4, 6, 8, 10\}$, step size $\gamma = 0.2$, $n_{grad} = 20$. Let n be the number of samples and m be the size of working set.
2: **Initialization:** Set initial feasible solution $(\boldsymbol{\alpha}, \boldsymbol{\alpha}^*) = \mathbf{0}$. Precompute the gradient. ($O(n^2)$ operations)
3: **while** there exist samples that violate the KKT conditions **do**
4: Select a working set W containing the most-violated samples pairs for each hyperplane with maximum size n_{ws}. ($O(n)$ operations)
5: **for** n_{grad} iterations **do**
6: Perform a gradient ascent update for variables $(\alpha_i^j, \alpha_i^{j*}), (i, j) \in W$. ($O(m)$ operations)
7: Project the solution to the feasible region by solving a quadratic RAP-NC.
8: Update the gradient for $(\alpha_i^j, \alpha_i^{j*}), (i, j) \in W$ incrementally. ($O(m^2)$ operations)
9: **end for**
10: Update the gradient for all the variables incrementally. ($O(nm)$ operations)
11: Check whether the stopping criteria is satisfied based on the gradient. ($O(n)$ operations)
12: **end while**

Since projection problem is a continuous RAP-NC, we implement a bisection method on the dual variables to solve the continuous RAP subproblems to obtain faster solution time. The solution statistics is summarized in Table 2.6. For each of the eight instances, the first column

[1]The data sets are available at `https://www.dcc.fc.up.pt/~ltorgo/Regression/DataSets.html`[80]

reports the number of samples N in the training set and the dimension D of the samples. We select the first N samples in the data set. The second column is the size of the working set. We run two tests, Grad-MDA and Grad-DCA, in which MDA and DCA are implemented to solve the projection subproblems in the projected gradient ascent procedure. In the third and the fourth column, we report the total number of iterations I_{ws}, total solution time T, time spent in the projection step T_{pro}, and the total number of RAP subproblems solved for Grad-MDA and Grad-DCA, respectively. Notice that if every subproblem is solved exactly, the Grad-MDA and Grad-DCA should have the same number of iterations I_{ws} for the same set of parameters. However, in general an exact solution is not possible for continuous problems. The number of iteration I_{ws} is slightly different due to the fact that solutions returned by the MDA and DCA are slightly different within the precision tolerance. Finally, we perform a sanity check by comparing the solutions obtained by different working set size and algorithms for projection subproblems. For the same data set, all the solutions turn out to be the same. In the gradient ascent procedure, the most expensive operations in Algorithm are Step 10 and Step 10. For all the eight data sets, as indicated by the test results, larger working set size can reduce the total number of iterations significantly and in turn reduce such expensive operations. For all the test instances, the solution time of Grad-DCA is competitive to Grad-MDA as it solves much fewer RAP subproblems, though the average size of the RAP subproblems of Grad-DCA might be larger. If we only compare the time spent in solving projection subproblems, DCA takes at least 50% less time than MDA.

2.5.4 Concluding remarks

According to the test results above, DCA is on average six times faster on DRAP-NC with linear objectives and five to six orders of magnitude faster on DRAP-NC with quadratic objectives than Gurobi. For DRAP-NC with general convex objectives, the averaged solution time of DCA is at least 50% less than that of MDA. For the RAP-NC subproblems in the projected gradient procedure for solving SOVREX, the solution time in solving projection subporblems of DCA is also at least 50% less than that of MDA. Even though MDA has a strictly better worst-case time complexity, DCA takes advantage of its simple infeasibility-guided divide-and-conquer framework and usually solve much fewer RAP subproblems. The solution time of DCA is competitive to that of MDA in solving RAP-NC and DRAP-NC on the test instances.

A natural question to ask is: in which case will DCA perform better than MDA and vice versa? Although we have already provided examples to show one way or the other, these instances are very specific. It is an interesting question that whether other factors, e.g., types of objectives, will have impact on the performance of the two algorithms. We do not have a definite answer to the question. We will leave the practitioners to decide which algorithm to use

based on their applications.

Dataset			n_{ws}	Grad-MDA				Grad-DCA			
Name	N	D		I_{ws}	$T(s)$	$T_{pro}(s)$	N_{RAP}	I_{ws}	$T(s)$	$T_{pro}(s)$	N_{RAP}
Abalone	1000	8	2	118467	14.166	7.938	66341520	118654	9.85	3.313	6579104
			4	124855	31.075	21.776	149826000	124498	18.433	9.87	13955896
			6	83350	31.973	24.022	153364000	84007	17.451	10.09	11481052
			8	70642	37.422	28.725	175158560	70449	17.543	10.284	9861040
			10	61604	40.157	31.786	191843840	58835	18.083	9.978	8233582
Bank	3000	32	2	57348	12.069	3.498	32114880	57348	11.105	2.012	4730000
			4	19158	8.284	3.371	22989600	19161	6.665	1.644	2480272
			6	7819	4.954	2.014	14386960	7765	3.702	0.926	1077764
			8	5087	4.296	1.947	12615760	5055	3.082	0.726	707584
			10	4335	4.632	2.271	13519760	4332	3.05	0.755	605792
Boston	300	13	2	11028	0.999	0.819	6175680	11028	0.815	0.444	746994
			4	6771	1.407	1.252	8125200	6768	0.729	0.579	864346
			6	4064	1.339	1.251	7477760	3895	0.584	0.484	542676
			8	2990	1.334	1.255	7407520	2876	0.525	0.437	402592
			10	2413	1.394	1.309	7471280	2377	0.499	0.411	332702
California	5000	8	2	359895	127.864	23.733	201541200	359325	116.138	12.134	24027962
			4	237386	146.917	41.531	284863200	236167	120.703	19.814	28281968
			6	172963	148.31	48.984	318251920	175589	122.193	20.312	24186180
			8	154286	175.368	63.189	382608160	152036	129.443	21.041	21282832
			10	139432	199.435	73.463	434853760	137885	141.489	24.178	19301354
Census	6000	16	2	171790	71.372	11.393	96202400	171790	68.202	6.261	13734990
			4	137168	96.736	25.13	164601600	137239	82.837	12.064	17437806
			6	98126	99.027	29.567	180551520	98727	80.665	12.425	13762340
			8	81162	106.582	33.925	201260000	81108	83.59	13.025	11354772
			10	70869	113.658	37.932	221018480	70701	88.648	12.204	9897152
Computer	4000	21	2	349175	99.381	23.988	195538000	349116	88.383	11.758	22475546
			4	213392	111.045	39.995	256070400	215489	88.985	17.597	25757486
			6	152462	115.837	45.81	280530080	150138	86.438	18.171	20829324
			8	132024	131.732	56.471	327411200	130958	93.271	18.742	18333576
			10	115968	144.582	64.051	361657920	116843	97.942	19.364	16357458
Machine CPU	150	6	2	28467	2.309	2.0	15941520	28621	1.51	1.048	1911252
			4	13569	2.584	2.394	16282800	13430	1.36	1.154	1666230
			6	7932	2.472	2.294	14594240	7897	1.128	0.969	1094396
			8	6619	2.786	2.628	16339920	6199	1.052	0.919	867636
			10	5863	3.176	2.978	18008400	5834	1.129	0.983	816334
Pyrimidines	50	27	2	629	0.119	0.096	352240	629	0.158	0.073	50180
			4	295	0.084	0.074	354000	295	0.043	0.031	39224
			6	168	0.083	0.068	308480	154	0.034	0.023	21464
			8	128	0.072	0.067	308800	131	0.034	0.023	18304
			10	108	0.06	0.055	321920	115	0.023	0.02	15994

Table 2.6: Statistic of Grad-DCA and Grad-MDA on SVOREX benchmark instances

2.6 Conclusions

In the chapter, we proposed a simple and efficient exact algorithm to solve the discrete resource allocation problem with nested constraints. The algorithm is an infeasibility-guided divide-and-conquer algorithm which makes multiple calls to a scaling-based subroutine. The algorithm does not require any additional assumption on the cost function other than convexity, so it can be applied to many applications where the form of the cost function is not known. We conducted extensive computational experiments on instances with a variety of convex objectives. The numerical results demonstrate the efficiency of our algorithm in solving large-sized DRAP-NC instances compared to Gurobi the state-of-the-art commercial solver, and MDA, the algorithm with the best worse case complexity in the literature.

Chapter 3

Minimum convex cost network flow over the dynamic lot size network

In this chapter, we study a minimum convex cost network flow problem on the dynamic lot size network $G = (N, A)$ as shown in Figure 3.1.

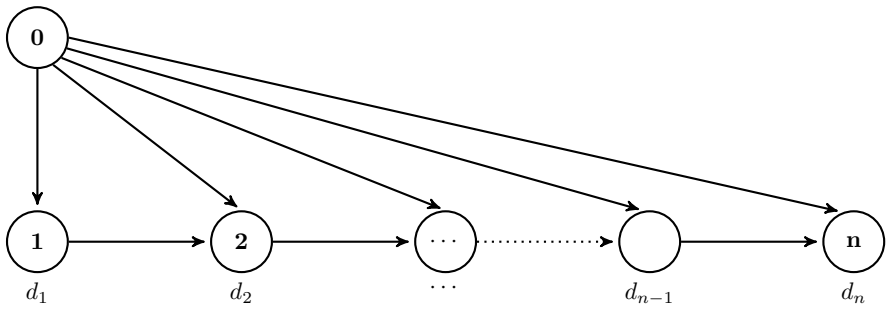


Figure 3.1: The dynamic lot size network.

The dynamic lot size network is defined as follows: there are one source node 0 and n sink nodes $1, 2, \ldots, n$, with demands $d_1, d_2, \ldots, d_n$. Each arc $e \in A$ is associated with a lower bound 0, an upper bound u_e, and a convex cost function $f_e(\cdot)$. We are interested in the following convex optimization problem (P2):

> Given a dynamic lot size network $G = (N, A)$ described in Figure 3.1, find a flow $\boldsymbol{x}$ on G to minimize the total arc cost $\sum_{e \in A} f_e(x_e)$ and satisfy all the flow balance and capacity constraints.

As it will be demonstrated in Section 3.2.1, (P2) is a generalization of (P1) with additional convex costs on the amount of resources allocated to the subsets of activities. To solve (P2) efficiently, we design a scaling-based algorithm, the Scaled Flow-improving Algorithm (SFA). SFA is designed for (P2) with restriction to integer flows but it can be easily extended to solve the continuous problem. The running time of our algorithm is $O(n^2 \log \frac{B}{n})$ for the integer problem and $O(n^2 \log \frac{B}{n\epsilon})$ for the continuous problem for finding an ϵ-optimal solution where $B = \sum_{i=0}^{n} d_i$. In particular, for (P1) that was considered in the last chapter, our algorithm terminates in $O(n \log n \log \frac{B}{n\epsilon})$ time for the continuous problem and $O(n \log n \log \frac{B}{n})$ time for the integer problem. The speed up is obtained by using data structures segment tree and red-black tree for two key operations in our algorithm. The complexity matches the best result for (P1) in the literature [44]. We evaluate the performance of our algorithm on test instances of (P1) with a variety of convex objectives, and then compare the results with the performance of DCA in Chapter 2 and MDA [44], which has the best worst-case time complexity for (P1). Numerical experiments demonstrate that SFA has the best performance among the three algorithms on

the benchmark instances.

3.1 Introduction

In this chapter, we are interested in (P2), a minimum convex cost network flow problem on the dynamic lot size network in Figure 3.2. For each positive integer $i \leq j$, let $[i:j]$ denote the set $\{i, i+1, \ldots, j\}$. For $i \in [1:n]$, let x_i denote the flow on the vertical arc $(0, i)$. For $i \in [1:n-1]$, let y_i denote the flow on the horizontal arc $(i, i+1)$.

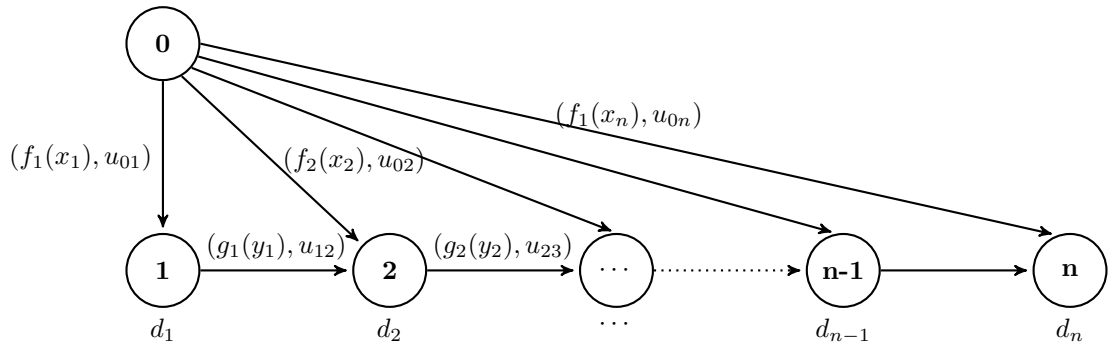


Figure 3.2: The dynamic lot size network. The pair $(f(x), u)$ alongside each arc indicates the arc incurs a cost $f(x)$ with x units of flow and its capacity is u.

(P2) can be formulated as a convex program with a separable objective function and linear constraints (3.1):

$$\min \quad \sum_{i=1}^{n} f_i(x_i) + \sum_{i=1}^{n-1} g_i(y_i) \tag{3.1a}$$

$$\text{s.t.} \quad y_1 = x_1 - d_1, \tag{3.1b}$$

$$y_i = y_{i-1} + x_i - d_i, \qquad \forall i \in [2:n-1], \tag{3.1c}$$

$$0 = y_{n-1} + x_n - d_n, \tag{3.1d}$$

$$0 \leq x_i \leq u_{0,i}, \qquad \forall i \in [1:n], \tag{3.1e}$$

$$0 \leq y_i \leq u_{i,i+1}, \qquad \forall i \in [1:n-1]. \tag{3.1f}$$

where the cost functions $f_i(\cdot), i \in [1:n]$ and $g_i(\cdot), i \in [1:n-1]$ are convex.

3.2 Literature review

3.2.1 Applications

(P2) appears in many applications. We give three examples below.

1. (P2) is the underlying optimization model of the dynamic lot-sizing problem, which is a classic problem in operation research. In [81], Ahuja and Hochbaum viewed the dynamic lot-sizing problem as a minimum linear cost network flow problem on the dynamic lot size network in Figure 3.1. Instead of solving it with classic polynomial-time algorithms for minimum cost network flow problems, they developed a faster algorithm that solves the problem in $O(n \log n)$ time. Their algorithm only solves the linear model with production cost and back-orders. (P2) essentially extends the models in [81] to convex production cost and convex holding cost.

2. The second example is the resource allocation problem, a fundamental problem with a wide variety of applications in search theory, economics, inventory systems, etc [41]. It aims to allocate a fixed amount of resources to a set of activities so as to minimize the allocation cost subjective to certain constraints. We refer the readers to [15] [40] [41] for a comprehensive review of the resource allocation problem and its algorithms. The resource allocation problem (RAP-NC) studied in [44] and Chapter 2 is a resource allocation problem with additional constraints on the amount of resources allocated to subsets of activities. It can be formulated as the following convex optimization problem:

$$\min_{x} \quad \sum_{i=1}^{n} f_i(x_i) \tag{3.2a}$$

$$\text{s.t.} \quad a_i \le \sum_{k=1}^{i} x_k \le b_i, \qquad \forall i \in [1 : n-1], \tag{3.2b}$$

$$\sum_{i=1}^{n} x_i = B, \tag{3.2c}$$

$$0 \le x_i \le d_i, \qquad \forall i \in [1 : n]. \tag{3.2d}$$

where the allocation cost for activity i is given by a convex function $f_i : [0, d_i] \to \mathbb{R}$, for $i \in [1 : n]$. We introduce new variables $y_i = \sum_{k=1}^{i} x_k - a_i, i \in [1 : n-1]$ and reorganize

the constraints in (3.2) as follows,

$$\min \quad \sum_{i=1}^{n} f_i(x_i) \tag{3.3a}$$

$$\text{s.t.} \quad y_1 = x_1 - a_1, \tag{3.3b}$$

$$y_i = y_{i-1} + x_i - (a_i - a_{i-1}), \qquad \forall i \in [2 : n-1], \tag{3.3c}$$

$$0 = x_{n-1} + x_n - (a_n - a_{n-1}), \tag{3.3d}$$

$$0 \le y_i \le b_i - a_i, \qquad \forall i \in [1 : n-1], \tag{3.3e}$$

$$0 \le x_i \le d_i, \qquad \forall i \in [1 : n]. \tag{3.3f}$$

It is a special case of (P2) without the cost functions $g_i(\cdot)$'s in Formulation (3.1).

3. The third example is the speed optimization problem arising in transportation science [10] [11] [12]. The goal of the problem is to find optimal speeds between two consecutive nodes over a fixed route while subjected to minimize the total cost, e.g., fuel cost, emission cost, and customer inconvenience cost. The speed optimization problem can be formulated as follows:

$$\min_{\boldsymbol{v},\boldsymbol{t}} \quad \sum_{i=1}^{n-1} d_i f_i(v_i) + \sum_{i=1}^{n} g_i(t_i) \tag{3.4a}$$

$$\text{s.t.} \quad a_i \le t_i \le b_i, \qquad \forall i \in [1 : n], \tag{3.4b}$$

$$t_i + \frac{d_i}{v_i} \le t_{i+1}, \qquad \forall i \in [1 : n-1], \tag{3.4c}$$

$$l_i \le v_i \le u_i, \qquad \forall i \in [1 : n-1], \tag{3.4d}$$

where $f_i(\cdot)$'s are convex cost functions representing the fuel cost per mile and $g_i(\cdot)$'s are convex functions representing customer inconveniences. In [10], Dumas et al. studied the speed optimization whose goal is to minimize total customer inconvenience cost. In [11], Fagerholt et al. aimed to find the optimal speed to minimize the total fuel cost while the fuel cost functions per mile over each arc are the same. In [12], He et al. developed an efficient divide-and-conquer algorithm for the problem with heterogeneous convex fuel costs. Let v_i^* denote the minimizer of $f_i(\cdot)$ for $i \in [1 : n-1]$. The speed optimization

problem (3.4) can be transformed into an equivalent optimization model as follows:

$$\min_{\boldsymbol{x},\boldsymbol{t}} \quad \sum_{i=1}^{n-1} h_i(\frac{d_i}{x_i}) + \sum_{i=1}^{n} g_i(t_i) \tag{3.5a}$$

$$\text{s.t.} \quad t_{i+1} = x_{i+1} + t_i, \qquad \forall i \in [1:n-1], \tag{3.5b}$$

$$a_i \le t_i \le b_i, \qquad \forall i \in [1:n], \tag{3.5c}$$

$$\frac{d_i}{u_i} \le x_{i+1}, \qquad \forall i \in [1:n-1], \tag{3.5d}$$

where

$$h_i(v) = \begin{cases} f_i(v_i^*) & \text{if } v \le v_i^\star, \\ f_i(v) & \text{if } v > v_i^\star. \end{cases}$$

It can be showed that $h_i(\cdot)$ is convex and therefore, (3.5) is a special case of (P2). An efficient algorithm is critical to the speed optimization problem, which needs to be solved as a subproblem many times in route-planning algorithms [82].

3.2.2 Existing algorithms

(P2) is a minimum convex cost network flow problem (MCCNFP) on a structured network. In this section, we review some algorithms for the minimum cost network flow problems (MCNFP) and MCCNFP on other structured networks.

In [83], Vaidyanathan and Ahuja considered MCNFP on lines and circles. In [84], Orlin and Vaidyanathan extended the result to problems with convex piece-wise cost functions. Under the assumption that the cost functions have at most $O(n)$ pieces, their algorithms are able to achieve an $O(n \log n)$ speed up over the general algorithms on structured networks of lines, circles, and trees. Their algorithm also solves (P2) when the vertical arcs have identical linear cost and the horizontal arcs have convex piece-wise linear cost.

In [81], Ahuja and Hochbaum studied the dynamic lot-sizing problems with linear production costs and back-orders. They proposed an efficient algorithm which we refer to as *A-H algorithm*. A-H algorithm is a special implementation of the successive shortest path algorithm for MCNFP that leverages the underlying network structure with two data structures. It is strongly polynomial with a time complexity of $O(n \log n)$.

In general, however, these algorithms can not be applied to (P2) directly. To the best of our knowledge, the best worst-case time complexity for (P2) is $O(n^2 \log n \log B)$ for integer problem and $O(n^2 \log n \log \frac{B}{\epsilon})$ for continuous problem, achieved by the capacity scaling algorithm [45].

Our contribution

In this chapter, we develop a new scaling-based algorithm, the Scaled Flow-improving Algorithm (SFA). The running time of our algorithm is $O(n^2 \log \frac{B}{n})$ for the integer problem and $O(n^2 \log \frac{B}{n\epsilon})$ for the continuous problem for finding an ϵ-optimal solution. It has a $\log n$ factor improvement over the current best algorithm. The algorithm is based on two ingredients: (1) a new scaling framework [67] that is not based on LP duality in the existing capacity scaling algorithm; (2) a stronger proximity result between the optimal solutions of the original problem and the scaled problem than the existing results on minimum convex cost flow problem.

In particular, for DRAP-NC and RAP-NC that were considered in Chapter 2, our algorithm terminates in $O(n \log n \log \frac{B}{n})$ time for the integer problem and $O(n \log n \log \frac{B}{n\epsilon})$ time for the continuous problem with $B = \sum_{i=0}^{n} d_i$. The speed up is obtained by using the data structures segment tree and red-black tree for two key operations in our algorithm. The complexity of SFA for RAP-NC and DRAP-NC matches the best known result in the literature [44].

We evaluate the performance of our algorithm on DRAP-NC test instances with a variety of convex objectives, and then compare the results with the performance of DCA in Chapter 2 and MDA [44], which has the best worst-case time complexity for DRAP-NC. The numerical experiment demonstrates that SFA has the best performance among the three algorithms.

The rest of this chapter is organized as follows: In Section 3.3, we give the general idea of SFA and briefly introduce three algorithms that are closely related to SFA. In Section 3.4, we present SFA and prove its correctness. In Section 3.5, we present a faster implementation of SFA for DRAP-NC and RAP-NC. In Section 3.6, we evaluate the performance of our algorithm with DCA and MDA on DRAP-NC test instances with a variety of convex objectives.

3.3 Preliminaries

In this section, we first sketch the idea of SFA and then briefly introduce three algorithms that are closely related to SFA. Based on the discussion in Section 1.6.2, to find an ϵ-solution forr the continuous MCCNFP, we can solve an integer MCCNFP with scaled data. In the following, we will focus on integer problem of (P2).

SFA falls in the scaling framework for separable convex optimization with linear constraints in [67]. It consists of several scaling phases. Each scaling phase is associated with a scaling parameter $s \in \mathbb{Z}$ and a flow $\boldsymbol{z}$. Initially, we start with sufficiently large s and zero flow. In an s-scaling phase, the flow $\boldsymbol{z}$ serves as a lower bound on the optimal flow of the original problem. In particular, each convex function in the objective of (3.1) is approximated by a piece-wise linear function with pieces of length s. We then employ an MCNFP algorithm to solve a piece-wise linear approximation of the original problem with $\boldsymbol{z}$ as the initial flow. A proximity theorem

indicates that a better lower bound $\boldsymbol{z}$ can be obtained by decrease each flow in the approximate solution. Then we decrease s by half and move to the next scaling phase with the new $\boldsymbol{z}$. When $s = 1$, an integer optimal solution can be found.

In the rest of this section, we briefly introduce three related algorithms. Two of them, the successive shortest path algorithm and the capacity scaling algorithm are general algorithms for MCNFP. The last one is the A-H algorithm [81], which is a specialized algorithm for MCNFP on the dynamic lot size network. Then we present a general approach for generalizing these algorithms to solve the problem with convex objectives, i.e., MCCNFP. These are building blocks of SFA. In fact, both of SFA and capacity scaling algorithm are scaling algorithms. Moreover, SFA adapts a generalized version of the A-H algorithm to solve the approximation problem efficiently. Finally, both of the capacity scaling algorithm and the A-H algorithm are based on the successive shortest path algorithm.

3.3.1 Pseudoflow and residual network

We start with the basic concepts in these algorithms: *pseudoflow* and *residual network*. Suppose $G = (N, A)$ is a directed graph with $n = |N|$ nodes and $m = |A|$ arcs, recall that MCCNFP can be formulated as the following convex optimization problem,

$$\min \quad \sum_{(i,j)\in A} f_{ij}(x_{ij}) \tag{3.6a}$$

$$\text{s.t.} \quad \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i), \qquad \forall i \in N, \tag{3.6b}$$

$$0 \le x_{ij} \le u_{ij}, \qquad \forall (i,j) \in A. \tag{3.6c}$$

where $f_{ij}(x_{ij}), (i,j) \in A$ are general convex functions, and $b(i)$ is the supply of node i if $b(i)$ is positive, and demand of node i otherwise.

Definition 19. ***(Pseudoflow [45])*** *A pseudoflow $\boldsymbol{x}$ is a solution of* (3.6) *that satisfies the nonnegativity and capacity constraints* (3.6c)*, but may violate the mass balance constraints* (3.6b) *of the nodes.*

For any pseudoflow $\boldsymbol{x}$, the *imbalance* of each node $i \in N$ is defined as

$$e(i) = b(i) - \Big(\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} \Big).$$

We say a node i is an *excess* node if $e(i) > 0$ and is a *deficit* node if $e(i) < 0$.

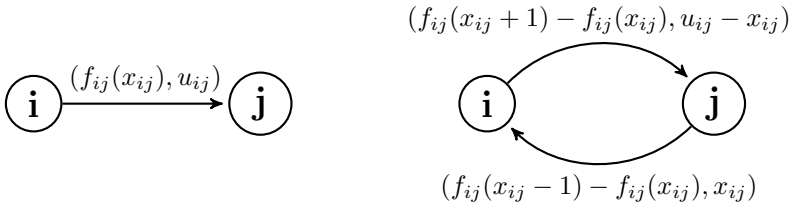


Figure 3.3: Illustrating the residual network $G(x)$. If $0 \leq x_{ij} \leq u_{ij}$, x_{ij} is a pseudoflow on arc (i,j). In the residual network, there is a regular if $x_{ij} < u_{ij}$ and a reverse arc is created if $x_{ij} > 0$. The cost in the residual network is defined as unit incremental cost.

The *residual network* $G(\boldsymbol{x})$ is defined on the network G corresponding a pseudoflow $\boldsymbol{x}$. Every arc (i,j) in G is replaced by two arcs (i,j) and (j,i). The new arc (i,j) has cost $c_{ij} = f_{ij}(x_{ij}+1) - f_{ij}(x_{ij})$ and residual capacity $r_{ij} = u_{ij} - x_{ij}$ and the arc (j,i) has cost $c_{ji} = f_{ij}(x_{ij}-1) - f_{ij}(x_{ij})$ and residual capacity $r_{ji} = x_{ij}$. In linear problems, $c_{ij} = -c_{ji}$. We refer to an arc $(i,j) \in A$ as a *regular arc* and its reversal (j,i) as a *reverse arc*. Finally, the residual network consists of only arcs with strictly positive capacities. In some scaling algorithms, we are interested in the *s-residual network*. Given a positive integer s, the s-residual network, denoted by $G(\boldsymbol{x}, s)$, is defined as the subgraph of the residual network $G(\boldsymbol{x})$ containing only arcs with capacity at least s. In $G(\boldsymbol{x}, s)$, the cost of an arc is the average incremental cost of the next s units of flow over that arc when the current flow in the network is x, i.e., $c_{ij} = \frac{f_{ij}(x_{ij}+s) - f_{ij}(x_{ij})}{s}$ for a regular arc and $c_{ji} = \frac{f_{ij}(x_{ij}-s) - f_{ij}(x_{ij})}{s}$ for a reverse arc.

3.3.2 Three algorithms

In the following, we give an overview of the *successive shortest path algorithm* and the *capacity scaling algorithm* for MCNFP, and the A-H algorithm for MCNFP on the dynamic lot size network. For more detailed and rigorous discussions, we refer readers to Chapter 9, 10, 14 of [45] and [81]. The successive shortest path algorithm is a classic pseudo-polynomial-time algorithm for MCNFP that is developed independently in [85], [86], and [87]. The capacity scaling algorithm by Orlin [59] is an improved version of the successive shortest path algorithm with a scaling technique. In [63] and [64], Minoux extended the capacity scaling algorithm to find the integer optimal solution for MCCNFP. The A-H algorithm [81] is a special implementation of the successive shortest path algorithm MCNFP on the dynamic lot size network.

The successive shortest path algorithm

The successive shortest path algorithm maintains a pseudoflow that satisfies the *reduced cost optimality conditions*. Initially, the pseudoflow is set to zero flow and the reduced cost optimality

is satisfied automatically. A pseudoflow is an optimal solution if it is a flow and satisfies the reduced cost optimality conditions at the same time. In each elementary iteration, we send flows along the shortest path from some excess node to some deficit node in the residual network. It can be showed that the new pseudoflow maintains the reduced cost optimality conditions. Finally, the pseudoflow becomes a flow and we obtain a feasible solution that is optimal. The algorithm is summarized in Algorithm 5.

Algorithm 5 The successive shortest path algorithm

1: **Input:** An instance of MCNFP over the network $G = (N, A)$.
2: **Output:** An optimal solution $\boldsymbol{x}^\star$.
3: **Initialization:** Set $x_{ij} = 0, (i, j) \in A$.
4: **while** $\boldsymbol{x}$ is not a flow **do**
5: Select an excess node s and a deficit node t. ▷ If there is no excess node or deficit node, then $\boldsymbol{x}$ must be a flow.
6: Find the shortest path P from node s to node t in the residual network $G(\boldsymbol{x})$.
7: Augment maximum possible flow along P and update the residual network $G(\boldsymbol{x})$.
8: **end while**
9: **return** $\boldsymbol{x}$.

The successive shortest path algorithm is a pseudo-polynomial-time algorithm in general. Yet it is very flexible. With carefully designed rules of selecting the excess and deficit node, we can greatly improve the asymptotic running time. The capacity scaling algorithm and the A-H algorithm are two examples.

The capacity scaling algorithm

The capacity scaling algorithm is an improved version of the successive shortest path algorithm for MCNFP. It suggests that focusing on the subgraph in which we can augment more flows is beneficial. It is a scaling-based algorithm.

Algorithm 6 The capacity scaling algorithm

Input: An instance of MCCNFP.
Output: An optimal solution $\boldsymbol{x}^\star$.
Initialization: Set $\boldsymbol{x} = \mathbf{0}$, node potential $\boldsymbol{\pi} = \mathbf{0}$, scaling parameter $s = 2^{\lfloor \log U \rfloor}$ where $U = \max\{u_{ij}, (i,j) \in A\}$.
while $s \geq 1$ **do** ▷ s-scaling phase
    Restore the invariant property based on $\boldsymbol{x}$ and $\boldsymbol{\pi}$. ($O(m)$ operation.)
    Excess nodes $S(s) := \{i \in N : e(i) \geq s\}$; Deficit nodes $T(s) := \{i \in N : e(i) \leq -s\}$;
    while $S(s) \neq \emptyset$ and $T(s) \neq \emptyset$ **do**
        Select an excess node $s \in S(s)$ and a deficit node $t \in T(s)$;
        Find the shortest path P from node s to node t in the s-residual network $G(\boldsymbol{x}, s)$;
        Update the node potential $\boldsymbol{\pi} : \boldsymbol{\pi} - \boldsymbol{d}$ where $d(i)$ is the **shortest distance** from the excess node s to node i in $G(\boldsymbol{x}, s)$;
        Augment $\boldsymbol{s}$ **unit** flow along P and update $G(\boldsymbol{x}, s), S(s), T(s)$.
    end while
end while
return $\boldsymbol{x}$.

It consists of several scaling phases, each of which is associated with a scaling parameter s and a pseudoflow. In each s-scaling phase, the algorithm maintains a pseudoflow that satisfies the reduced cost optimality conditions on the s-residual network $G(\boldsymbol{x}, s)$.Initially, we start with sufficiently large s and a zero flow. In each elementary iteration of the s-scaling phase, we send s units of flow along the shortest path from an excess node and a deficit node with demand at least s whenever it is possible. Then we decrease s by half and update the s-residual network. This may introduce new arcs that do not satisfy the reduced cost optimality conditions as we have changed s and arcs costs. We can somehow slightly modify the current pseudoflow to restore the optimality conditions. When $s = 1$, an integer optimal solution can be found.

The algorithm is summarized in Algorithm 6. It is adapted from Chap 10.2 and Chap 14.5 of [45]. Essentially, the capacity scaling algorithm is seeking for a primal feasible solution while it maintains the dual feasibility. It requires that the shortest-path algorithm used in the inner loop is a single-source-to-all-nodes shortest path algorithm. With the single-source-to-all-nodes distances, the algorithm maintains and updates the dual variables $\boldsymbol{\pi}$ in each iteration. The

capacity scaling algorithm terminates in $O(m \log U(m + n \log n))$ time where n is the number of nodes, m is the number of arcs, and U is largest capacity in the network.

The A-H algorithm

The A-H algorithm is a special implementation of the successive shortest path algorithm for MCNFP on the dynamic lot size network. It selects the n deficit nodes in the order of increasing index, i.e., $i = 1, 2, \cdots, n$. With such an order, the algorithm maintains a simple residual network. Moreover, two data structures can speed up Step 6 and Step 7 of Algorithm 5 to $O(\log n)$ time.We summarize A-H algorithm in Algorithm 7.

Algorithm 7 The A-H algorithm

1: **Input:** An instance of the dynamic lot-sizing problem: the network $G = (N, A)$ as in Figure 1.1.
2: **Output:** An optimal solution $\boldsymbol{x}^\star$.
3: **Initialization:** Set $x_i = 0, i \in [1 : n]$.
4: **for** $i = 1, 2, \cdots, n$ **do**
5: **while** $e(i) < 0$ **do**
6: Select the excess node 0 and the deficit node i;
7: Find the shortest path P from node 0 to node i in the residual network $G(x)$;
8: Augment maximum possible flow along P and the residual network $G(x)$.
9: **end while**
10: **end for**
11: **return** $\boldsymbol{x}$.

Now we elaborate the details of step 7 and 8 in Algorithm 7. At iteration i, demand at nodes $1, \cdots, i-1$ is satisfied and we send a flow along the shortest path until the demand at node i is met. On the other hand, demand at node $i+1$ is not yet satisfied. Therefore, there is zero flow on the arc $(i, i+1)$. By the definition of residual network, there is no reverse arc $(i+1, i)$ in the residual network. Hence, the shortest path from node 0 to node i must be one of the following paths,

$$0 \to k \to (k+1) \cdots \to i, \quad k \in [1 : i].$$

Using the notation in [81], we refer to these paths as $P_{ki}, k \in [1 : i]$. In iteration i of the outer loop, we can maintain a list of all the possible paths, each of which is associated with a unit cost c_{0k} of the arc $(0, k)$. Since all the horizontal arcs have zero cost, P_{ki} and $P_{k,i+1}$ have the same cost. Therefore, when we move to iteration $i+1$ of the outer loop, only one new path $P_{i+1,i+1}$

is added into the list while all the other path costs stay unchanged. We implement the list with a red-black tree and Step 7 can be executed in $O(\log n)$ time.

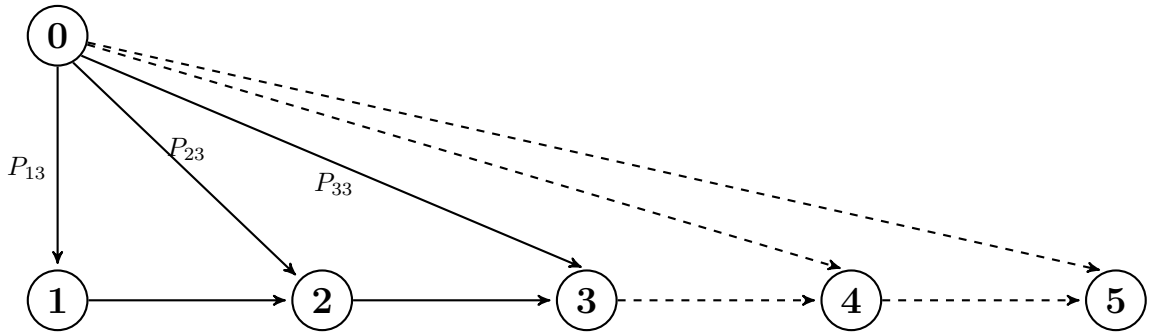


Figure 3.4: Illustrating the candidates of shortest paths from node 0 to 3 when $i = 3$. The possible paths are the following solid lines: (a) $P_{13} : 0 \to 1 \to 2 \to 3$; (b) $P_{23} : 0 \to 2 \to 3$; (c) $P_{33} : 0 \to 3$. The reverse arcs in the residual network $G(x)$ are omitted to keep the graph simple. Notice that there is no reverse arc $(4, 3)$ in the residual network $G(x)$ due to zero flow on the arc $(3, 4)$.

In Step 8, we need to figure out the maximum flow that can be augmented along the shortest path. If P_{ki} is selected, then the maximum amount is the minimum residual capacity among the arcs of the path and the deficit of the node i, i.e.,

$$\delta = \min\{r_{0k}, r_{k,k+1}, \cdots, r_{i-1,i}, -e(i)\}.$$

In [81], the data structure *dynamic tree* is used to make this step efficient. Alternatively, we may use a simpler data structure, *segment tree*, to achieve the same worst-case complexity. A segment tree, also known as a statistic tree, is a tree data structure used for storing information about intervals, or segments. It allows querying which of the stored segments contain a given point [88]. For a list of n values with fixed indices, $z_i \in \mathbb{R}, i \in [1 : n]$, it takes $O(n \log n)$ time to build a segment tree of these n values. Once it is built, we can perform the following two operations in $O(\log n)$ time:

1. *rangeMin*(i, j): return the index of the minimum among the values $\{z_i, z_{i+1}, \cdots, z_j\}$. Return None if $j < i$.
2. *rangeAdd*(i, j, v): augment a value v to each of the value in the set $\{z_i, z_{i+1}, \cdots, z_j\}$. Do nothing if $j < i$.

If we build a segment tree of the residual capacities of the horizontal arcs, we can find the bottleneck along the path P_{ki} by calling the *rangeMin*$(k, i-1)$, and one extra minimum

operation with the value $\min\{r_{0k}, -e(i)\}$. Furthermore, to update the costs on the residual network, we execute a single call of $rangeAdd(k, i-1, \delta)$.

In each iteration of the while loop in Algorithm 7, we either reduce the capacity of a path to 0 or set $e(i)$ to 0. Since there are $O(n)$ candidate paths and $O(n)$ deficit nodes in the dynamic lot size network, the total number of iterations in the while loop is bounded by $O(n)$. Moreover, each iteration in the while loop takes $O(\log n)$ time. Hence, the complexity of the A-H algorithm is $O(n \log n)$.

3.3.3 Transforming MCCNFP to MCNFP

The successive shortest path algorithm and the A-H algorithm cannot be applied to the network flow problem with convex arc costs directly. However, we can transform any instance of MCCNFP into an instance of MCNFP on an expanded network $G' = (N', A')$, and solve it by any MCNFP algorithm. We now give the general idea of the transformation. We refer readers to Chapter 14.3 of [45] for a rigorous discussion. Suppose we have an instance of MCCNFP on a directed graph $G = (N, A)$. In the transformation, for each arc $(i, j) \in A$ with a convex arc cost, we introduce u_{ij} parallel arcs in G', one arc corresponding to a unit incremental cost. The capacity of each parallel arc is one and their cost are $f_{ij}(1)-f_{ij}(0), f_{ij}(2)-f_{ij}(1), \cdots, f_{ij}(u_{ij})-f_{ij}(u_{ij}-1)$. It has been shown in [45] that solving the minimum convex cost flow problem in G is equivalent to solving the minimum cost flow problem in G'.

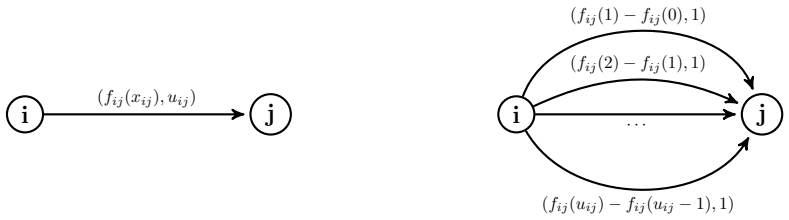


Figure 3.5: Illustrating the transformation of a MCCNFP to a MCNFP on an expanded network. For every arc $(i, j) \in A$ with capacity u_{ij}, there are u_{ij} arcs in the expanded network.

Usually, the parallel arcs of the graph G' can be handled implicitly. Because of the convexity of the cost function, the u_{ij} arcs have to be used in order. For instance, suppose we are trying to find the shortest path within the expanded network as defined in Figure 3.6. Even though in the residual network there are five arcs between node i and node j, only the third and the fourth (the first reverse arc) are the possible arcs that are in a shortest path due to convexity of the cost function. It suffices to consider a residual network with those two arcs instead of the whole expanded graph.

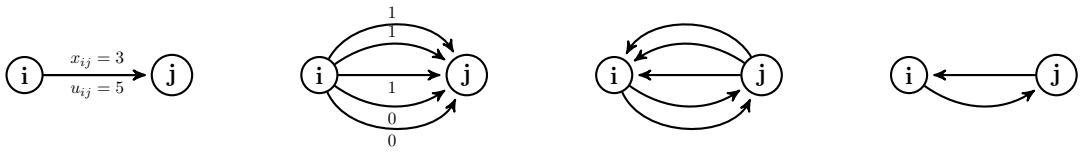


Figure 3.6: Illustrating the construction of the residual network. The four networks form left to right are the original network, the expanded network, the residual network of the expanded network, the residual network of the original network, respectively. Notice that arcs with zero capacity are excluded in the residual network. The capacity of each arc in the two residual networks is one.

Therefore, the residual network of the original network indeed handles the expanded network implicitly. Notice that all the arcs have unit capacity, one needs to modify Algorithm 5 to solve MCCNFP by sending exactly one unit of flow on each augmentation. The algorithm is summarized in Algorithm 8.

Algorithm 8 The successive shortest path algorithm with unit step size for MCCNFP

1: **Input:** An instance of MCCNFP.
2: **Output:** An optimal solution $\boldsymbol{x}^\star$.
3: **Initialization:** Set $x_{ij} = 0, (i,j) \in A$.
4: **while** $\boldsymbol{x}$ is not a flow **do**
5: Select an excess node s and a deficit node t;
6: Find the shortest path P from node s to node t in the residual network $G(x)$;
7: Augment **one unit** of flow along P and update $G(x)$. ▷ The arc cost in $G(x)$ needs to be updated
8: **end while**
9: **return** $\boldsymbol{x}$.

The only difference is that we send one unit flow and update the arc costs in the residual network $G(x)$ in Step 6 as the capacity is one for any arc in expanded network G'. Since only one unit is allowed in each elementary iteration, the time complexity of Algorithm 8 is $O(U(m + n \log n))$ where U is the total supply. The same idea can be applied to the A-H algorithm to the dynamic lot-sizing problem with convex production costs as well. The running time of the A-H algorithm, however, becomes $O(U \log n)$.

The capacity scaling algorithm suggests that it may be beneficial to send larger units, say step size s, in the early elementary iterations. Even though we may not obtain an optimal solution after the s-phase, we can modify the solution to restore some loop invariant. Then we

decrease the step size by a factor two and start another scaling phase. Finally, we can obtain the optimal solution in the last scaling phase with step size $s = 1$. Inspired by the above ideas, we develop the Scaled Flow-improving Algorithm (SFA), a scaling-based algorithm to solve (P2) efficiently.

3.4 SFA: a scaling-based algorithm for (P2)

In this section, we present the Scaled Flow-improving Algorithm (SFA), a scaling-based algorithm for (P2). It combines an adapted version of the A-H algorithm in [81] and the scaling framework for convex optimization with a separable objective in [67]. For the ease of exposition, we reorganize the constraints of the integer problem as follows,

$$\min \quad \sum_{i=1}^{n} f_i(x_i) + \sum_{i=1}^{n-1} g_i(y_i) \tag{3.7a}$$

$$\text{s.t.} \quad y_i = \sum_{k=1}^{i} x_k - \sum_{k=1}^{i} d_i, \qquad \forall i \in [1:n], \tag{3.7b}$$

$$y_n = 0, \tag{3.7c}$$

$$0 \le y_i \le u_{i,i+1}, \qquad \forall i \in [1:n-1], \tag{3.7d}$$

$$0 \le x_i \le u_{0,i}, x_i \in \mathbb{Z}, \qquad \forall i \in [1:n]. \tag{3.7e}$$

We summarize SFA in Algorithm 9 and sketch the idea here: SFA consists of several scaling phases. Each of its scaling phase is associated with a scaling parameter $s \in \mathbb{Z}$ and a flow $\boldsymbol{z}$.

Algorithm 9 Scaled Flow-improving Algorithm for integer (P2) (SFA)

1: **Input:** An instance of (3.7).
2: **Output:** An optimal solution $(\boldsymbol{x}^\star, \boldsymbol{y}^\star)$.
3: **Initialization:** $s \leftarrow \left\lceil \frac{B}{2n} \right\rceil, \boldsymbol{z} \leftarrow \boldsymbol{0}$.
4: **while** $s > 1$ **do**
5: $\quad (\boldsymbol{x}, \boldsymbol{y}) \leftarrow \mathrm{FAC}(s, \boldsymbol{z})$;
6: $\quad \boldsymbol{z} \leftarrow \max\{\boldsymbol{x} - s\boldsymbol{e}, \boldsymbol{0}\}$. ▷ Element-wise maximum operation.
7: $\quad s \leftarrow \left\lceil \frac{s}{2} \right\rceil$.
8: **end while**
9: **return** $\mathrm{FAC}(1, \boldsymbol{z})$.

Initially, we start with sufficiently large s and zero flow. In the s-scaling phase, we execute

Algorithm 10 which is adapted from the A-H algorithm (Algorithm 7). It starts with initial flow $\boldsymbol{z}$. In the elementary iteration of Algorithm 10, we find the shortest path with respect to the unit incremental cost, and then send at most s units of flow along that path. A feasible solution for the original problem is obtained at the end of each s-scaling phase. Then we modify the solution so that it becomes a lower bound on some optimal solution for the optimal solution, decrease s by half, and start the next scaling phase. At the end, the optimal solution can be found with in the last s-scaling phase with $s = 1$.

The subroutine FAC$(1, \boldsymbol{z})$ is adapted from Algorithm 7 and it is summarized in Algorithm 10:

Algorithm 10 FlowAugmenting-Convex$(s, \boldsymbol{z})$ (FAC$(s, \boldsymbol{z})$)

1: **Input:** Step size s, initial pseudoflow $\boldsymbol{z}$.
2: **Output:** A solution $(\boldsymbol{x}, \boldsymbol{y})$.
3: **Initialization:** Set $\boldsymbol{x} = \boldsymbol{z}$ and compute the horizontal flow $\boldsymbol{y}$, and the imbalance $e(i), i \in [1 : n]$ based on the flow balance constraints.
4: **for** $i = 1, 2, \cdots, n$ **do**
5: **while** $e(i) < 0$ **do**
6: Find the path from node 0 to node i with the minimum unit incremental cost, break ties with the largest index rule;
7: Augment at most s units along the path; if the path has a remaining capacity $\delta < s$, augment δ units; update the cost of each arc; remove arcs with capacity zero.
8: **end while**
9: **end for**
10: **return** the flow $(\boldsymbol{x}, \boldsymbol{y})$.

Even though SFA is also a scaling-based algorithm, it is different from and more efficient than the capacity scaling algorithm. We illustrate the distinction between the two algorithms below. At a high level, the loop invariants of the two algorithms are different. The capacity scaling algorithm maintains a pseudoflow that is dual feasible. Moreover, it also maintains the dual variables in order to restore the loop invariant. On the other hand, as we will see in the next section, SFA maintains a pseudoflow which is a lower bound on some optimal solution, which is not necessarily dual feasible. A simple element-wise maximization operation (Step 6 of Algorithm 9) restores the loop invariant between two consecutive scaling phases.

The two algorithms are also different in their elementary iterations in the while loop. SFA finds the shortest-path w.r.t the unit incremental cost, instead of the criterion of average incremental cost of the next s units (in the s-residual network) in the capacity scaling algorithm.

Finally, SFA augments at most s units of flow in each elementary iteration, while the capacity scaling algorithm sends exactly s units of flow. In this way, SFA obtains a (primal) feasible solution at the end of each scaling phase while the capacity scaling algorithm finds a dual feasible solution that may not be primal feasible.

SFA is more efficient as it handles the residual graph in a more dedicated and implicit manner. It avoids implementation of the sophisticated single-source shortest-path algorithms since it does not maintain the dual variables.

3.4.1 Correctness of SFA

In this section, we show that SFA finds the optimal solution of (3.7). In Algorithm 9, we try to maintain the following loop invariant at the beginning of each s-scaling phase:

- After Step 6 in Algorithm 9, the vector $\boldsymbol{z}$ is a lower bound on some optimal solution $(\boldsymbol{x}^\star, \boldsymbol{y}^\star)$ of the (P2) instance in the sense that $\boldsymbol{z} \leq \boldsymbol{x}^\star$.

Firstly, we show that the subroutine Algorithm 10 can find the optimal solution to any instance of (3.7) if $s = 1$ and if the initial pseudoflow $\boldsymbol{z}$ is a lower bound on the optimal solution. Here, we introduce the notation FAC(s, z, P) to indicate that Algorithm 10 is applied to an instance P of problem (3.7).

Proposition 4. *Suppose an instance P of problem (3.7) is feasible and $\boldsymbol{z}$ is a lower bound on an optimal solution $(\boldsymbol{x}^\star, \boldsymbol{y}^\star)$ of P in the sense that $\boldsymbol{z} \leq \boldsymbol{x}^\star$, then FAC$(1, \boldsymbol{z}, P)$ will return an optimal solution of P.*

Proof. The proof is straightforward. We add the lower bound $\boldsymbol{z}$ as a constraint to (3.7).

$$\min \quad \sum_{i=1}^{n} f_i(x_i) + \sum_{i=1}^{n-1} g_i(y_i) \tag{3.8a}$$

$$\text{s.t.} \quad y_i = \sum_{k=1}^{i} x_k - \sum_{k=1}^{i} d_i, \qquad \forall i \in [1:n], \tag{3.8b}$$

$$y_n = 0, \tag{3.8c}$$

$$0 \leq y_i \leq u_{i,i+1}, \qquad \forall i \in [1:n-1], \tag{3.8d}$$

$$0 \leq x_i \leq u_{0,i}, x_i \in \mathbb{Z}, \qquad \forall i \in [1:n] \tag{3.8e}$$

$$\boldsymbol{z} \leq \boldsymbol{x}. \tag{3.8f}$$

An optimal solution of (3.8) is also an optimal solution of P. Substituting the variable $\boldsymbol{x}$ by $\boldsymbol{t} = \boldsymbol{x} - \boldsymbol{z}$, we obtain a new instance $\bar{P}$ of (3.7), with slightly different parameters from P. Then sequence of flow augmentations the procedure FAC$(1, \boldsymbol{z}, P)$ is exactly the same as that of

FAC$(1, \mathbf{0}, \bar{P})$, which is the successive shortest path algorithm with unit step size. Suppose the flow on the vertical arcs in the output of FAC$(1, \mathbf{0}, \bar{P})$ is $\boldsymbol{t}^*$, then $\boldsymbol{x}^* = \boldsymbol{t}^* + \boldsymbol{z}$ is the vertical flow in some optimal solution of P. Hence, FAC$(1, \boldsymbol{z}, P)$ finds the optimal solution to P correctly. $\square$

Now we discuss how the invariant property can be maintained between two scaling phases. At the first period, the flow $\boldsymbol{z} = \mathbf{0}$ obviously satisfies the loop invariant. Notice that as long as (3.7) is feasible, the procedure FAC$(s, \boldsymbol{y})$ will always return a feasible solution. We argue that only the last increment in the last s-phase on each $x_i's$ may be incorrect. We can restore the invariant by executing Step 6 of Algorithm 9. In the following, we give a theorem proving the correctness of Algorithm 9. Here, we use $P(d)$ in FAC$(s, \boldsymbol{z}, P(d))$ to indicate the dependence of instance with demand d. Let $\boldsymbol{e}$ be the n-dimensional vector of all ones.

Theorem 2. *Suppose an instance $P(d)$ of problem (3.7) is feasible, $\boldsymbol{z}$ is a lower bound on some optimal solution $(\boldsymbol{x}, \boldsymbol{y})$ of $P(d)$ such that $\boldsymbol{z} \leq \boldsymbol{x}$, s is any positive integer, and $\boldsymbol{\alpha}^\star$ is the output of FAC$(s, \boldsymbol{z}, P(d))$, then there exists an optimal solution $(\boldsymbol{x}^\star, \boldsymbol{y}^\star)$ of (3.7), such that $\boldsymbol{x}^\star \geq \boldsymbol{\alpha}^\star - s\boldsymbol{e}$.*

Notice that the lower bound in Theorem 2 is stronger than that in the proximity theorem on the scaling framework for convex optimization with a separable objective in [67], which essentially states that the optimal solution $\boldsymbol{x}$ is bounded below by $\boldsymbol{z} - ns\boldsymbol{e}$ where n is the dimension of the problem.

We will prove Theorem 2 by contradiction. In the subroutine Algorithm 10, we break ties with the largest index rule. Without loss of generality, we assume that $\boldsymbol{z} = \mathbf{0}$ because we can always redefine variables by a linear transformation.. We assume that there are no upper bound capacity constraints since they can be interpreted as huge unit incremental cost in the cost function. The proof is organized as follows:

1. We assume that there exists some counterexample $P(d)$ of Theorem 2 and we construct a special counterexample $P(\bar{d})$ of Theorem 2 based on $P(d)$.
2. We prove several propositions that connect the outputs of FAC$(1, \mathbf{0}, P(\bar{d}))$ and FAC$(s, \mathbf{0}, P(\bar{d}))$.
3. We then derive a contradiction on the amount of flow augmented in the procedures FAC$(1, \mathbf{0}, P(\bar{d}))$ and FAC$(s, \mathbf{0}, P(\bar{d}))$.

Suppose there exists some instance $P(d)$ of (3.7) that violates Theorem 2. In FAC$(1, \mathbf{0}, P(d))$, we apply Algorithm 10 to $P(d)$ with step size 1. Let $(\boldsymbol{x}^t(d), \boldsymbol{y}^t(d))$ denote the pseudoflows of the vertical and horizontal arcs in the network at the t-th iteration of FAC$(1, \mathbf{0}, P(d))$, and $(\boldsymbol{x}^\star(d), \boldsymbol{y}^\star(d))$ be the final output of FAC$(1, \mathbf{0}, P(d))$.

Similarly, in FAC$(s, \mathbf{0}, P(d))$, we apply Algorithm 10 to $P(\boldsymbol{d})$ with step size s. Let $(\boldsymbol{\alpha}^t(d), \boldsymbol{\beta}^t(d))$ be the pseudoflow of the vertical and horizontal arcs in the network at the t-th iteration of FAC$(s, \mathbf{0}, P(d))$ and $(\boldsymbol{\alpha}^\star(d), \boldsymbol{\beta}^\star(d))$ be the final output of FAC$(s, \mathbf{0}, P(d))$.

Since $P(d)$ violates Theorem 2 and $\boldsymbol{x}^\star$ is an optimal solution of $P(d)$. There exists an index I such that

$$x_I^\star(d) < \alpha_I^\star(d) - s.$$

Let T be the greatest index t such that at the t-th iteration of FAC$(s, \mathbf{0}, P(d))$, $\alpha_I^t(d) < \alpha_I^\star(d)$. Let $\bar{d}$ be the demand following the flow balance constraints with $(\boldsymbol{\alpha}^{T+1}(d), \boldsymbol{\beta}^{T+1}(d))$ in the network, i.e., the demand that has been satisfied by the pseudoflow $(\boldsymbol{\alpha}^{T+1}(d), \boldsymbol{\beta}^{T+1}(d))$ after the T-th iteration. In FAC$(s, \mathbf{0}, P(d))$, we always select the deficit node (a node with demand) in the order of increasing indices. Therefore, $\bar{d}$ has the following structure,

$$\begin{aligned}
&\bar{d}_i = d_i, i \in [1 : p-1],\\
&\bar{d}_p \leq d_p,\\
&\bar{d}_i = 0, i \in [p+1; n].
\end{aligned}$$

where p is the index of the selected deficit node at the T-th iteration of FAC$(s, \mathbf{0}, P(d))$. We defined a "truncated" instance $P(\bar{d})$ with the same network structure and cost as $P(d)$ except the demand vector is $\bar{d}$. For instance $P(\bar{d})$, let's define $(\boldsymbol{x}^\star(\bar{d}), \boldsymbol{y}^\star(\bar{d}))$, $(\boldsymbol{x}^t(\bar{d}), \boldsymbol{y}^t(\bar{d}))$, and $(\boldsymbol{\alpha}^\star(\bar{d}), \boldsymbol{\beta}^\star(\bar{d}))$, $(\boldsymbol{\alpha}^t(\bar{d}), \boldsymbol{\beta}^t(\bar{d}))$ similarly. Since $P(\bar{d})$ has a truncated demand of $P(d)$, we have

$$(\boldsymbol{\alpha}^{T+1}(d), \boldsymbol{\beta}^{T+1}(d)) = (\boldsymbol{\alpha}^\star(\bar{d}), \boldsymbol{\beta}^\star(\bar{d})).$$

There exists an integer q such that at the q-th iteration of

$$(\boldsymbol{x}^q(d), \boldsymbol{y}^q(d)) = (\boldsymbol{x}^\star(\bar{d}), \boldsymbol{y}^\star(\bar{d})).$$

For index I, we have

$$x_I^\star(\bar{d}) = x_I^q(d) \leq x_I^\star(d) < \alpha_I^\star(d) - s = \alpha_I^\star(\bar{d}) - s.$$

where the second inequality is based on the fact that the pseudoflow is non-decreasing in Algorithm 12.

Hence, $P(\bar{d})$ also violates Theorem 2. Now we will focus on $P(\bar{d})$ to derive a contradiction. In the following, we omit the dependence of $\bar{d}$ to simplify the notation.

In the procedure FAC$(1, \mathbf{0}, P)$, let's denote the pseudoflows in the elementary iterations by

$$(\boldsymbol{x}^0, \boldsymbol{y}^0) = (\mathbf{0}, \mathbf{0}), (\boldsymbol{x}^1, \boldsymbol{y}^1), (\boldsymbol{x}^2, \boldsymbol{y}^2), \cdots, (\boldsymbol{x}^r, \boldsymbol{y}^r) = (\boldsymbol{x}^\star, \boldsymbol{y}^\star),$$

for some finite integer r. In the last iteration of FAC$(s, \mathbf{0}, P)$, i.e., the T-th iteration, we choose to send $s_1 = \alpha_I^{T+1} - \alpha_I^T \leq s$ units of flow to the deficit node p. We can remove the nodes $j > p$ since they has a demand $d_j = 0$ and they won't affect the solution of both FAC$(1, \mathbf{0}, P)$ and FAC$(s, \mathbf{0}, P)$. Hence, without loss of generality, we can assume the deficit node at the T-th iteration of FAC$(s, \mathbf{0}, P)$ is n. Based on the fact that we choose the path $0 \to I \to I+1 \to \cdots n$ at the T-th iteration, we claim that the unit incremental cost along this path is minimal among all the paths. In other words, we have the following result.

Proposition 5. *Let $(\boldsymbol{\alpha}^\star, \boldsymbol{\beta}^\star)$ be the final output of procedure FAC$(s, \mathbf{0}, P)$ and I be the selected index in the last elementary iteration. Then we have,*

- *For $j \in [1 : I-1]$,*

$$\nabla f_I(\alpha_I^\star - s) \leq \nabla f_j(\alpha_j^\star) + \sum_{k=j}^{I-1} \nabla g_k(\beta_k^\star). \tag{3.9}$$

- *For $j \in [I+1 : n]$,*

$$\nabla f_I(\alpha_I^\star - s) + \sum_{k=I}^{j-1} \nabla g_k(\beta_k^\star - s) < \nabla f_j(\alpha_j^\star). \tag{3.10}$$

where $\nabla f(x) = f(x+1) - f(x)$ denotes the unit incremental cost of an arc.

Proof. By the definition of T, we send s_1 units of flow along the path $0 \to I \to I+1 \to \cdots n$ at the T-th iteration of FAC$(s, \mathbf{0}, P(\bar{d}))$. Since the maximum step size is s, $s_1 \leq s$. Therefore,

$$\alpha_I^T = \alpha_I^{T+1} - s_1 = \alpha_I^\star - s_1 \leq \alpha_I^\star - s.$$

Now let's consider the cost of all other paths from node 0 to node n. For $j \in [1 : I-1]$, the unit incremental cost of the path $0 \to j \to \cdots I \to \cdots n$ should be no less than the cost of the selected path $0 \to I \to I+1 \to \cdots n$, and it yields

$$\nabla f_I(\alpha_I^\star - s) \leq \nabla f_I(\alpha_I^T) \leq \nabla f_j(\alpha_j^\star) + \sum_{k=j}^{I-1} \nabla g_k(\beta_k^\star)$$

where the first inequality is due to $\alpha_I^T = \alpha_I^\star - s_1 \geq \alpha_I^\star - s$ and the convexity of the cost function.

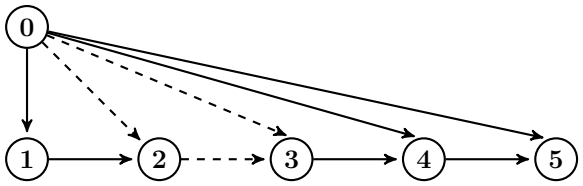


Figure 3.7: Illustrating the relation of the unit incremental cost. In this network, $I = 3$. At iteration t, we prefer the path $0 \to 3 \to 4 \to 5$ to the path $0 \to 2 \to 3 \to 4 \to 5$. The costs on path $3 \to 4 \to 5$ can be ignored as it appears in both path.

For $j \in [I+1 : n]$, the path $0 \to I \to I+1 \to \cdots j \to \cdots n$ has a lower unit incremental cost than the path $0 \to j \to \cdots n$, and it yields

$$\nabla f_I(\alpha_I^\star - s) + \sum_{k=I}^{j-1} \nabla g_k(\beta_k^\star - s) \leq \nabla f_I(\alpha_I^T) + \sum_{k=I}^{j-1} \nabla g_k(\beta_k^\star - s_1) < \nabla f_j(\alpha_j^\star).$$

where the first inequality is due to $\alpha_I^\star = \alpha_I^\star - s_1 \geq \alpha_I^\star - s$.

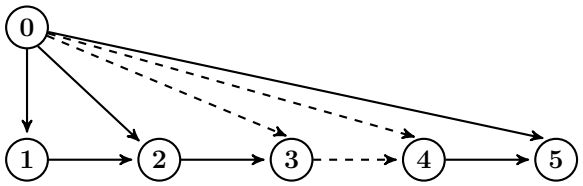


Figure 3.8: Illustrating the relation of the unit incremental cost. In this network, $I = 3$. At iteration t, we prefer the path $0 \to 3 \to 4 \to 4$ to the path $0 \to 4 \to 5$. The costs on path $4 \to 5$ can be ignored as it appears in both path.

□

In the following, we prove a proposition that connects $(\boldsymbol{x}^t, \boldsymbol{y}^t)$ and $(\boldsymbol{\alpha}^\star, \boldsymbol{\beta}^\star)$.

Proposition 6. *For any index $1 \leq j < n$, if $x_{j+1}^t < \alpha_{j+1}^\star$ and the value of y_j^t, i.e., the flow on the arc $(j, j+1)$, is increased from $\beta_j^\star$ to $\beta_j^\star + 1$ at the t-th iteration, then there exists some index $q \leq j$ such that*

$$\nabla f_{j+1}(x_{j+1}^t) > \nabla f_q(\alpha_q^\star) + \sum_{k=q}^{j} \nabla g_k(\beta_k^\star)$$

where $\nabla f(x) = f(x+1) - f(x)$ denotes the unit incremental cost of an arc.

Proof. We will prove this claim by an induction on the node indices $j \in [1 : n-1]$. Notice that when we start to satisfy the demand of node $j+1$, the flow on the arc $(j, j+1)$ is always zero,

i.e., $y_j = 0 \leq \beta_j^\star$. In other words, the path $0 \to (j+1) \to (j+2) \to \cdots$ becomes available since then. Let m be the deficit node at the t-th iteration. When we increase the value of y_j^t from $\beta_j^\star$ at iteration t, since this path $0 \to (j+1) \to (j+2) \to \cdots$ is available and is not selected, it must has greater unit incremental cost than the selected path.

1. Base case $j = 1$. Notice that $\beta_1^\star = \alpha_1^\star - \bar{d}_1$. At iteration t, we increase y_1^t from $\beta_1^\star$ to $\beta_1^\star + 1$, i.e., we increase x_1^t from $\alpha_1^\star$ to $\alpha_1^\star + 1$. Since we prefer the path $0 \to 1 \to 2 \cdots \to m$ to the path $0 \to 2 \to 3 \cdots \to m$, we have

$$\nabla f_2(x_2^t) > \nabla f_1(\alpha_1^\star) + \nabla g_1(\beta_1^\star).$$

   Hence, $q = 1$ in this case.

2. Now suppose that the statement is true for indices $1, 2, \cdots, j$, we will prove it is also true for index $j+1$ with $j+1 \leq n$. Suppose that we choose to increase x_i^t with $i \leq j+1$ at iteration t such that y_{j+1}^t is increased from $\beta_{j+1}^\star$ to $\beta_{j+1}^\star + 1$. Let's define a special index

$$J = \arg\max\{k : x_k^t > \alpha_k^\star, k \leq j+1\}.$$

   If such J does not exist, it implies $x_k^t \leq \alpha_k^\star$ for $k \in [1 : j+1]$. Then it means $x_k^t = \alpha_k^\star$ for $k \in [1 : j+1]$. Since we prefer the path $0 \to i \to i+1 \cdots j+2 \cdots \to m$ to the path $0 \to j+2 \to j+3 \cdots \to m$,

$$\begin{aligned} \nabla f_{j+2}(x_{j+2}^t) &> \nabla f_i(x_q^t) + \sum_{k=i}^{j+1} \nabla g_k(y_k^t) \\ &= \nabla f_i(\alpha_i^\star) + \sum_{k=i}^{j+1} \nabla g_k(\beta_k^\star). \end{aligned}$$

   The index $q = i$ in this case.

   If J is well-defined, we can make some observation about y_l^t and β_l^t, the flow on the horizontal arcs $(l, l+1)$ for $l \in [J, J+2]$. By the definition of J,

$$x_k^t \leq \alpha_k^\star,$$

   for $k \in [J+1 : j+1]$. Let's expand $y_{j+1}^t = \beta_{j+1}^\star$,

$$\sum_{k=1}^{j+1} x_k^t - \sum_{k=1}^{j+1} d_k = y_{j+1}^t = \beta_{j+1}^\star = \sum_{k=1}^{j+1} \alpha_k^\star - \sum_{k=1}^{j+1} d_k.$$

Since the tails $x_k^t \le \alpha_k^\star$ for $k \in [J+1 : j+1]$, we have

$$\sum_{k=1}^{l} x_k^t \ge \sum_{k=1}^{l} \alpha_k^\star,$$

for $l \in [J : j+1]$, which implies

$$y_l^t \ge \beta_l^\star, \tag{3.11}$$

for $l \in [J : j+1]$. Moreover, if $x_l^t < \alpha_l^\star$, the inequality $y_l^t \ge \beta_l^\star$ becomes strict.

Now let's discuss the chosen path $0 \to i \to i+1 \to \cdots \to m$ through four cases of possible i.

(a) $i = J$. Since we prefer the path $0 \to J \to J+1 \cdots j+2 \cdots \to m$ to the path $0 \to j+2 \to j+3 \cdots \to m$,

$$\begin{aligned}\nabla f_{j+2}(x_{j+2}^t) &> \nabla f_J(x_J^t) + \sum_{k=J}^{j+1} \nabla g_k(y_k^t) \\ &\ge \nabla f_J(\alpha_J^\star) + \sum_{k=J}^{j+1} \nabla g_k(\beta_k^\star),\end{aligned}$$

where the second inequality is due to the convexity of the cost function. The index $q = J$ in this case.

(b) $i < J$. Since $x_J^t > \alpha_J^\star$, there is an earlier iteration $p < t$ at which we choose to increment x_J^p from $\alpha_J^\star$ to $\alpha_J^\star + 1$. At iteration p, we prefer the path $0 \to J \to J+1 \to \cdots$ to the path $0 \to i \to \cdots J \to \cdots$,

$$\begin{aligned}\nabla f_J(\alpha_J^\star) = \nabla f_J(x_J^p) &\le \nabla f_j(x_i^p) + \sum_{k=i}^{J-1} \nabla g_k(y_k^p) \\ &\le \nabla f_i(x_i^t) + \sum_{k=i}^{J-1} \nabla g_k(y_k^t),\end{aligned}$$

where the second inequality is due to the fact that the pseudoflow is non-decreasing throughout the procedure FAC$(1, \mathbf{0}, P(\bar{d}))$. At iteration t, we prefer the path $0 \to$

$i \to \cdots j+2 \to \cdots m$,

$$\begin{aligned}\nabla f_{j+2}(x_{j+2}^t) &> \nabla f_i(x_i^t) + \sum_{k=i}^{j+1}(\nabla g_k(y_k^t)) \\ &\geq \nabla f_i(x_i^t) + \sum_{k=i}^{J-1} \nabla g_k(y_k^t) + \sum_{k=J}^{j+1} \nabla g_k(y_k^t) \\ &\geq \nabla f_J(\alpha_J^\star) + \sum_{k=J}^{j+1} \nabla g_k(\beta_k^\star).\end{aligned}$$

Hence, the index $q = J$ in this case.

(c) $i > J, x_i^t = \alpha_i^\star$.

By (3.11), $y_l^t \geq \beta_l^\star$ for $l \in [i : j+1]$. Since we prefer the path $0 \to i \to \cdots j+2 \to \cdots m$,

$$\begin{aligned}\nabla f_{j+2}(x_{j+2}^t) &> \nabla f_i(x_i^t) + \sum_{k=i}^{j+1} \nabla g_k(y_k^t) \\ &\geq \nabla f_i(\alpha_i^\star) + \sum_{k=i}^{j+1} + \nabla g_k(\beta_k^\star).\end{aligned}$$

Hence, the index $q = i$ in this case.

(d) $i > J, x_i^t < \alpha_i^\star$.

By (3.11), $y_i^t > \beta_i^\star$. There must be an earlier iteration p at which we increase y_i^p from $\beta_i^\star$ to $\beta_i^\star + 1$. Moreover, since $x_i^p \leq x_i^t < \alpha_q^t$, by the induction hypothesis, there exists some $i' < i$ such that

$$\nabla f_i(y_i^p) > \nabla f_{i'}(\alpha_{i'}^\star) + \sum_{k=i'}^{i-1}(\nabla g_k(\beta_k^\star)).$$

At iteration t, we prefer the path $0 \to q \to \cdots j+2 \to \cdots m$,

$$\begin{aligned}
\nabla f_{j+2}(x_{j+2}^t) &> \nabla f_i(x_i^t) + \sum_{k=i}^{j+1} \nabla g_k(y_k^t) \\
&\geq \nabla f_i(x_i^p) + \sum_{k=i}^{j+1} \nabla g_k(\beta_k^\star) \\
&> \nabla f_{i'}(\alpha_{i'}^\star) + \sum_{k=i'}^{i-1} \nabla g_k(\beta_k^\star) + \sum_{k=i}^{j+1} \nabla g_k(\beta_k^\star) \\
&= \nabla f_{i'}(\alpha_{i'}^\star) + \sum_{k=i'}^{j+1} \nabla g_k(\beta_k^\star).
\end{aligned}$$

Hence, the index $q = i'$ in this case.

We prove the induction step and the proof is complete.

□

An immediate corollary from the last proposition is

Proposition 7. *If $x_I^\star < \alpha_I^{(s)} - s$, for any elementary iteration t of FAC$(1, \mathbf{0}, P(\bar{d}))$, we have*

$$\sum_{k=1}^{I-1} x_k^t \leq \sum_{k=1}^{I-1} \alpha_k^\star. \tag{3.12}$$

Proof. We proof this by contradiction. Assume that there is some iteration such that $y_{I-1} > \beta_{I-1}^\star$, there must be an iteration t at which we increase y_{I-1}^t from $\beta_{I-1}^\star$ to $\beta_{I-1}^\star + 1$. Since $x_I^t < \alpha_I^t - s$, there exists a $q < I$ such that

$$\nabla f_I(x_I^t) > \nabla f_q(\alpha_q^\star) + \sum_{k=q}^{I-1} \nabla g_k(\beta_k^\star),$$

which contradicts (3.9) since $\nabla f_I(x_I^t) \leq \nabla f_I(\alpha_I^\star - s)$. Hence,

$$\sum_{k=1}^{I-1} x_k^t - \sum_{k=1}^{I-1} d_k = y_{I-1}^t \leq \beta_k^\star = \sum_{k=1}^{I-1} \alpha_k^\star - \sum_{k=1}^{I-1} d_k,$$

which is,

$$\sum_{k=1}^{I-1} x_k^t \leq \sum_{k=1}^{I-1} \alpha_k^\star.$$

□

Now we are ready to prove Theorem 2.

Proof of Theorem 2. We prove it by contradiction. We claim that if $x_I^\star < \alpha_I^{(s)} - s$, then for any iteration t of FAC$(1, \mathbf{0}, P(\bar{d}))$ and $j \in [I+1:n]$, we have

$$x_j^t \le \alpha_j^\star. \tag{3.13}$$

We prove the claim above by an induction on

$$(\boldsymbol{x}^0, \boldsymbol{y}^0) = (\mathbf{0}, \mathbf{0}), (\boldsymbol{x}^1, \boldsymbol{y}^1), (\boldsymbol{x}^2, \boldsymbol{y}^2), \cdots, (\boldsymbol{x}^r, \boldsymbol{y}^r) = (\boldsymbol{x}^\star, \boldsymbol{y}^\star).$$

The base case is trivial since we start with initial flow $\mathbf{0}$. Suppose for $j \in [I+1:n]$, we have $x_j^t \le \alpha_j^\star$. Since $\sum_{k=1}^{I-1} x_k^t \le \sum_{k=1}^{I-1} \alpha_k^\star$, we have $y_l^t \le \beta_l^\star - s$ for $l \in [I:n]$. Hence, for any $j \in [I+1:n]$ we have

$$\begin{aligned}
&\nabla f_I(x_I^t) + \sum_{k=I}^{j-1} \nabla g_k(y_k^t) \\
&\le \nabla f_I(\alpha_I^\star - s) + \sum_{k=I}^{j-1} \nabla g_k(\beta_k^\star - s) \\
&< \nabla f_j(\alpha_j^\star),
\end{aligned}$$

where the last inequality is due to (3.10).

Hence, if $x_j^t = \alpha_j^\star$, we will prefer the path $0 \to I \to \cdots j \cdots n$ to the path $0 \to j \to \cdots n$, i.e., we would not increase x_j^t at this iteration and hence $x_j^t = x_j^{t+1}$. Now that we prove the induction step, we can conclude that the induction statement is true for the last iteration $(\boldsymbol{x}^\star, \boldsymbol{y}^\star)$.

Combining (3.12) with (3.13), at iteration r we have,

$$\begin{aligned}
\sum_{k=1}^n \bar{d}_k = \sum_{k=1}^n x_k^\star &= \sum_{k=1}^{I-1} x_k^\star + x_I^\star + \sum_{k=I+1}^n x_k^\star \\
&< \sum_{k=1}^{I-1} \alpha_k^\star + \alpha_I^\star - s + \sum_{k=I+1}^n \alpha_k^\star \\
&= \sum_{k=1}^n \alpha_k^\star - s \\
&= \sum_{k=1}^n \bar{d}_k - s,
\end{aligned}$$

which is a contradiction. The proof is complete. □

3.4.2 Complexity of SFA

The following proposition states the complexity of SFA.

Proposition 8. *The running time of Algorithm 9 is* $O(n^2 \log \frac{B}{n})$ *for integer (P2) and* $O(n^2 \log \frac{B}{n\epsilon})$ *for continuous (P2).*

Proof. To analyze the complexity, notice that FAC$(s, \boldsymbol{z})$ is called for at most $\lceil \log \frac{B}{2n} \rceil$ times. Suppose in the j-th call of FAC$(s, \boldsymbol{z})$ we obtain the flow vector $\boldsymbol{z}_j$ with step size s_j.

In the $(j+1)$-th call, we start with $\boldsymbol{z}_{j+1} = \max\{\boldsymbol{z}_j - s_j \boldsymbol{e}, \boldsymbol{0}\}$ and use step size $s_{j+1} = \frac{1}{2} s_j$. The preprocessing step takes $O(n)$ time. The flow is augmented several times as described in Step 7 of Algorithm 10. The size of the increments can be either s or less than s.

- Type A: the size of the increment is strictly less than s.

  In this case, either one arc on the path has reached its capacity or the demand is satisfied. Because we have $2n-1$ arcs and n deficit nodes, the number of Type A increment is upper bounded by $3n$.

- Type B: the size of the increment is s.

  The number of Type B increment can be bounded by the remaining supply in the source node, i.e.,

  $$\frac{B - \boldsymbol{e}^\top \boldsymbol{z}_{j+1}}{s_{j+1}},$$

  where the numerator is the remaining supply of node 0 after initialization. Notice that

  $$\boldsymbol{e}^\top \boldsymbol{z}_{j+1} = \boldsymbol{e}^\top \max\{\boldsymbol{z}_j - s_j \boldsymbol{e}, \boldsymbol{0}\} \geq \boldsymbol{e}^\top (\boldsymbol{z}_j - s_j \boldsymbol{e}) = B - 2n s_j,$$

  Hence, the number of Type B increments is bounded by $\lceil \frac{2n s_j}{s_{j+1}} \rceil \leq 4n$.

Therefore, the number of augmentations within a single call of FAC$(s, \boldsymbol{z})$ is bounded by $O(n)$. If we maintains an n-dimension array with prefix-sum implementation to keep track of the cost for path $0 \to k \to k+1 \to \cdots \to i, k \in [1:i]$, it takes $O(n)$ time to find the path to node i with the minimum unit incremental cost. Moreover, it takes $O(n)$ time to find the minimum capacity along the path and update the unit incremental cost of all paths. Therefore, each call of FAC$(s, \boldsymbol{z})$ takes $O(n^2)$ time. Finally, the complexity of SFA for solving integer (P2) is $O(n^2 \log \frac{B}{n})$.

To obtain an ϵ-accurate solution to continuous (P2), we can scale the data by $\frac{1}{\epsilon}$ and the complexity of SFA for solving continuous (P2) is $O(n^2 \log \frac{B}{n\epsilon})$. □

3.5 Faster implementation of SFA for RAP-NC

RAP-NC is a special case of (P2) where the cost functions $g_i(\cdot)$'s over the horizontal arcs $(i, i+1), i \in [1 : n-1]$ are zero. In this section, we show that with the data structure segment tree introduced in Section 2, SFA can be sped up in solving RAP-NC and its complexity matches the best complexity in the literature for RAP-NC.

3.5.1 Transformation of RAP-NC to a minimum convex cost flow problem

Recall that RAP-NC can be transformed into a minimum convex cost flow problem defined on the directed network in Figure 3.9.

$$\min \quad \sum_{i=1}^{n} f_i(x_i) \tag{3.14a}$$

$$\text{s.t.} \quad y_1 = x_1 - a_1, \tag{3.14b}$$

$$y_i = y_{i-1} + x_i - (a_i - a_{i-1}), \qquad \forall i \in [2 : n-1], \tag{3.14c}$$

$$0 = x_{n-1,n} + x_n - (a_n - a_{n-1}), \tag{3.14d}$$

$$0 \le y_i \le b_i - a_i, \qquad \forall i \in [1 : n-1], \tag{3.14e}$$

$$0 \le x_i \le d_i, \qquad \forall i \in [1 : n]. \tag{3.14f}$$

In the directed graph $G = (N, A)$, there are $n+1$ nodes with node i representing activity i for $i = 1, 2, \cdots, n$ and an additional source node 0. The supply at source node 0 is $\sum_{i=1}^{n}(a_i - a_{i-1}) = a_n = B$. The demand at node i is $a_i - a_{i-1}$ for $i \in [1 : n]$. There is an arc from node 0 to node i for $i \in [1 : n]$. The flow x_i on this arc has an upper bound of d_i and incurs a cost of $f_i(x_i)$. There is an arc from node i to node $i+1$ for $i \in [1 : n-1]$. The flow $x_{i,i+1}$ on this arc has an upper bound of $b_i - a_i$ and has zero cost.

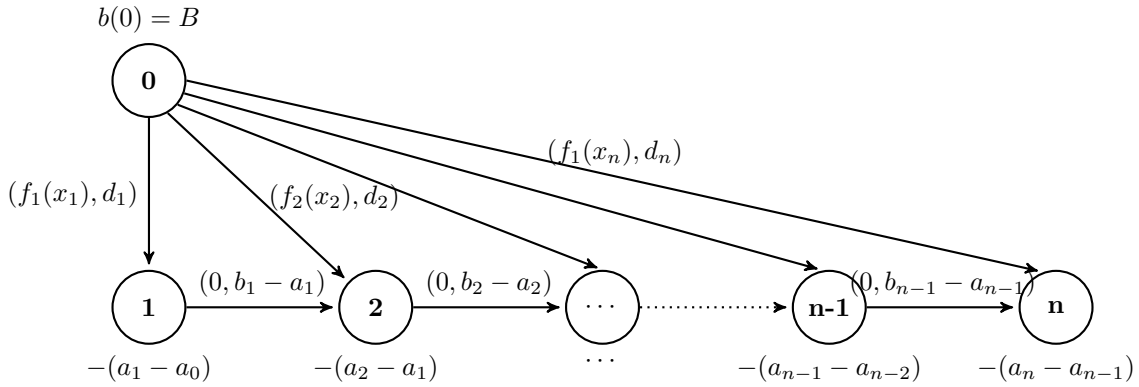


Figure 3.9: The network defined by (3.14). The pair (c, u) alongside each arc means the arc has a cost c and a capacity of u units.

Now we present SFA for integer (P1), i.e., DRAP-NC. It is a special implementation of SFA using data structure segment tree and red-black tree.

Algorithm 11 Scaled Flow-improving Algorithm for (P1) (SFA for DARP-NC)

Input: An instance of (3.14).
Output: An optimal solution $(\boldsymbol{x}^\star, \boldsymbol{y}^\star)$.
Initialization: $s \leftarrow \left\lceil \frac{B}{2n} \right\rceil, \boldsymbol{z} \leftarrow \boldsymbol{0}$.
while $s > 1$ **do**
    $(\boldsymbol{x}, \boldsymbol{y}) \leftarrow \text{FAC}(s, \boldsymbol{z})$;
    $\boldsymbol{z} \leftarrow \max\{\boldsymbol{x} - s\boldsymbol{e}, \boldsymbol{0}\}$. ▷ The maximum operation is element-wise.
    $s \leftarrow \left\lceil \frac{s}{2} \right\rceil$.
end while
return FAC-E$(1, \boldsymbol{z})$.

The subroutine Algorithm 12 is Algorithm 10 with use of segment tree and red-black tree.

Algorithm 12 FAC$(s, \boldsymbol{z})$ for DRAP-NC

Input: An instance of (3.14); step size s, initial pseudoflow $\boldsymbol{z}$.
Output: An solution $(\boldsymbol{x}, \boldsymbol{y})$.
Initialization: Initialize an empty red-black tree $F \leftarrow \emptyset$ to store the unit production cost;
Compute the residual network $G(\boldsymbol{z})$. ▷ $O(n)$ time.
Build a segment tree C with the residual capacity of the horizontal arcs; ▷ $O(n \log n)$ time
for $i = 1, 2, \cdots, n$ **do**
    Add a candidate shortest path: insert the path P_{ii} into the red-black tree F;
    while $e(i) < 0$ **do**
        Use F to find the shortest path, break ties with the largest index rule;
        Use the segment tree C to find the bottleneck capacity δ along the path and augment it along the path.
        Augment $\min\{\delta, s, -e(i)\}$ units of flow along the path, update the residual capacity and remove the paths in F with zero capacity.
    end while
end for
return the flow $(\boldsymbol{x}, \boldsymbol{y})$.

3.5.2 Complexity of SFA for (P1)

The following proposition establishes the complexity of SFA for (P1). Its proof is similar to that of SFA for (P2).

Proposition 9. *Algorithm 11 is correct and the running time is* $O(n \log n \log \frac{B}{n})$ *for DRAP-NC and* $O(n \log n \log \frac{B}{n\epsilon})$ *for RAP-NC.*

Proof. The correctness follows directly from Theorem 2. We analyze the complexity of Algorithm 11 below. First notice that the transformation from a DRAP-NC instance to a (P2) instance can be done in $O(n)$ time. To analyze the complexity of the algorithm, notice that FAC$(s, \boldsymbol{z})$ is called for at most $\lceil \log \frac{B}{2n} \rceil$ times. Suppose in the j-th call of FAC$(s, \boldsymbol{z})$ we obtain the flow vector $\boldsymbol{z}_j$ with step size s_j.

In the $(j+1)$-th call, we start with $\boldsymbol{z}_{j+1} = \max\{\boldsymbol{z}_j - s_j \boldsymbol{e}, \boldsymbol{0}\}$ and use step size $s_{j+1} = \frac{1}{2} s_j$. A segment tree of $n-1$ values is built and it takes $O(n \log n)$ time. The preprocessing step takes

$O(n)$ time. The flow is augmented several times as described in Step 11 of Algorithm 12. The size of the increments can be either s or less than s.

- Type A: the size of the increment is strictly less than s.

  In this case, either one arc on the path has reached its capacity or the demand is satisfied. Because we have $2n-1$ arcs and n deficit nodes, the number of Type A increment is upper bounded by $3n$.

- Type B: the size of the increment is s.

  The number of Type B increment can be bounded by the remaining supply in the source node, i.e.,

  $$\frac{B - \boldsymbol{e}^\top \boldsymbol{z}_{j+1}}{s_{j+1}},$$

  where the numerator is the remaining supply of node 0 after initialization. Notice that

  $$\boldsymbol{e}^\top \boldsymbol{z}_{j+1} = \boldsymbol{e}^\top \max\{\boldsymbol{z}_j - s_j \boldsymbol{e}, \boldsymbol{0}\} \geq \boldsymbol{e}^\top (\boldsymbol{z}_j - s_j \boldsymbol{e}) = B - 2ns_j,$$

  Hence, the number of Type B increments is bounded by $\lceil \frac{2ns_j}{s_{j+1}} \rceil \leq 4n$.

Therefore, the number of increments within a single call of FAC$(s, \boldsymbol{z})$ is bounded by $O(n)$. In each increment, the operations of the red-black tree and the segment tree take $O(\log n)$ time. There are only $O(1)$ calls of these operations except Step 11 of Algorithm 12. In Step 11, a path is removed whenever we exhaust the capacity of some arc of it. However, these removals will happen for at most n times during a single call of FAC$(s, \boldsymbol{z})$. Therefore, each call of FAC$(s, \boldsymbol{z})$ takes $O(n \log n)$ time. Finally, the complexity of SFA for solving DRAP-NC is $O(n \log n \log \frac{B}{n})$.

To obtain an ϵ-accurate solution to RAP-NC, we need to scale the data by $\frac{1}{\epsilon}$. Hence, the complexity of SFA for solving RAP-NC is $O(n \log n \log \frac{B}{n\epsilon})$. □

3.6 Computational experiments

3.6.1 Discrete resource allocation with nested bound constraints

In this section, we evaluate the performance of SFA on test instances of DRAP-NC. We compare the performance of our algorithm with DCA in Chapter 2 and MDA in [44]. The latter has the best worst-case time complexity for DRAP-NC. SFA achieves the same time complexity as well. Since all the three algorithms are able to solve instances with arbitrary convex functions and only require access to value oracles of these functions, we generate test instances with linear cost objectives, quadratic cost objectives, and three classes of more general objectives.

The parameters of the DRAP-NC test instances are generated in the same way as Chapter 2. We omit the procedure here.

Linear costs

In each instance with linear costs, the function $f_i(x)$ has the form $f_i(x) = p_i x$, where p_i is drawn from a uniform distribution over $[-1, 1]$. The parameter V_b is set to 100. We generate 14 different n values and 10 instances for each value of n. We summarize the performance of the three algorithms in Table 3.1 and Figure 3.10. The running time is averaged over 10 instances for each set of parameters.

Parameters	CPU time (s)			Parameters	CPU time (s)		
n	MDA	DCA	SFA	n	MDA	DCA	SFA
800	0.006	0.003	0.006	102400	0.597	0.109	0.119
1600	0.009	0.003	0.003	204800	1.278	0.209	0.256
3200	0.015	0.003	0.003	409600	2.671	0.465	0.547
6400	0.031	0.006	0.003	819200	5.883	0.937	1.381
12800	0.053	0.009	0.016	1638400	13.836	2.536	3.171
25600	0.131	0.022	0.028	3276800	28.599	6.214	7.142
51200	0.291	0.056	0.053	6553600	64.158	11.694	15.831

Table 3.1: Solution statistics of MDA, DCA, and SFA for instances with linear costs.

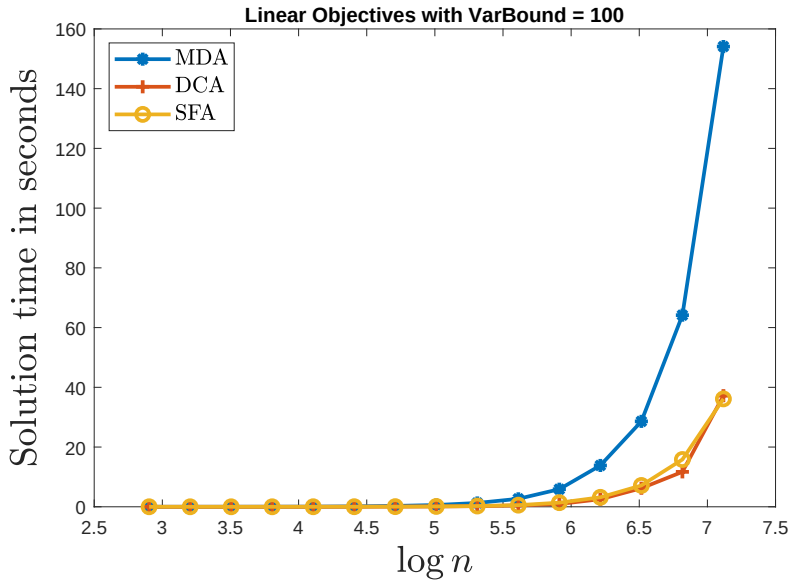


Figure 3.10: Solution time of MDA, DCA, and SFA for instances with linear costs.

Quadratic costs

In each instance with quadratic costs, the function $f_i(x)$ has the form $f_i(x) = p_i x^2 + q_i x$, where p_i is drawn from a uniform distribution over $[0, 1]$ and q_i is drawn from a uniform distribution over $[-1, 1]$. The parameter V_b is set to 100. We generate 12 different n values and 10 instances for each value of n.

We summarize the performance of the three algorithms in Table 3.2 and Figure 3.11 below. All statistics are obtained by averaging over 10 instances.

Parameters	CPU time (s)			Parameters	CPU time (s)		
n	MDA	DCA	SFA	n	MDA	DCA	SFA
1600	0.041	0.022	0.013	102400	5.473	3.679	1.575
3200	0.097	0.034	0.031	204800	12.258	10.358	3.062
6400	0.203	0.110	0.062	409600	29.355	25.200	7.203
12800	0.468	0.250	0.141	819200	71.690	59.557	17.064
25600	1.043	0.497	0.316	3276800	361.683	282.398	76.899
51200	2.445	1.304	0.696	6553600	886.682	699.724	156.630

Table 3.2: Solution statistics of MDA, DCA, and SFA for instances with quadratic costs.

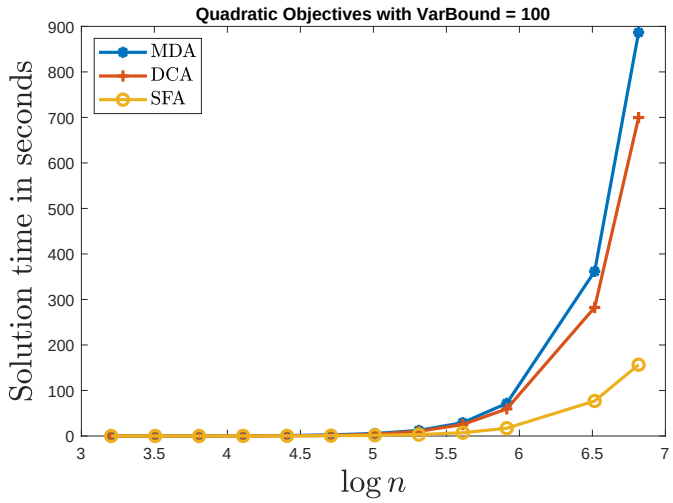


Figure 3.11: Solution time of DCA, MDA, and SFA for instances with quadratic costs.

Convex objective: [F]

[F] is a convex cost function that has the form

$$f_i(x) = \frac{x^4}{4} + p_i x.$$

In each instance, the function $f_i(x)$ has the form $f_i(x) = \frac{x^4}{4} + p_i x$, where p_i is drawn from a uniform distribution over $[-1, 1]$. The parameter V_b is set to 100. We generate 14 different n values ranging from 800 to 6 millions and 10 instances for each value of n. We summarize the performance of the three algorithms in Table 3.3 and Figure 3.12 below. All statistics are obtained by averaging over 10 instances.

Parameters	CPU time (s)			Parameters	CPU time (s)		
n	MDA	DCA	SFA	n	MDA	DCA	SFA
800	0.021	0.011	0.011	102400	4.945	2.199	1.384
1600	0.038	0.015	0.012	204800	11.163	5.693	3.038
3200	0.090	0.033	0.027	409600	26.132	12.044	7.098
6400	0.197	0.071	0.059	819200	62.631	32.200	15.854
12800	0.447	0.155	0.139	1638400	149.035	80.897	35.024
25600	0.989	0.375	0.296	3276800	363.447	148.142	79.684
51200	2.493	1.221	0.617	6553600	868.560	431.889	161.210

Table 3.3: Solution statistics of MDA, DCA, and SFA for instances with convex cost objectives [F].

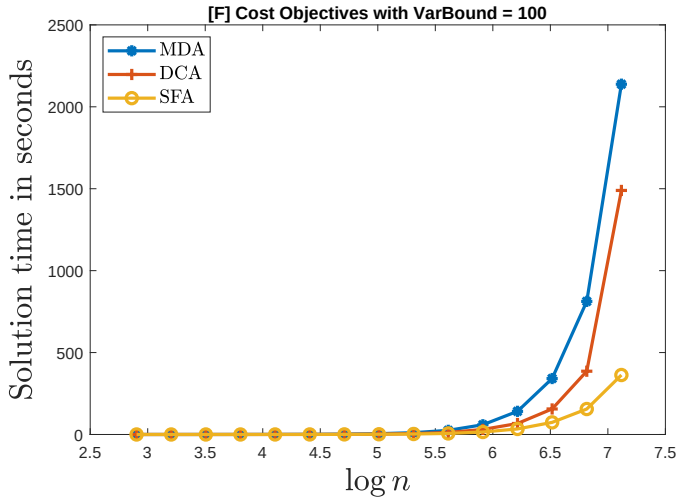


Figure 3.12: Solution time of MDA, DCA, and SFA for instances with convex cost objectives [F].

Convex objective [CRASH]

[CRASH] is a convex cost function that has the form

$$f_i(x) = k_i + \frac{p_i}{x}.$$

In each instance, the function $f_i(x)$ has the form $f_i(x) = k_i + \frac{p_i}{x}$, where p_i and k_i are drawn from a uniform distribution over $[0, 1]$. The parameter V_b is set to 100. We generate 14 different n values ranging from 800 to 6 millions and 10 instances for each value of n. We summarize the performance of the three algorithms in Table 3.4 and Figure 3.13 below.

Parameters	CPU time (s)			Parameters	CPU time (s)		
n	MDA	DCA	SFA	n	MDA	DCA	SFA
800	0.024	0.013	0.011	102400	5.829	2.665	1.504
1600	0.044	0.020	0.012	204800	12.982	7.333	3.151
3200	0.102	0.041	0.029	409600	30.030	18.617	7.005
6400	0.225	0.099	0.062	819200	72.228	40.721	16.859
12800	0.506	0.234	0.133	1638400	167.963	108.050	35.610
25600	1.131	0.539	0.290	3276800	402.043	237.597	76.527
51200	2.493	1.221	0.617	6553600	936.209	586.074	159.689

Table 3.4: Solution statistics of MDA, DCA, and SFA for instances with convex cost objectives [CRASH].

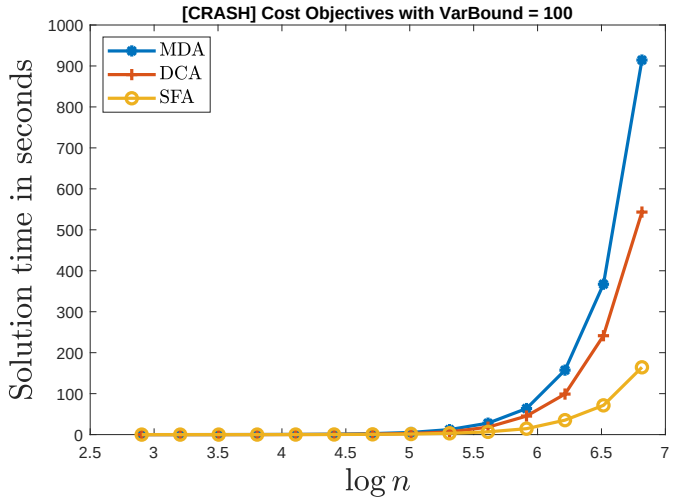


Figure 3.13: Solution time of MDA, DCA, and SFA for instances with convex cost objectives [CRASH].

Convex objective: [FUEL]

[FUEL] is a convex cost function that has the form

$$f_i(x) = p_i \times c_i \times (\frac{c_i}{x})^3.$$

In each instance, the function $f_i(x)$ has the form $f_i(x) = p_i \times c_i \times (\frac{c_i}{x})^3$, where p_i and c_i are drawn from a uniform distribution over $[0, 1]$. The parameter V_b is set to 100. We generate 14 different n values ranging from 800 to 6 millions and 10 instances for each value of n. We summarize the performance of the three algorithms in Table 3.5 and Figure 3.14 below.

Parameters	CPU time (s)			Parameters	CPU time (s)		
n	MDA	DCA	SFA	n	MDA	DCA	SFA
800	0.021	0.011	0.009	102400	5.234	2.424	1.432
1600	0.040	0.016	0.012	204800	12.115	6.704	3.244
3200	0.089	0.039	0.027	409600	27.381	16.761	6.698
6400	0.202	0.097	0.060	819200	62.899	39.872	15.158
12800	0.446	0.229	0.127	1638400	146.509	86.280	32.854
25600	1.077	0.454	0.294	3276800	346.476	222.655	69.194
51200	2.266	1.165	0.597	6553600	846.707	470.372	158.158

Table 3.5: Solution statistics of MDA, DCA, and SFA for instances with convex cost objectives [FUEL].

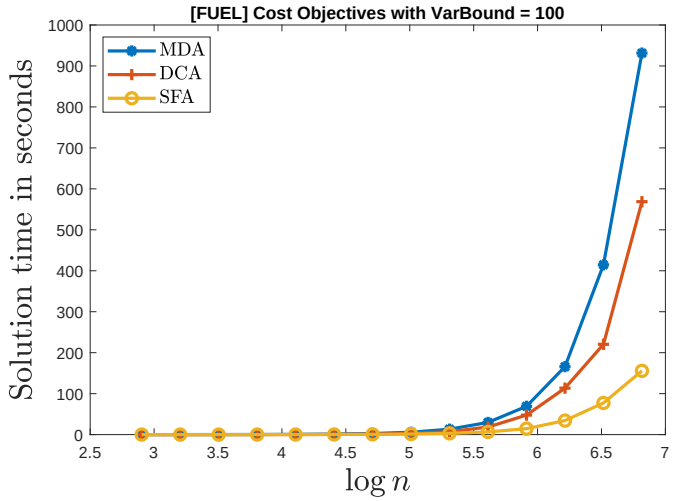


Figure 3.14: Solution time of MDA, DCA, and SFA for instances with convex cost objectives [FUEL].

It can be seen that for all test instances, the performance of SFA is better than MDA and DCA. Moreover, SFA can solve large size of instances with several millions of variables.

3.7 Conclusions

In this chapter, we study a minimum convex cost network flow problem on the dynamic lot size network. This problem appears in many applications, including dynamic lot-sizing, speed optimization and resource allocation with nested bound constraints. We develop a new scaling-based algorithm to solve the continuous problem as well as its restriction on integer flows. Our algorithm is efficient and it has best worst-case time complexity in the literature. In particular, for the resource allocation problem studied in Chapter 2, the key operations in this algorithm can be accelerated with data structures. The improved complexity matches the best result for that problem in the literature. We conduct extensive computational experiments to evaluate the performance of our algorithm on instances with five classes of benchmark convex cost objectives in the literature. Numerical results demonstrate the efficiency of our algorithm in solving large-sized instances.

Chapter 4

Switched linear systems

In this chapter, we study an NP-hard problem: the optimal control problem of discrete-time switched linear systems. Consider a switched linear system

$$x(k+1) = T_k x(k), \qquad T_k \in \Sigma,\ k = 0, 1, \ldots, \tag{4.1}$$

where $x(k)$ is an n-dimensional real vector that captures the system state at period k, the set Σ contains m given $n \times n$ real matrices, each of which describes the dynamics of a linear subsystem, and the initial vector $x(0)$ is a given n-dimensional real vector a. We are interested in the following optimization problem (P3)

> Given a switched linear system described in (4.1), a positive integer K, and a convex function $f : \mathbb{R}^n \to \mathbb{R}$, find a sequence of K matrices $T_0, T_1, \ldots, T_{K-1} \in \Sigma$ to maximize $f(x(K))$.

We show that (P3) is NP-hard for a pair of stochastic matrices or binary matrices. We propose a polynomial-time exact algorithm for the problem when all input data are rational and the given set of matrices Σ has the oligo-vertex property, a new concept we introduce. We derive a set of sufficient and easy to verify conditions for a set of matrices to have the oligo-vertex property. In particular, we show that a pair of 2×2 binary matrices has the oligo-vertex property. Finally, we conjecture that any pair of 2×2 real matrices has the oligo-vertex property.

4.1 Introduction

Many real-world systems exhibit significantly different dynamics under various modes or conditions, for example a manual transmission car operating at different gears, a chemical reactor under different temperatures and flow rates of reactants, and a group of cancer cells responding to different drugs. Such phenomena can be modeled under a unified framework of switched systems. A switched system is a dynamical system that consists of several subsystems and a rule that specifies the switching among the subsystems. Finding a switching rule to optimize the dynamics of a switched system under certain criteria has found numerous applications in power system operations, chemical process control, air traffic management, and medical treatment design [46, 47, 48, 49]. In this chapter, we study the following discrete-time switched linear system:

$$x(k+1) = T_k x(k), \qquad T_k \in \Sigma,\ k = 0, 1, \ldots, \tag{4.2}$$

where $x(k)$ is an n-dimensional real vector that captures the system state at period k, the set Σ contains m given $n \times n$ real matrices, each of which describes the dynamics of a linear

subsystem, and the initial vector $x(0)$ is a given n-dimensional real vector a. Such a system with switching only at fixed time instants appear in many practical applications, and is also employed to approximate the more complex dynamics of a continuous-time hybrid system with switching rules defined over the real line [46, 48].

We are interested in the following optimization problem (P3) related to the system in (4.2):

> Given a switched linear system described in (4.2), a positive integer K, and a convex function $f : \mathbb{R}^n \to \mathbb{R}$, find a sequence of K matrices $T_0, T_1, \ldots, T_{K-1} \in \Sigma$ to maximize $f(x(K))$.

One of such convex functions is the ℓ_p norm.

Example 2. *Consider a switched linear system consisting of two subsystems with system matrices* $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ *and* $B = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, *an initial vector* $a = (2,1)^\top$, *and* $K = 8$. *Figure 4.1 illustrates the trajectory of* $x(k)$ *under three switching sequences, with the final state* $x(8)$ *being* $(53,23)^\top$, $(58,41)^\top$, *and* $(71,41)^\top$, *respectively.*

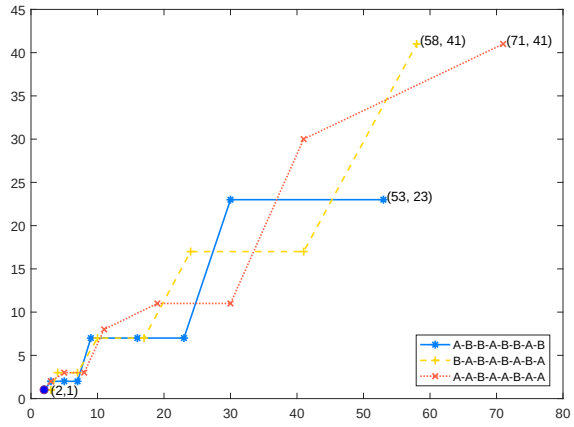


Figure 4.1: The trajectory of $x(k)$ under different matrix sequences

We give three examples below to illustrate the applications of Problem (P3) and its connection to other problems in control and optimization.

The first example is on design of treatment plans. Antibiotic resistance renders diseases that were once easily treatable dangerous infections, and has become one of the most pressing public health problems around the world. Several groups of researchers studied how to design sequential antibiotic treatment plans to restore susceptibility after bacteria develop resistance [89, 90].

They model the percentages of n genotypes of an enzyme produced by bacteria in a population with vector $x(k)$ after k periods of treatment, and a probability transition matrix to model the mutation rates among n genotypes under each antibiotic. The goal is to design a sequence of antibiotics to maximize the percentage of the wild type at the end of the treatment, which is sensitive to all antibiotics. The treatment design problem is equivalent to solve (P3) with $a = e_1$, a unit vector with the first component being 1 which denotes 100% wild type in the beginning, and $f(x(K)) = -e_1^\top x(K)$. In the same vein, (P3) can model the sequential therapy design problem for many other diseases when $x(k)$ describes related biometrics of a patient at period k and each matrix models the evolution of patient biometrics under a particular treatment [49].

The second example is the matrix mortality problem in control [91, 92]. Given a positive integer k, a set of matrices is said to be k-mortal if the zero matrix can be expressed as a product of k matrices in the set (duplication allowed). A set of matrices is said to be mortal if it is k-mortal for some finite k. The matrix mortality problem captures the stability of switched linear systems under certain switching rules. It can be shown that a finite set of $n \times n$ non-negative matrices is k-mortal if and only if the optimal objective value of (P3) is 0 with $a = \mathbf{1}$, $K = k$, and $f(x(K)) = -\mathbf{1}^\top x(K)$, where $\mathbf{1}$ is a n-dimensional vector with each component being 1.

The third example concerns the joint spectral radius of a set of matrices, an important quantity which has found many applications in wavelet functions, constrained coding, and network security management, etc [93]. The joint spectral radius of a finite set Σ of matrices [94] is defined as $\rho(\Sigma) = \limsup_{k\to\infty} \hat{\rho}_k(\Sigma, \|\cdot\|)$, where

$$\hat{\rho}_k(\Sigma, \|\cdot\|) = \max\{\|T_{k-1}T_{k-2}\dots T_0\|^{1/k} \mid T_j \in \Sigma, j = 0, \dots, k-1\} \tag{4.3}$$

and $\|\cdot\|$ is some matrix norm. If we select the matrix norm in (4.3) to be induced by the ℓ_p norm of a vector, then

$$(\hat{\rho}_k(\Sigma, \|\cdot\|))^k = \sup_{\|a\|_p=1} \max\{\|x(K)\|_p \mid (4.2)\}. \tag{4.4}$$

Observe that the inner optimization problem of the right-hand side of (4.4) is a special case of (P3) with the convex function $f(x) = \|x\|_p$. In general, let v^* be the optimal objective value of (P3) with $f(x) = \|x\|$ for some norm $\|\cdot\|$ and an initial vector a. Then $(v^*)^{1/k}$ provides a lower bound of the quantity $\hat{\rho}_k(\Sigma, \|\cdot\|)$.

A simple way to solve (P3) is to enumerate all possible matrix sequences, but such an approach quickly becomes impractical as m and K increase. Even for $m = 5$ and $K = 30$, we need to enumerate 5^{30} solutions, which is unrealistic in practice. Another general approach to solve (P3) is to formulate it as a mixed-integer nonlinear optimization problem, which can be

solved by global optimization solvers, but the problem size that can be handled by state-of-the-art commercial solvers is also limited. In addition, the time complexity of the tree-based search algorithms employed by these global solvers is difficult to analyze in general. In many applications, problem (P3) has to be solved repeatedly with different parameters, so it is of vital importance to have a fast algorithm for (P3).

We now introduce our results. We develop a simple dynamic programming algorithm to solve (P3) exactly, where several linear programs are solved at each iteration. One advantage of our algorithm is that it does not require any additional property of f such as smoothness or strong convexity. Our algorithm is very efficient in practice, based on computational results in Section 4.6. To analyze the time complexity of the algorithm, we assume that all input data are integers and the value of the convex function f can be queried through an oracle in constant time; we adopt the random-access machine [95] as the model of computation, in which each basic operation (addition, comparison, multiplication, etc.) is assume to take the same amount of time and the time complexity of an algorithm is the number of steps/operations required to execute the algorithm. We define the following notations that are useful for presenting the time-complexity results. Given a finite set Σ of $n \times n$ real matrices and a vector $a \in \mathbb{R}^n$, let

$$P_k(\Sigma, a) := \text{conv}(\{x(k) \mid x(k) = T_{k-1} \cdots T_0 a, T_j \in \Sigma, j = 0, \ldots, k-1\})$$

be the convex hull of all possible values of $x(k)$ in (4.2) for each integer $k \geq 0$. Let $N_k(\Sigma, a)$ be the number of extreme points of $P_k(\Sigma, a)$ and

$$N_k(\Sigma) = \sup_{a \in \mathbb{R}^n} \{N_k(\Sigma, a)\}.$$

We introduce the following concept for a set of matrices.

Definition 20. *A set of matrices Σ is said to have **the oligo-vertex property** if there exists $\alpha > 0$, positive integer k_0, and positive constant d such that $N_k(\Sigma) \leq \alpha k^d$ for any $k \geq k_0$.*

The oligo-vertex property of a set of matrices indicates the number of extreme points of $P_k(\Sigma, a)$ grows at most polynomially in k for any initial vector a, despite the number of possible values of $x(k)$ grows exponentially with k in general. With the Big-oh notation commonly used in computer science, the oligo-vertex property basically states that $N_k(\Sigma) = O(k^d)$ as $k \to \infty$ for some positive constant d.

Our contributions

We summarize the contributions of this chapter as follows.

1. We present a simple dynamic programming algorithm to solve (P3) exactly. Our algorithm does not require any additional property of f other than convexity. The running time of our algorithm is $O(m^2 n^{4.5}(\log n + \log M)\sum_{k=0}^{K-1} kN_k(\Sigma)^2)$, and can be reduced to $O(m \log m \sum_{k=0}^{K-1} N_k(\Sigma) + m\sum_{k=0}^{K-1} N_k(\Sigma)\log N_k(\Sigma))$ when $n = 2$, where M is the maximum absolute value of the entries of $A_1, \ldots, A_m$, and a.

2. We introduce the concept of the oligo-vertex property for a finite set of matrices, and show that our algorithm runs in polynomial time if the given set of matrices has the oligo-vertex property. To the best of our knowledge, this is the first time such a property is introduced for a set of matrices. We derive several sufficient conditions for a set of matrices to have this property. On the other hand, we show that (P3) is NP-hard for a pair of stochastic matrices or a pair of binary matrices, which implies that the oligo-vertex property is unlikely to hold for an arbitrary pair of $n \times n$ integer matrices unless P=NP.

3. Numerical experiments demonstrate that our proposed algorithm is very efficient in practice, and has significant advantages over state-of-the-art global optimization software in solving large size instances.

The oligo-vertex property we propose may be of independent interest to readers. We want to point out some similarities between the oligo-vertex property and another important property for a set of matrices that is also concerned with long matrix products—the finiteness property. A finite set Σ of matrices is said to have the finiteness property if the joint spectral radius $\rho(\Sigma)$ is equal to $(\rho_k(T_{k-1}T_{k-2}\ldots T_0))^{1/k}$ with $T_{k-1}, T_{k-2}, \ldots, T_0 \in \Sigma$ for some finite integer k, where $\rho(T)$ denotes the spectral radius of the matrix T. The finiteness property has been studied extensively for different families of matrices [96, 97], as it has many implications on stability and stabilization of switched systems. The finiteness property and the oligo-vertex property both hold for the following sets of matrices: commuting matrices, any finite set of matrices with at most one matrix's rank being greater than one [98], and a pair of 2×2 binary matrices [99]. We suspect that there might be a deeper connection between these two properties. Finally we pose several open questions on the oligo-vertex property at the end of this chapter.

The rest of the chapter is organized as follows. In Section 4.2, we review results related to the problem we study, with a main focus on computational complexity. In Section 4.3, we first prove that (P3) is NP-hard for a pair of stochastic matrices or binary matrices, and then introduce an exact algorithm for (P3) and analyze its time complexity for general n and $n = 2$. In Section 4.4, we introduce the oligo-vertex property and present several sufficient conditions for a set of matrices to have the oligo-vertex property. In Section 4.5, we prove that a pair of 2×2 binary matrices has the oligo-vertex property. We present some computational results in Section 4.6 and conclude in Section 4.7 with some open problems.

4.2 Related Work

Our problem aims to find the optimal switching rule of a discrete-time switched linear system without continuous control input. There have been a rich body of theoretical and computational results on optimal control of switched linear systems, such as finding optimal switching instants given a fixed switching sequence [100], minimizing the number of switches with known initial and final states [101], finding suboptimal policies [102], study of the exponential growth rates of the trajectories under different switching rules [103], and characterizing the value function of switched linear systems with linear and quadratic objectives [104]. We refer interested readers to the books [46, 48] and recent surveys [105, 106] for more details on switched linear systems. Finding the optimal switching sequence for a switched linear system also belongs to a broader class of problems called mixed-integer optimal control [107, 108] or optimal control of hybrid systems [109], which can be reformulated as a mixed-integer nonlinear optimization problem and solved by general mixed-integer optimization solvers.

We now survey computational complexity results related to the problem we study. Blondel and Tsitsiklis showed that the matrix mortality problem is undecidable for a pair of 48×48 integer matrices and the matrix k-mortality problem is NP-complete for a pair of $n \times n$ binary matrices with n being an input parameter [91]. The complexity of the matrix k-mortality problem is however unknown when the matrix dimension n is fixed. For the antibiotics time machine problem, Mira et al. used exhaustive search to find the optimal sequence of antibiotics for a small sized problem [89]. Tran and Yang showed that the antibiotics time machine problem is NP-hard when the number of matrices and the matrix dimension are both input parameters [110]. The antibiotics time machine can be also seen as a special finite-horizon discrete-time Markov decision process in which no state is observable. It has been shown in [111] that the finite-horizon unobservable Markov decision process is NP-hard. Therefore, our results identify several polynomially solvable cases of finite-horizon unobservable Markov decision processes. Computing the joint spectral radius for a finite set of matrices either exactly or approximately has been shown to be NP-hard [112], and has been a topic of active research [113, 114, 115]. The finiteness conjecture [96], which states that the finiteness property holds any set of real matrices, had remained a major open problem in the control community until early 2000s when a group of researchers showed that there exists a pair of 2×2 matrices that does not have the finiteness property [116, 117, 118]. The first constructive counterexample for the finiteness conjecture was proposed in [119]. The finiteness conjecture was shown to be true for a pair of 2×2 binary matrices [99] and a finite set of matrices with at most one matrix's rank being greater than one [98].

4.3 Computational Complexity

4.3.1 Notations

We first introduce some notations that will be used throughout this paper. Let $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{R}_+$, and $\mathbb{R}_-$ denote the sets of natural numbers (including 0), integers, real numbers, non-negative real numbers, and non-positive real numbers, respectively. We use x_i to denote the i-th component of a given vector x. Let $\|x\|_\infty$ and $\|T\|_\infty$ denote the infinity norm of vector x and matrix T, respectively. Given two positive integers i, j, let $[i:j]$ denote the set of integers $\{i, i+1, \ldots, j\}$ if $i \leq j$ and $\emptyset$ if $i > j$. Given two scalar functions f and g defined on some subset of real numbers, we write $f(x) = O(g(x))$ as $x \to \infty$, if there exist α and $x_0 \in \mathbb{R}$ such that $|f(x)| \leq \alpha|g(x)|$ for all $x \geq x_0$. Given a set S, let $|S|$ denote the cardinality of S, $\text{conv}(S)$ denote the convex hull of S, $\text{int}(S)$ denote the interior of S, and ∂S denote the boundary of S, respectively. Let $\text{ext}(S)$ denote the set of extreme points of a convex set S. Given a set $S \subseteq \mathbb{R}^n$ and a matrix $T \in \mathbb{R}^{n\times n}$, let $TS := \{Tx \mid x \in S\}$ be the image of S under the linear mapping defined by T. Let $\mathcal{Q}_i$ denote the i-th quadrant of the plane under the standard two-dimensional Cartesian system, for $i = 1, 2, 3, 4$. For example, $\mathcal{Q}_1 = \{x \in \mathbb{R}^2 \mid x_1 \geq 0, x_2 \geq 0\}$.

4.3.2 Complexity

Theorem 3. *(P3) is NP-hard for a pair of left (right) stochastic matrices and a linear function f.*

Proof. We prove the result based on a reduction from the 3-SAT problem. A 3-SAT problem asks whether there exists a truth assignment of several variables such that a given set of clauses defined over these variables, each with three literals, can all be satisfied. The 3-SAT problem is known to be NP-complete [120].

Given an instance of the 3-SAT problem with n variables $y_1, \ldots, y_n$ and m clauses $C_1, \ldots, C_m$, we construct an instance of (P3) with $\Sigma = \{A, B\}$ as follows. Matrices A and B are $m(2n+1) \times m(2n+1)$ adjacency matrices of two directed graphs G_A and G_B, respectively. The construction of G_A and G_B will be explained in detail below. We set the total number of periods $K = n$. Let $e_k \in \mathbb{R}^{m(2n+1)}$ be a vector with the k-th entry being 1 and all other entries being 0. We set $x(0) = \sum_{j=1}^{m} e_{(j-1)(2n+1)+1}$ and $f(x) = c^\top x$ with $c = -\sum_{j=1}^{m} e_{j(2n+1)}$. We claim that the 3-SAT instance is satisfiable if and only if the optimal objective value of the constructed instance of (P3) is $-m$.

Graph G_A is constructed as follows. It contains $m(2n+1)$ nodes, divided equally into m groups, each group corresponding to a clause. There is no arc between nodes in different groups. Let $u_{j,1}, u_{j,2}, \ldots, u_{j,2n+1}$ be the $2n+1$ nodes corresponding to clause j. The arcs among these

nodes are as follows. Node $u_{j,2n+1}$ has a self loop. There is an arc from $u_{j,l+1}$ to $u_{j,l}$ for $l = [1 : 2n]$ unless literal y_l is included in clause C_j; in that case, there will be an arc from node $u_{j,n+l+1}$ to node $u_{j,l}$. Graph G_B is constructed similarly with the same set of nodes. There is an arc from $u_{j,l+1}$ to $u_{j,l}$ for $l = [1 : 2n]$ unless literal y_l^c is included in clause C_j; in that case, there will be an arc from node $u_{j,n+l+1}$ to node $u_{j,l}$. An example for the clause $C_j = y_1 \vee y_3^c \vee y_4$ with a total of 4 variables is shown in Figure 4.2.

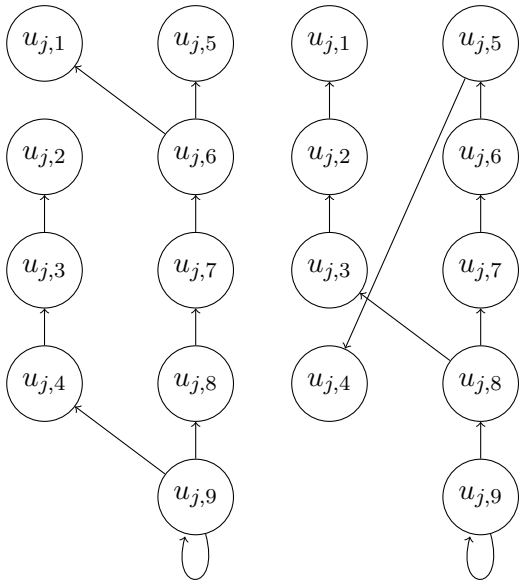


Figure 4.2: The nodes and arcs in G_A and G_B corresponding to the clause $C_j = y_1 \vee y_3^c \vee y_4$ with a total of 4 variables.

For $j \in [1 : n]$, let A^j (B^j) be the adjacency matrix of the component of G_A (G_B) corresponding to the j-th clause. Since each node has in-degree 1, each column of A^j (B^j) has exactly one entry being 1, so A^j (B^j) is a left stochastic matrix. We can associate each truth assignment of $y_1, \ldots, y_n$ with a sequence of matrices $T_0^j, \ldots, T_{n-1}^j$ with $T_t^j \in \{A^j, B^j\}$ for $t \in [0 : n-1]$. In particular, if y_t is true (false), then T_{t-1}^j is A (B). Consider the product

$$[0, \cdots, 0, -1] T_{n-1}^j T_{n-2}^j \cdots T_0^j \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

It can be verified that this product is $-1(0)$ if any only if the truth assignment of $y_1, \ldots, y_n$

makes clause j satisfied (unsatisfied).

Order the nodes of G_A or G_B lexicographically, i.e.,

$$u_{1,1}, u_{1,2}, \ldots, u_{1,2n+1}, u_{2,1}, \ldots, u_{2,2n+1}, \ldots, u_{m,2n+1}.$$

Let A and B be the adjacency matrix of G_A and G_B, respectively. Then both A and B are block diagonal matrices with m blocks of $(2n+1) \times (2n+1)$ matrices. In particular,

$$A = \begin{bmatrix} A^1 & & & \\ & A^2 & & \\ & & \ddots & \\ & & & A^m \end{bmatrix}, B = \begin{bmatrix} B^1 & & & \\ & B^2 & & \\ & & \ddots & \\ & & & B^m \end{bmatrix}. \tag{4.5}$$

Both A and B are left stochastic matrices. When $x(0) = \sum_{j=1}^{m} e_{(j-1)(2n+1)+1}$, $c = -\sum_{j=1}^{m} e_{j(2n+1)}$, $T_t \in \{A, B\}$ for $t \in [0 : n-1]$,

$$c^\top T_{n-1} \ldots T_0 x(0) = \sum_{j=1}^{m} [0, \cdots, 0, -1] T_{n-1}^j T_{n-2}^j \cdots T_0^j \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Therefore, there exists a truth assignment such that the 3-SAT instance is satisfied if and only if the optimal objective value of the constructed instance of (P3) is $-m$. This reduction is done in time polynomial in m and n.

To prove that (P3) is NP-hard for a pair of right stochastic matrices, we can construct an instance of (P3) in a similar way to the case of left stochastic matrices and show that there exists a truth assignment such that the 3-SAT instance is satisfied if and only if the optimal objective value of the constructed instance is $-m$. In particular, we let $x(0) = -\sum_{j=1}^{m} e_{j(2n+1)}$ (the vector c in the instance of (P3) with left stochastic matrices above), $f(x) = c^\top x$ with $c = \sum_{j=1}^{m} e_{(j-1)(2n+1)+1}$ (the initial vector $x(0)$ in the instance of (P3) with left stochastic matrices above), and the two matrices be the transpose of the two matrices A and B defined in (4.5). □

Since the matrices constructed in the proof of Theorem 3 are also binary matrices, we have the following result.

Corollary 1. *(P3) is NP-hard for a pair of binary matrices and a linear function f.*

4.3.3 The Algorithm

In this section, we present a simple forward dynamic programming algorithm to solve (P3) exactly, described in Algorithm 13. The critical step of Algorithm 13 is Step 6, which constructs E_k, the set of extreme points of $P_k(\Sigma, a)$, sequentially for $k = 0, 1, \ldots, K$.

Algorithm 13 A forward dynamic programming algorithm to solve (P3).

1: **Input:** Matrices $\Sigma = \{A_1, \ldots, A_m\} \in \mathbb{Z}^{n\times n}$, initial vector $a \in \mathbb{Z}^n$, value oracle f, and positive integer K.
2: **Output:** A sequence of matrices $T_0, \ldots, T_{K-1} \in \Sigma$ that maximize $f(T_{k-1}T_{k-2}\cdots T_0 a)$.
3: **Initialize:** Set $E_0 = \{a\}$.
4: **for** $k = 0, 1, \ldots, K-1$ **do**
5: Set $F_k^i = A_i E_k$ for $i = 1, \ldots, m$.
6: For each point $x \in \cup_{i=1}^m F_k^i$, check if x is an extreme point of $\text{conv}(\cup_{i=1}^m F_k^i)$, by solving a linear program. Let E_{k+1} be the set of all extreme points of $\text{conv}(\cup_{i=1}^m F_k^i)$.
7: **end for**
8: Find an $x^*(K) \in \arg\max\{f(x) \mid x \in E_K\}$ by enumeration.
9: Retrieve the optimal matrix sequence $T_{K-1}, T_{K-2}, \ldots, T_0$ from $x^*(K)$.

We specify the details of Step 6 later. In fact, Step 6 can be any algorithm that takes a set of points S as input and output $\text{ext}(\text{conv}(S))$. There are several efficient algorithms the construct the convex hull of a set of points on the plane, more efficient than linear programs. It is, however, difficult to construct $\text{conv}(S)$ efficiently in higher dimensional space. The correctness of Algorithm 13 follows directly from the proposition below.

Proposition 10. *Algorithm 13 solves (P3) correctly.*

Proof. First it is not difficult to show by induction that the set E_k constructed in Algorithm 13 is the set of extreme points of $P_k(\Sigma, a)$ for each $k \in [0 : K]$. Since maximizing a convex function f over a finite set S is equivalent to maximizing f over $\text{conv}(S)$ as well as maximizing f over $\text{ext}(\text{conv}(S))$ [121], (P3) is equivalent to $\max\{f(x) \mid x \in P_K(\Sigma, a)\} = \max\{f(x) \mid x \in E_K\}$. Then the result follows. □

Remark 1. *The fact that we are maximizing a convex function in the objective is critical for the correctness of Algorithm 13. If we minimize $f(x(K))$ in (P3) instead, then Algorithm 13 will not give the correct optimal solution in general.*

We now specify the linear program in Step 6 of Algorithm 13. Given a finite set $S = \{p^1, \ldots, p^l\} \subseteq \mathbb{R}^n$, checking if a point $p^j \in S$ is an extreme point of $\text{conv}(S)$ can be done by

solving the linear program below:

$$v^* = \max_{z,z_0} \ (p^j)^\top z - z_0 \tag{4.6a}$$

$$\text{s.t.} \ (p^i)^\top z - z_0 \le 0, \ i = 1, \ldots, l, i \ne j \tag{4.6b}$$

$$(p^j)^\top z - z_0 \le 1. \tag{4.6c}$$

Problem (4.6) is always feasible and bounded. Suppose $((z^*)^\top, z_0^*)$ is an optimal solution and v^* is the corresponding objective value. If $v^* > 0$, then we find a hyperplane $(z^*)^\top x = z_0^*$ that separates p^j and the set $S \setminus \{p^j\}$, and p^j is an extreme point of $\text{conv}(S)$. Otherwise p^j is not an extreme point of $\text{conv}(S)$. Problem (4.6) can be solved by various interior point methods in polynomial time, for example Karmarkar's algorithm. Recall that M is the maximum absolute value of the entries of $A_1, \ldots, A_m$, and a.

Proposition 11. *If Karmarkar's algorithm is employed to solve the linear programs at Step 6, the running time of Algorithm 13 is* $O(m^2 n^{4.5}(\log n + \log M) \sum_{k=0}^{K-1} k N_k(\Sigma)^2)$.

Proof. We first show that the sizes of all data in Algorithm 13 are polynomial in the problem input size, which is polynomial in K, n, and $\log M$. To see this, for any integer $k \ge 0$,

$$\begin{aligned} \|x(k)\|_\infty &= \max\{\|A_i x(k-1)\|_\infty \mid A_i \in \Sigma\} \le \max\{\|A_i\|_\infty \mid A_i \in \Sigma\} \cdot \|x(k-1)\|_\infty \\ &\le (\max\{\|A_i\|_\infty \mid A_i \in \Sigma\})^k \cdot \|a\|_\infty \le (nM)^k M. \end{aligned}$$

Therefore, the size of $x(k)$ is $O(n \log \|x(k)\|_\infty) = O(kn(\log n + \log M))$.

At Step 6 of iteration k, the number of operations of solving one linear program (4.6) with $S = \cup_{i=1}^m F_k^i$ using Karmarkar's algorithm is $O(n^{3.5} L)$ [68], where the input length $L = O(\sum_{i=1}^m |F_k^i| n \log \|x(k)\|_\infty) = O(kmn(\log n + \log M)|E_k|)$. Since we need to solve $m|E_k|$ linear programs, one for each point in S, the running time of Step 6 is $m|E_k| O(n^{3.5} L) = O(km^2 n^{4.5}(\log n + \log M)|E_k|^2)$. At iteration k, Step 5 takes $O(mn^2)$ time, Step 8 takes $|E_K|$ queries to the value oracle of function f, and Step 9 can be performed in K steps if a m-ary tree is used to store the values of $x(k)$ for each k. Therefore, the step with the dominating complexity is Step 6, and the overall running time of Algorithm 13 is $O(m^2 n^{4.5}(\log n + \log M) \sum_{k=0}^{K-1} k|E_k|^2)$. Since $|E_k| \le N_k(\Sigma)$, the result follows. □

Speeding up Algorithm 13 when $n = 2$

When $n = 2$, there are many efficient algorithms to construct the convex hull of a set of points directly, such as Graham's scan and Jarvis's march [21]. Graham's scan constructs the convex

hull of l points on the plane in $O(l \log l)$ time [122]. With a similar analysis as in Proposition 11, we have the result below.

Proposition 12. *When $n = 2$ and Graham's scan is employed at Step 6 of Algorithm 13 to construct E_{k+1}, the running time of Algorithm 13 is $O(m \log m \sum_{k=0}^{K-1} N_k(\Sigma) + m \sum_{k=0}^{K-1} N_k(\Sigma) \log N_k(\Sigma))$.*

4.4 Polynomially Solvable Cases

In this section, we focus on discovering conditions on a set of matrices for which (P3) is polynomially solvable. Propositions 11 and 12 indicate that (P3) is polynomially solvable if $N_k(\Sigma)$ is polynomial in k. This is the motivation that we introduce the concept of the oligo-vertex property in Section 4.1. Recall that a set of matrices Σ has the oligo-vertex property if $N_k(\Sigma) = O(k^d)$ for some constant d. The following proposition gives the detailed time complexity of our algorithms for matrices with the oligo-vertex property, following directly from Propositions 11 and 12.

Proposition 13. *If the set of matrices Σ in (P3) has the oligo-vertex property and $N_k(\Sigma) = O(k^d)$ for some constant d, then (P3) can be solved in $O(m^2 n^{4.5} K^{2d+2}(\log n + \log M))$ time for general n and in $O(mK^{d+1}(\log m + \log K))$ time when $n = 2$.*

Thus our focus in this section is to discover conditions for a set of matrices to have the oligo-vertex property. We introduce additional notations that will be used in the rest of the paper. Given a set of matrices $\Sigma = \{A_1, A_2, \ldots, A_m\} \subseteq \mathbb{R}^n$ and a vector $a \in \mathbb{R}^n$, define

$$X_k(\Sigma, a) = \{x(k) \mid x(k) = T_{k-1} \cdots T_0 a, T_j \in \Sigma, j \in [0 : k-1]\} \tag{4.7}$$

$$E_k(\Sigma, a) = \text{ext}(P_k(\Sigma, a)) \tag{4.8}$$

for each integer $k \geq 0$. Recall that $P_k(\Sigma, a) = \text{conv}(X_k(\Sigma, a))$, $N_k(\Sigma, a) = |E_k(\Sigma, a)|$, and $N_k(\Sigma) = \sup_{a \in \mathbb{R}^n}\{N_k(\Sigma, a)\}$. Since $P_k(\Sigma, a)$ is the convex hull of at most m^k points, both $N_k(\Sigma, a)$ and $N_k(\Sigma)$ are well defined and bounded above by m^k.

Some obvious cases that have the oligo-vertex property include a set Σ of m pairwise commuting matrices with constant m (for which $N_k(\Sigma) = O(k^{m-1})$ since there are at most $\binom{k+m-1}{m-1}$ elements in $X_k(\Sigma, a)$), and a pair of projection matrices since there are at most $2k$ elements in $X_k(\Sigma, a)$.

Proposition 14. *A set Σ of m matrices in $\mathbb{R}^{n \times n}$ with at most one matrix with rank greater than one has the oligo-vertex property and $N_k(\Sigma) = O(mk)$.*

Proof. Let $\Sigma = \{A_1, \ldots, A_m\}$. With loss of generality, assume that no A_i is the zero matrix, and $A_1, A_2, \ldots, A_{m-1}$ are of rank one. Then for any $a \in \mathbb{R}^n$ the set $A_i P_k(\Sigma, a)$ contains at most two extreme points for $i = 1, \ldots, m-1$. For each integer $k \geq 0$, $P_{k+1}(\Sigma, a) = \text{conv}(\cup_{i=1}^m A_i P_k(\Sigma, a))$, so $N_{k+1}(\Sigma, a) \leq \sum_{i=1}^m |\text{ext}(A_i P_k(\Sigma, a))| \leq 2(m-1) + N_k(\Sigma, a)$. Then $N_{k+1}(\Sigma, a) \leq N_0(\Sigma, a) + 2k(m-1)$, so $N_k(\Sigma) = O(mk)$. □

Proposition 15. *A set Σ of two 2×2 matrices that share at least one common eigenvector has the oligo-vertex property and $N_k(\Sigma) = O(k)$.*

Proof. If matrices A and B in Σ share two eigenvectors, then they commute and there are at most $k+1$ different points in $X_k(\Sigma, a)$ for any a. Now suppose that A and B in Σ share exactly one eigenvector q_1. Then q_1 must be a real vector. Assume the corresponding eigenvalues of q_1 in A and B are λ_{11} and μ_{11}, respectively. Since q_1 is a real vector, λ_{11} and μ_{11} are both real-valued. Without loss of generality, assume $\|q_1\|_2 = 1$. Let $q_2 \in \mathbb{R}^2$ be a unit vector orthogonal to q_1. Consider the vector Aq_2. Since q_1 and q_2 form a basis of $\mathbb{R}^2$, we have $Aq_2 = \lambda_{12} q_1 + \lambda_{22} q_2$ for some $\lambda_{12}, \lambda_{22} \in \mathbb{R}$. Similarly, we have $Bq_2 = \mu_{12} q_1 + \mu_{22} q_2$ for some $\mu_{12}, \mu_{22} \in \mathbb{R}$. Let $Q = \begin{bmatrix} q_1 & q_2 \end{bmatrix}$, $\Lambda = \begin{bmatrix} \lambda_{11} & \lambda_{12} \\ 0 & \lambda_{22} \end{bmatrix}$, $M = \begin{bmatrix} \mu_{11} & \mu_{12} \\ 0 & \mu_{22} \end{bmatrix}$. We have Λ and M as real matrices, $QQ^\top = I$, $A = Q\Lambda Q^\top$, and $B = QMQ^\top$.

Any product of k matrices with A and B can be written in the form of $A^{l_1} B^{m_1} A^{l_2} B^{m_2} \ldots A^{l_s} B^{m_s}$ with $l_1, m_s \in \mathbb{N}$, $l_2, \ldots, l_s, m_1, \ldots, m_{s-1} > 0$ for some $s \geq 1$, and $\sum_{j=1}^s (l_j + m_j) = k$. We simplify the product as follows.

$$\begin{aligned} & A^{l_1} B^{m_1} A^{l_2} B^{m_2} \ldots A^{l_s} B^{m_s} \\ = \; & Q \begin{bmatrix} \lambda_{11}^{l_1+\ldots+l_s} \mu_{11}^{m_1+\ldots+m_s} & * \\ 0 & \lambda_{22}^{l_1+\ldots+l_s} \mu_{22}^{m_1+\ldots+m_s} \end{bmatrix} Q^\top \\ = \; & Q \begin{bmatrix} \lambda_{11}^p \mu_{11}^{k-p} & * \\ 0 & \lambda_{22}^p \mu_{22}^{k-p} \end{bmatrix} Q^\top, \end{aligned}$$

where $p = l_1 + \ldots + l_s$ and $*$ represents some real number. Let Π_p be the set of all matrices in the form of $\begin{bmatrix} \lambda_{11}^p \mu_{11}^{k-p} & * \\ 0 & \lambda_{22}^p \mu_{22}^{k-p} \end{bmatrix}$ calculated from a product of k matrices with p matrix A's and $(k-p)$ matrix B's. The set Π_0 contains one matrix in the form of $\begin{bmatrix} \mu_{11}^k & * \\ 0 & \mu_{22}^k \end{bmatrix}$. Call this matrix C_0. The set Π_k contains one matrix in the form of $\begin{bmatrix} \lambda_{11}^k & * \\ 0 & \lambda_{22}^k \end{bmatrix}$. Call this matrix C_k. For $1 \leq p \leq k-1$, any matrix in Π_p can be represented as a convex combination of two matrices in Π_p, the ones with the smallest and largest $*$ entries. Call these two matrices C_p

and D_p. Then for any $p \in [1 : k-1]$, the vector $x(k) = A^{l_1}B^{m_1}A^{l_2}B^{m_2}\dots A^{l_s}B^{m_s}a$ with $\sum_{j=1}^{s} l_j = p$ can be represented by a convex combination of $C_p a$ and $D_p a$. Hence $P_k(\Sigma, a) = \text{conv}(\{C_0a, C_1a, D_1a, C_2a, D_2a, \dots, C_ka\})$. Therefore $N_k(\Sigma, a) \leq 2k$ and $N_k(\Sigma) = O(k)$. □

Remark 2. *Each right stochastic matrix has an eigenvector* $(1,1)^\top$. *Therefore, any pair of* 2×2 *right stochastic matrices has the oligo-vertex property and the corresponding problem (P3) is polynomially solvable.*

We present a lemma showing that the oligo-vertex property is invariant under any similarity transformation.

Lemma 1. *A finite set of* $n \times n$ *matrices* Σ *has the oligo-vertex property if and only if* $S\Sigma S^{-1}$ *has the oligo-vertex property for any nonsingular real matrix* S.

Proof. It suffices to show that $N_k(\Sigma) = N_k(S\Sigma S^{-1})$. We claim that $P_k(\Sigma, a) = P_k(S\Sigma S^{-1}, Sa)$ for any $a \in \mathbb{R}^n$. To see this, note that any extreme point p of $P_k(\Sigma, a)$ can be written as $p = T_{k-1}T_{k-2}\cdots T_0 a$ with $T_j \in \Sigma$ or $j \in [0 : k-1]$. Then

$$p = T_{k-1}T_{k-2}\cdots T_0 a = S^{-1}(ST_{k-1}S^{-1})(ST_{k-2}S^{-1})\cdots(ST_0S^{-1})Sa.$$

We have $p \in S^{-1}P_k(S\Sigma S^{-1}, Sa)$. Therefore, $P_k(\Sigma, a) \subseteq S^{-1}P_k(S\Sigma S^{-1}, Sa)$. Similarly, we can show that $P_k(\Sigma, a) \supseteq S^{-1}P_k(S\Sigma S^{-1}, Sa)$, so $P_k(\Sigma, a) = S^{-1}P_k(S\Sigma S^{-1}, Sa)$. Since S is nonsingular, the number of extreme points of $P_k(\Sigma, a)$ equals the number of extreme points of $P_k(S\Sigma S^{-1}, Sa)$, i.e., $N_k(\Sigma, a) = N_k(S\Sigma S^{-1}, Sa)$. Thus $N_k(\Sigma) = \sup_{a\in\mathbb{R}^n} N_k(\Sigma, a) = \sup_{a\in\mathbb{R}^n} N_k(S\Sigma S^{-1}, Sa) \leq N_k(S\Sigma S^{-1})$. By symmetry, we can show that $N_k(S\Sigma S^{-1}) \leq N_k(\Sigma)$. Therefore, $N_k(\Sigma) = N_k(S\Sigma S^{-1})$. □

Finally, we present a proposition that the oligo-vertex property is invariant under union of a rank one matrix.

Proposition 16. *Suppose* Σ *is a set of matrices that has the oligo-vertex property and* M *is a rank one matrix, then the set* $\Sigma_0 = \Sigma \cup \{M\}$ *has the oligo-vertex property.*

Proof. For a switched linear system (Σ_0, u), $X_k(\Sigma_0, u)$ is the set of all possible values of $x(k)$ and $P_k(\Sigma_0, u)$ is the convex hull of all possible values of $x(k)$.

For any control sequence of length k that is drawn from Σ_0,

$$T_0, T_1, \cdots, T_{k-1}, \qquad T_i \in \Sigma_0,\ i = 0, 1, \dots, k-1$$

let $I(T_0, T_1, \cdots, T_{k-1})$ be the index of the last appearance of M in this sequence. We define

$$I(T_0, T_1, \cdots, T_{k-1}) = \begin{cases} \max\{i \mid T_i = M\}, & \text{if } M \text{ is in the sequence,} \\ k, & \text{otherwise.} \end{cases}$$

With the notation $I(T_0, T_1, \cdots, T_{k-1})$, we can partition the points in $X_k(\Sigma_0, u)$ based on the last appearance of M in their control sequence. Let

$$E^j = \{T_{k-1} \cdots T_1 T_0 u \mid T_i \in \Sigma_0,\ i = 0, 1, \ldots, k-1; I(T_0, T_1, \cdots, T_{k-1}) = j\},$$

we have

$$P_k(\Sigma_0, u) = \text{conv}(X_k(\Sigma_0, u)) = \text{conv}(\cup_{j=0}^{k} E^j).$$

Hence, $N_k(\Sigma_0, u) \leq \sum_{j=0}^{k} |\text{ext}(\text{conv}(E^j))|$.

- Notice that by definition,

  $$E^k = X_k(\Sigma, u).$$

  Hence, $|\text{ext}(\text{conv}(E^k))| = |\text{ext}(=)N_k(\Sigma, u)| \leq N_k(\Sigma)$.

- For $j = 0$,

  $$\begin{aligned} E^0 =& \{T_{k-1} \cdots T_1 M u \mid T_i \in \Sigma_0,\ i = 1, \ldots, k-1\} \\ =& X_{k-1}(\Sigma, Mu). \end{aligned}$$

  Hence, $|\text{ext}(\text{conv}(E^0))| \leq N_{k-1}(\Sigma)$.

- For $1 \leq j < k$,

  $$\begin{aligned} E^j =& \{T_{k-1} \cdots T_1 T_0 u \mid T_i \in \Sigma_0,\ i = 0, 1, \ldots, j-1; T_j = M; T_i \in \Sigma, i = j+1, j+2, \ldots, k-1\} \\ =& \{T_{k-1} \cdots T_{j+1} T_j x \mid x \in X_{j-1}(\Sigma_0, u); T_j = M; T_i \in \Sigma, i = j+1, j+2, \ldots, k-1\} \\ =& \{T_{k-1} \cdots T_{j+1} x \mid x \in M X_{j-1}(\Sigma_0, u); T_i \in \Sigma, i = j+1, j+2, \ldots, k-1\}. \end{aligned}$$

  Since M is a rank one matrix, the points in the set $MX_{j-1}(\Sigma_0, u)$ lay on a line. The convex hull of $MX_{j-1}(\Sigma_0, u)$ has at most two extreme points, which we denote by u_1, u_2

(If there is only one extreme point, we have $u_1 = u_2$). Hence,

$$\begin{aligned}\mathrm{conv}(E^j) =& \mathrm{conv}(\{T_{k-1}\cdots T_{j+1}x \mid x\{u_1,u_2\}; T_i \in \Sigma, i = j+1, j+2, \ldots, k-1\})\\ =& \mathrm{conv}(\mathrm{conv}(X_{k-j-1}(\Sigma,u_1)) \cup \mathrm{conv}(X_{k-j-1}(\Sigma,u_2)))\\ =& \mathrm{conv}(P_{k-j-1}(\Sigma,u_1) \cup P_{k-j-1}(\Sigma,u_2)),\end{aligned}$$

which implies

$$\begin{aligned}|\mathrm{ext}(E^j)| &\le |\mathrm{ext}(P_{k-j-1}(\Sigma,u_1))| + |\mathrm{ext}(P_{k-j-1}(\Sigma,u_2))|\\ &= N_{k-j-1}(\Sigma,u_1) + N_{k-j-1}(\Sigma,u_2)\\ &\le 2N_{k-j-1}(\Sigma).\end{aligned}$$

Since Σ has the oligo-vertex property, there exist constants $c, d, k_0 \in \mathbb{Z}_+$ such that $N_k(\Sigma) \le ck^d$ for $k \ge k_0$. When $k < k_0$, $N_k(\Sigma) \le m^k \le m^{k_0}$ where $m = |\Sigma|$. Hence, we can choose sufficiently large c, d such that $N_k(\Sigma) \le ck^d$ for $k \ge 0$. Therefore,

$$\begin{aligned}N_k(\Sigma_0,u) &\le \sum_{j=0}^{k} |\mathrm{ext}(\mathrm{conv}(E^j))|\\ &= |\mathrm{ext}(\mathrm{conv}(E^k))| + \sum_{j=0}^{k-1} |\mathrm{ext}(\mathrm{conv}(E^j))|\\ &\le N_k(\Sigma) + 2\sum_{j=0}^{k-1} N_{k-j-1}(\Sigma)\\ &\le ck^d + 2\sum_{j=0}^{k-1} cj^d\\ &\le (c + 2kc)k^d\\ &= O(k^{d+1}).\end{aligned}$$

Hence, the set Σ_0 has the oligo-vertex property.

□

4.5 The 2×2 Binary Matrices

Our main result in this section is the following theorem.

Theorem 4. *A pair of 2×2 binary matrices has the oligo-vertex property.*

The seemingly innocent looking statement above is the most difficult to prove in this paper. In fact, we are unable to provide a unified argument for all 2×2 binary matrices. This is not too surprising, however, since to the best of our knowledge there is no unified argument to show that any pair of 2×2 binary matrices has the finiteness property either [99]. We hope that the techniques we develop in this paper can be useful in proving the oligo-vertex property for other matrices in the future.

There are a total of 16 binary matrices, resulting in a total of 120 different pairs of 2×2 binary matrices. To prove Theorem 4, we first show that the result holds for most of the 120 pairs, and then provide separate proofs for each of the remaining pairs. Among the 16 binary matrices, one matrix has rank zero, nine matrices have rank one, and six matrices have rank two. The pair of matrices has the oligo-vertex property if one matrix is the zero or identity matrix. According to Proposition 14, the pair of matrices has the oligo-vertex property if one matrix is singular. Therefore, only the following five binary matrices of rank two give rise to interesting pairs:

$$A_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad A_3 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \quad A_4 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}, \quad A_5 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

The five matrices above give rise to ten different pairs of binary matrices. Observe that

$$A_1 A_1 A_1^{-1} = A_1, \ A_1 A_2 A_1^{-1} = A_3, \ A_1 A_4 A_1^{-1} = A_5, \ A_2 A_5 A_2^{-1} = A_4.$$

Then by Lemma 1, we can group the ten pairs of matrices into the following five clusters:

1. $\{A_1, A_2\}, \{A_1, A_3\}$
2. $\{A_1, A_4\}, \{A_1, A_5\}$
3. $\{A_2, A_3\}$
4. $\{A_4, A_5\}$
5. $\{A_2, A_4\}, \{A_3, A_5\}, \{A_2, A_5\}, \{A_3, A_4\},$

and it suffices to show that one pair of matrices within each cluster has the oligo-vertex property. In the rest of this section, we are going to show separately that each of the following five pairs of matrices has the oligo-vertex property.

$$\Sigma_1 = \{A_1, A_2\}, \Sigma_2 = \{A_1, A_4\}, \Sigma_3 = \{A_2, A_3\}, \Sigma_4 = \{A_4, A_5\}, \Sigma_5 = \{A_2, A_4\}.$$

We first present in the table below a complete description of how $N_k(\Sigma, a)$ grows with k for the five pairs of matrices, according to the location of the initial vector a.

	Σ_1	Σ_2	Σ_3	Σ_4	Σ_5
$a \in \mathcal{Q}_1 \cup \mathcal{Q}_3$	$O(k^2)$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
$a \in \text{int}(\mathcal{Q}_2) \cup \text{int}(\mathcal{Q}_4)$	$O(k^4)$	$O(k)$	$O(k^2)$	$O(k^2)$	$O(k^2)$

Table 4.1: The number of extreme points $N_k(\Sigma, a)$

The results in Table 4.1 show that the number of extreme points of $P_k(\Sigma, a)$ grows linearly with k when the initial vector is in the first or the third quadrant for most pairs of binary matrices except Σ_1.

Example 3. *Figure 4.3 illustrates how the number of extreme points $N_k(\Sigma_1, a)$ changes with k given different initial vector a's. For the chosen a's, the growth is at most linear in k for $k \leq 40$.*

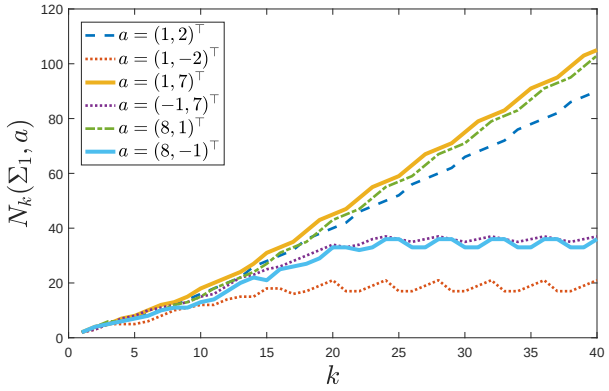


Figure 4.3: The number of extreme points $N_k(\Sigma_1, a)$ given different initial vector a's.

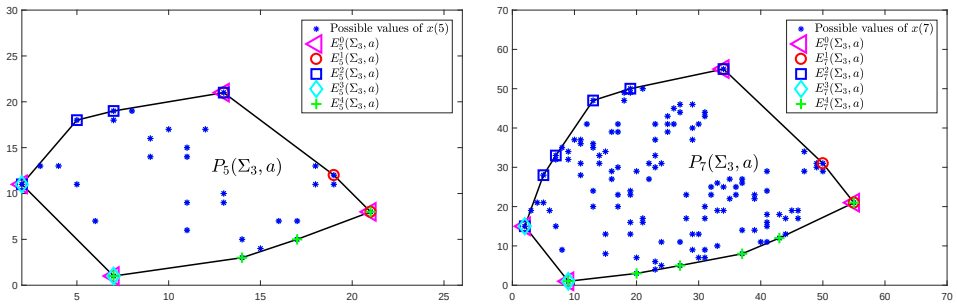
To prove the results in Table 4.1, we first introduce a few notations that will be used in the rest of this section. Given a pair Σ of matrices and a vector $a \in \mathbb{R}^2$, we divide the set of extreme points $E_k(\Sigma, a)$ of $P_k(\Sigma, a)$ into five groups.

Definition 21. *Let $E_k^i(\Sigma, a)$ be the set of extreme points of $P_k(\Sigma, a)$ that are maximizers of the linear program $\max\{cx \mid x \in P_k(\Sigma, a)\}$ for some $c \in \text{int}(\mathcal{Q}_i)$, for $i = 1, 2, 3, 4$. Let $E_k^0(\Sigma, a)$ be the set of extreme points of $P_k(\Sigma, a)$ that are maximizers of the linear programs $\max\{cx \mid x \in P_k(\Sigma, a)\}$ where $c \in \{(1, 0), (0, 1), (-1, 0), (0, -1)\}$.*

Then

$$E_k(\Sigma, a) = \cup_{i=0}^4 E_k^i(\Sigma, a) \text{ and } N_k(\Sigma, a) \leq \sum_{i=0}^4 |E_k^i(\Sigma, a)|. \tag{4.9}$$

Example 4. *Figure 4.4 illustrates the polytopes* $P_k(\Sigma_3, a)$ *and the sets of extreme points* $E_k^i(\Sigma_3, a)$ *for* $i \in [0:4]$ *with* $a = (2,1)^\top$*, for* $k = 5$ *and* $k = 7$*.*



(a) $P_5(\Sigma_3, (2,1)^\top)$ and $E_5^i(\Sigma_3, (2,1)^\top)$ (b) $P_7(\Sigma_3, (2,1)^\top)$ and $E_7^i(\Sigma_3, (2,1)^\top)$

Figure 4.4: Examples of polytopes $P_k(\Sigma_3, a)$ and associated sets of extreme points $E_k^i(\Sigma_3, a)$ for $i \in [0:4]$.

4.5.1 $\Sigma_1 = \{A_1, A_2\}$

Proposition 17. *The pair* Σ_1 *has the oligo-vertex property and* $N_k(\Sigma_1) = O(k^4)$*.*

Proposition 17 is an immediate consequence of the following propositions.

Proposition 18. *For any* $a \in \mathrm{int}(\mathcal{Q}_1) \cup \mathrm{int}(\mathcal{Q}_3)$*,* $N_k(\Sigma_1, a) = O(k^2)$*.*

Proposition 19. *For any* $a \in \partial\mathcal{Q}_1 \cup \partial\mathcal{Q}_3$*,* $N_k(\Sigma_1, a) = O(k^2)$*.*

Proposition 20. *For any* $a \in \mathrm{int}(\mathcal{Q}_2) \cup \mathrm{int}(\mathcal{Q}_4)$*,* $N_k(\Sigma_1, a) = O(k^4)$*.*

We first focus on proving Proposition 18. Our strategy is to bound the cardinality of $E_k^i(\Sigma_1, a)$ for each i. Then according to (4.9), $N_k(\Sigma_1)$ will be bounded as well.

Lemma 2. *For any* $a \in \mathrm{int}(\mathcal{Q}_1)$ *and integer* $k \geq 2$*,* $|E_k^1(\Sigma_1, a)| \leq k + 2$*.*

Proof. To simplify the notations, we write E_k^1 and P_k instead of $E_k^1(\Sigma_1, a)$ and $P_k(\Sigma_1, a)$ respectively in the rest of the proof. We claim that $|E_k^1| \leq |E_{k-1}^1| + 1$ for $k \geq 2$. Then $|E_k^1| \leq |E_1^1| + (k-1) \leq 2 + (k-1) = k + 1$. To prove the claim, we first show that $E_k^1 \subseteq A_1 E_{k-1}^1 \cup A_2 E_{k-1}^1$. Note that

$$\max\{cx \mid x \in P_k\} = \max\{\max\{cA_1 x \mid x \in P_{k-1}\}, \max\{cA_2 x \mid x \in P_{k-1}\}\}. \tag{4.10}$$

Given $c \in \text{int}(\mathcal{Q}_1)$, both cA_1 and cA_2 are in the interior of $\mathcal{Q}_1$, so the maximizers of linear programs on the right are in the set E^1_{k-1}. Therefore, $E^1_k \subseteq A_1E^1_{k-1} \cup A_2E^1_{k-1}$.

Next we show that some points in $A_1E^1_{k-1} \cup A_2E^1_{k-1}$ cannot belong to E^1_k. Let $p = (p_1, p_2)^\top \in E^1_{k-1}$ be the maximizer of the linear program $\max\{x_1 + x_2 \mid x \in P_{k-1}\}$ with the smallest x_2-coordinate. Note that there is no other point in E^1_{k-1} whose x_2-coordinate is p_2. Otherwise suppose that there is such a point p'. The fact that p is the maximizer of $\max\{x_1+x_2 \mid x \in P_{k-1}\}$ implies $p_1 > p'_1$. Then $cp' < cp$ for any $c \in \text{int}(\mathcal{Q}_1)$, which contradicts that $p' \in E^1_{k-1}$. Now we can partition E^1_{k-1} into three sets $S_1 = \{x \mid x \in E^1_{k-1}, x_2 > p_2\}$, $S_2 = \{p\}$, and $S_3 = \{x \mid x \in E^1_{k-1}, x_2 < p_2\}$. Then $E^1_k \subseteq A_1S_1 \cup A_1S_2 \cup A_1S_3 \cup A_2S_1 \cup A_2S_2 \cup A_2S_3$. We show below that the points in A_1S_1 or A_2S_3 cannot be in E^1_k.

First consider any point $x \in S_1$.

- If $x_1 < x_2$, we have $cA_2x - cA_1x = c_1x_1 + c_2(x_2 - x_1) > 0$.
- Suppose $x_1 \geq x_2$ and $c_1 < c_2$. Since $x_1 + x_2 \leq p_1 + p_2$, we have $x_1 - p_1 \leq p_2 - x_2 < 0$. Therefore, $cA_1p - cA_1x = c_1(p_2 - x_2) + c_2(p_1 - x_1) \geq c_1(x_1 - p_1) + c_2(p_1 - x_1) = (c_1 - c_2)(x_1 - p_1) > 0$.
- Suppose $x_1 \geq x_2$ and $c_1 \geq c_2$. Since $x_1 + x_2 \leq p_1 + p_2$, $p_2 - x_1 \geq x_2 - p_1$. Since $x_1 \geq x_2 > p_2$ and $x_1 + x_2 \leq p_1 + p_2$, we have $p_1 \geq x_2$. Then $cA_2p - cA_1x = c_1p_1 + c_1(p_2 - x_2) + c_2(p_2 - x_1) \geq c_1p_1 + c_1(p_2 - x_2) + c_2(x_2 - p_1) = (c_1 - c_2)(p_1 - x_2) + c_1p_2 > 0$.

Therefore, $A_1x \in A_1S_1$ cannot be a maximizer of linear program (4.10) with $c \in \text{int}(\mathcal{Q}_1)$.

Now consider any point $x \in S_3$. Since $p_2 - x_2 > 0$ and $p_1 + p_2 \geq x_1 + x_2$, $cA_2p - cA_2x = c_1(p_1 + p_2 - x_1 - x_2) + c_2(p_2 - x_2) > 0$. Therefore, $A_2x \in A_2S_3$ cannot be a maximizer of linear program (4.10) with $c \in \text{int}(\mathcal{Q}_1)$. Hence, $|E^1_k| \leq |A_1S_2| + |A_1S_3| + |A_2S_1| + |A_2S_2| = |S_1| + |S_2| + |S_3| + |S_2| = |E^1_{k-1}| + 1$. □

Lemma 3. *For any $a \in \text{int}(\mathcal{Q}_1)$ and integer $k \geq 2$, $|E^3_k(\Sigma_1, a)| \leq 2$.*

Proof. To simplify the notations, we write E^3_k instead of $E^3_k(\Sigma_1, a)$ in the rest of the proof. Let $a = (a_1, a_2)^\top \in \text{int}(\mathcal{Q}_1)$. Assume that $a_1 \leq a_2$. The case in which $a_1 > a_2$ can be proved similarly. We show below by induction that $E^3_k \subseteq \{A^k_1a, A^{k-2}_1A_2A_1a\}$ for any $k \geq 2$. For the base case $k = 2$, given any $c \in \text{int}(\mathcal{Q}_3)$, $cA^2_1a - cA^2_2a = -2c_1a_2 > 0, cA^2_1a - cA_1A_2a = c_1(a_1 - a_2) - c_2a_1 > 0$. Hence, $E^3_2 \subseteq \{A^2_1a, A_2A_1a\}$.

Now suppose that $E^3_t \subseteq \{A^t_1a, A^{t-2}_1A_2A_1a\}$ for some $t \geq 2$. We want to show that $E^3_{t+1} \subseteq \{A^{t+1}_1a, A^{t-1}_1A_2A_1a\}$. We assume that t is even (a similar argument can be used to prove the result when t is odd). Similar to the proof of (4.10) in Lemma 2, we have $E^3_k \subseteq A_1E^3_{k-1} \cup A_2E^3_{k-1}$ for $k \geq 2$. Then by the induction hypothesis, we have $E^3_{t+1} \subseteq \{A^{t+1}_1a, A^{t-1}_1A_2A_1a, A_2A^t_1a, A_2A^{t-2}_1A_2A_1a$. Since t is even, $A^t_1a = a$ and $A^{t-2}_1A_2A_1a =$

$(a_1 + a_2, a_1)^\top$. For any $c \in \text{int}(\mathcal{Q}_3)$, $cA_1^{t+1}a - cA_2A_1^t a = -c_1a_1 + c_2(a_1 - a_2) > 0$, and $cA_1^{t+1}a - cA_2A_1^{t-2}A_2A_1a = -2c_1a_1 > 0$.

Hence, $E^3_{t+1} \subseteq \{A_1^{t+1}a, A_1^{t-1}A_2A_1a\}$. We conclude that $|E^3_k| \leq 2$ for any integer $k \geq 2$. □

Lemma 4. *For any $a \in \text{int}(\mathcal{Q}_1)$ and integer $k \geq 2$, $|E^4_k(\Sigma_1, a)| \leq E^4_{k-1}(\Sigma_1, a)|+|E^1_{k-1}(\Sigma_1, a)|+2$ and $|E^2_k(\Sigma_1, a)| \leq |E^4_{k-1}(\Sigma_1, a)|$.*

Proof. To simplify the notations, we omit the dependence of Σ_1 and a in the rest of the proof. We first prove that $|E^4_k| \leq |E^4_{k-1}| + |E^1_{k-1}| + 2$. Note that

$$\begin{aligned}\max\{cx \mid x \in P_k\} = \max\{ &\max\{cA_1A_1x \mid x \in P_{k-2}\}, \max\{cA_1A_2x \mid x \in P_{k-2}\},\\ &\max\{cA_2A_1x \mid x \in P_{k-2}\}, \max\{cA_2A_2x \mid x \in P_{k-2}\}\}.\end{aligned}$$

Since $P_{k-2} \subseteq \text{int}(\mathcal{Q}_1)$, for any c with $c_1 > 0$ and $c_2 < 0$ and $x \in P_{k-2}$, $cA_2^2x = (c_1, 2c_1 + c_2)x > (c_1, c_2)x = cA_1^2x, cA_2^2x = (c_1, 2c_1 + c_2)x > (c_2, c_1 + c_2)x = cA_1A_2x$. Therefore, $\max\{cx \mid x \in P_k\} = \max\{\max\{cA_2A_1x \mid x \in P_{k-2}\}, \max\{cA_2A_2x \mid x \in P_{k-2}\}\} = \max\{cA_2x \mid x \in P_{k-1}\}$. Now that $cA_2 = (c_1, c_1 + c_2)$ is a vector in the first or the fourth quadrant, the maximizers of $\max\{cx \mid x \in P_k\}$ must be in $A_2E^1_{k-1} \cup A_2E^4_{k-1} \cup A_2S$, where S is the set of extreme points of P_{k-1} that are maximizers of $\max\{x_1 \mid x \in P_{k-1}\}$. Therefore, $|E^4_k| \leq |A_2E^4_{k-1}| + |A_2E^1_{k-1}| + |A_2S| \leq |E^4_{k-1}| + |E^1_{k-1}| + 2$.

To prove that $|E^2_k| \leq |E^4_{k-1}|$, consider $c = (c_1, c_2)$ with $c_1 < 0$ and $c_2 > 0$. For any $x \in P_{k-2}$,

$$\begin{aligned}cA_1A_2x &= (c_2, c_1 + c_2)x > (c_1 + c_2, c_1)x = c^\top A_2A_1x,\\ cA_1A_2x &= (c_2, c_1 + c_2)x > (c_1, 2c_1 + c_2)x = c^\top A_2A_2x.\end{aligned}$$

Thus we have $\max\{cx \mid x \in P_k\} = \max\{\max\{cA_1A_1x \mid x \in P_{k-2}\}, \max\{cA_1A_2x \mid x \in P_{k-2}\}\} = \max\{cA_1x \mid x \in P_{k-1}\}$. Since $cA_1 = (c_2, c_1)$ is a vector in the interior of the fourth quadrant, the optimal solutions of $\max\{cx \mid x \in P_k\}$ must be in $A_1E^4_{k-1}$. Therefore, $|E^2_k| \leq |E^4_{k-1}|$. □

Now we are ready to prove Proposition 18,

Proof of Proposition 18. We only need to prove the case where $a \in \text{int}(\mathcal{Q}_1)$. When $a \in \text{int}(\mathcal{Q}_3)$, it is easy to verify that $N_k(\Sigma_1, a) = N_k(\Sigma_1, -a)$. By Lemma 2 and Lemma 3, we have $|E^1_k(\Sigma_1, a)| \leq k + 1$ and $|E^3_k(\Sigma_1, a)| \leq 2$ for any $a \in \text{int}(\mathcal{Q}_1)$ and integer $k \geq 2$. By Lemma 4, for any $a \in \text{int}(\mathcal{Q}_1)$ and integer $k \geq 3$, $|E^4_k(\Sigma_1, a)| \leq |E^4_{k-1}(\Sigma_1, a)| + |E^1_{k-1}(\Sigma_1, a)| + 2 \leq |E^4_{k-1}(\Sigma_1, a)| + (k + 2) \leq |E^4_2(\Sigma_1, a)| + \sum_{i=2}^{k-1}(i + 3) \leq \frac{1}{2}k^2 + \frac{5}{2}k - 3$, and $|E^2_k(\Sigma_1, a)| \leq |E^4_{k-1}(\Sigma_1, a)| \leq \frac{1}{2}k^2 + \frac{3}{2}k - 5$. Therefore, $N_k(\Sigma_1, a) \leq |E^1_k(\Sigma_1, a)| + |E^2_k(\Sigma_1, a)| + |E^3_k(\Sigma_1, a)| + |E^4_k(\Sigma_1, a)| + |E^0_k(\Sigma_1, a)| = O(k^2)$. □

The conclusion $N_k(\Sigma_1, a) = O(k^2)$ can be easily extended to the case where a is on the boundary of the first or third quadrant.

Proof of Proposition 19. We only need to prove the case where $a \in \partial\mathcal{Q}_1$. The case where $a \in \partial\mathcal{Q}_3$ follows from the fact $N_k(\Sigma_1, a) = N_k(\Sigma_1, -a)$. We first prove the result when a is on the positive x_1-axis. Without loss of generality, assume that $a = (1,0)^\top$. We claim that for any integer $k \geq 3$,

$$X_k(\Sigma_1, (1,0)^\top) = X_{k-2}(\Sigma_1, (1,1)^\top) \cup \{(1,0)^\top, (0,1)^\top\}.$$

To see this, consider any value of $x(k)$ in $X_k(\Sigma_1, (1,0)^\top)$ that is different from $(1,0)^\top$ and $(0,1)^\top$. Since $A_1^t a = (0,1)^\top$ for odd integer $t \geq 1$, $A_1^t a = (1,0)^\top$ for even integer $t \geq 1$, $A_2^t a = (1,0)^\top$ for any integer $t \geq 1$, and $A_2 A_1 a = (1,1)^\top$. For $x(k)$ to take a value different from $(0,1)^\top$ and $(1,0)^\top$, $x(k)$ must be in the form of $T_{k-1} \cdots T_l x(l)$ with $T_j \in \Sigma_1$ for $j \in [l : k-1]$ and $x(l) = (1,1)^\top$ for some $l \geq 2$. But when $x(l) = (1,1)^\top$, we have $A_1^j x(l) = x(l)$ for any integer $j \geq 1$. Then $x(k) = T_{k-1} \cdots T_l A_1^{l-2} x(l)$, which is a point in $X_{k-2}(\Sigma_1, (1,1)^\top)$. Thus $X_k(\Sigma_1, (1,0)^\top) \subseteq X_{k-2}(\Sigma_1, (1,1)^\top) \cup \{(1,0)^\top, (0,1)^\top\}$. On the other hand, given a point in $X_{k-2}(\Sigma_1, (1,1)^\top)$ written in the form of $T_{k-3} \cdots T_0 (1,1)^\top$ with $T_j \in \Sigma_1$ for $j \in [0 : k-3]$, we can also write it in the form of $T_{k-3} \cdots T_0 A_2 A_1 (1,0)^\top$. Thus $X_k(\Sigma_1, (1,0)^\top) \supseteq X_{k-2}(\Sigma_1, (1,1)^\top) \cup \{(1,0)^\top, (0,1)^\top\}$. Therefore, $N_k(\Sigma_1, (1,0)^\top) \leq N_{k-2}(\Sigma_1, (1,1)^\top) + 2 = O(k^2)$. The last equality follows from Proposition 18. The case where a is on the positive x_2-axis can be proved similarly. □

We proceed to prove Proposition 20. Let $X_k^{2,4}(\Sigma_1, a)$ be the set of points in $X_k(\Sigma_1, a)$ that are in the interior of the second or fourth quadrant, i.e.,

$$X_k^{2,4}(\Sigma_1, a) = X_k(\Sigma_1, a) \cap (\text{int}(\mathcal{Q}_2) \cup \text{int}(\mathcal{Q}_4)).$$

Lemma 5. *For any $a \in \text{int}(\mathcal{Q}_4)$ and integer $k \geq 2$, $X_k^{2,4}(\Sigma_1, a)$ contains no more than $4k + 4$ points.*

Proof. Without loss of generality, assume $a = (1, a_2)^\top$ with $a_2 < 0$. Let $u_0 = \max\{1, -a_2\}$ and $v_0 = \min\{1, -a_2\}$. Define the following sequence of non-negative numbers recursively $u_j = \max\{v_{j-1}, u_{j-1} - v_{j-1}\}$ and $v_j = \min\{v_{j-1}, u_{j-1} - v_{j-1}\}$ for $j \in [1 : k]$. For each $t \in [0 : k]$, define $S_t = \{(u_t, -v_t)^\top, (-u_t, v_t)^\top, (v_t, -u_t)^\top, (-v_t, u_t)^\top\}$. Given any $s^k \in X_k^{2,4}(\Sigma_1, a)$, assume that $s^k = T_{k-1} \cdots T_0 a$ with $T_j \in \Sigma_1$ for $j \in [0 : k-1]$.

We claim that for any integer $k \geq 0$, if t out of the k matrices $T_0, \cdots, T_{k-1}$ are A_2, then $s^k \in S_t$. We prove the claim by induction on k. First consider the base case $k = 0$. If $|a_2| \geq 1$, then $u_0 = -a_2$ and $v_0 = 1$, so $s^k = a = (v_0, -u_0)^\top \in S_0$. If $|a_2| < 1$, then $u_0 = 1$ and $v_0 = -a_2$, so $s^k = a = (u_0, -v_0)^\top \in S_0$. Now suppose that the claim holds for integer $k = l \geq 0$. Specifically, $s^l = T_{l-1} \cdots T_0 a \in S_t$ if $t \in [0 : l]$ out of the l matrices $T_0, \cdots, T_{l-1}$ are A_2. We want

to prove that any point $s^{l+1} = T_{l+1} \cdots T_0 a$ in $X_{l+1}^{2,4}(\Sigma_1, a)$ also belongs to S_t, if $t \in [0 : l+1]$ out of the $l+1$ matrices $T_0, \cdots, T_{l+1}$ are A_2. If $T_{l+1} = A_1$, then t out of the l matrices $T_l, \ldots, T_0$ are A_2. Based on the induction hypothesis, the point $s = T_l \cdots T_0 a \in S_t$. Since $A_1 S_t = S_t$, $s^{l+1} = A_1 s$ must be in S_t as well. If $T_{l+1} = A_2$, then $(t-1)$ out of the l matrices $T_l, \ldots, T_0$ are A_2. Based on the induction hypothesis, the point $s = T_l \cdots T_0 a \in S_{t-1}$. The set S_t contains four points. We consider one case $s = (u_{t-1}, -v_{t-1})^\top$ here, and the result for the other cases can be proved similarly. We have $s^{l+1} = A_2 s = (u_{t-1} - v_{t-1}, -v_{t-1})^\top$. Since s^{l+1} is in the interior of second or fourth quadrant and $-v_{t-1} < 0$, we must have $u_{t-1} - v_{t-1} > 0$. If $v_{t-1} \geq u_{t-1} - v_{t-1}$, then $u_t = v_{t-1}$, $v_t = u_{t-1} - v_{t-1}$, and $s^{l+1} = (v_t, -u_t)^\top \in S_t$. If $v_{t-1} < u_{t-1} - v_{t-1}$, then $u_t = u_{t-1} - v_{t-1}$, $v_t = v_{t-1}$, and $s^{l+1} = (u_t, -v_t)^\top \in S_t$. With the claim, we conclude that $X_k^{2,4}(\Sigma_1, a)$ contains at most $4k+4$ different points. □

Proof of Proposition 20. We omit the dependence of Σ_1 in the rest of the proof to simplify the notation. Given a set $S \subseteq \mathbb{R}^2$, define $X_k(S) = \cup_{a \in S} X_k(a)$.

First note that for any x in the first (third) quadrant, $A_1 x$ and $A_2 x$ are both in the first (third) quadrant. Thus the points in $X_{i+1}^{2,4}(a)$ can only be linear transformations of points in $X_i^{2,4}(a)$ under A_1 or A_2. In addition, for any x in the second or fourth quadrant, $A_1 x$ is also in the second or fourth quadrant. Therefore, for any integer $i \geq 0$, $A_1 X_i^{2,4}(a) \cup A_2 X_i^{2,4}(a) = X_{i+1}^{2,4}(a) \cup (A_2 X_i^{2,4}(a) \cap (\mathcal{Q}_1 \cup \mathcal{Q}_3))$. Given any a in the interior of the second quadrant, we have

$$\begin{aligned} X_k(a) =& X_k(X_0^{2,4}(a)) = X_{k-1}(A_1 X_0^{2,4}(a) \cup A_2 X_0^{2,4}(a)) \\ =& X_{k-1}(X_1^{2,4}(a)) \cup X_{k-1}(A_2 X_0^{2,4}(a) \cap (\mathcal{Q}_1 \cup \mathcal{Q}_3)) \\ =& (X_{k-2}(X_2^{2,4}(a)) \cup X_{k-2}(A_2 X_1^{2,4}(a) \cap (\mathcal{Q}_1 \cup \mathcal{Q}_3))) \\ & \cup X_{k-1}(A_2 X_0^{2,4}(a) \cap (\mathcal{Q}_1 \cup \mathcal{Q}_3)) \\ & \cdots \\ =& X_l(X_{k-l}^{2,4}(a)) \cup \cup_{j=l}^{k-1} X_j(A_2 X_{k-1-j}^{2,4}(a) \cap (\mathcal{Q}_1 \cup \mathcal{Q}_3)) \end{aligned} \tag{4.11}$$

for any $l \geq 1$.

On the other hand, for any x in the first or third quadrant, we have shown that there exists some integer k_0 and $\alpha > 0$ such that $N_k(x) \leq \alpha k^2$ for any integer $k \geq k_0$. Setting $l = k_0$ in equation (4.11) we have $X_k(a) = X_{k_0}(X_{k-k_0}^{2,4}(a)) \cup \cup_{j=k_0}^{k-1} X_j(A_2 X_{k-1-j}^{2,4}(a) \cap (\mathcal{Q}_1 \cup \mathcal{Q}_3))$.

Therefore,

$$\begin{aligned}
N_k(a) &\le |X_{k_0}(X^{2,4}_{k-k_0}(a))| + \sum_{j=k_0}^{k-1} \sum_{x \in A_2 X^{2,4}_{k-1-j}(a) \cap (\mathcal{Q}_1 \cup \mathcal{Q}_3)} N_j(x) \\
&\le \sum_{x \in X^{2,4}_{k-k_0}(a)} |X_{k_0}(x)| + \sum_{j=k_0}^{k-1} |A_2 X^{2,4}_{k-1-j}(a)| \alpha j^2 \\
&\le |X^{2,4}_{k-k_0}(a)| 2^{k_0} + \sum_{j=k_0}^{k-1} |X^{2,4}_{k-1-j}(a)| \cdot \alpha j^2 \\
&\le (4k - 4k_0 + 4) 2^{k_0} + \sum_{j=k_0}^{k-1} (4k - 4j) \alpha j^2 \le \beta k^4,
\end{aligned}$$

for some constant β. The second last inequality follows from Lemma 5. Therefore $N_k(a) = O(k^4)$. □

4.5.2 $\Sigma_2 = \{A_1, A_4\}$

In this section, we will prove that $N_k(\Sigma_2) = O(k)$.

Lemma 6. *Given any polytope $P \subseteq \mathbb{R} \times \mathbb{R}_+$ or $P \subseteq \mathbb{R} \times \mathbb{R}_-$, the number of extreme points of $\mathrm{conv}(P \cup A_2 P)$ is at most two more than the number of extreme points of P.*

Proof. We first prove the case where $P \subseteq \mathbb{R} \times \mathbb{R}_+$. The result is easy to show if P is a singleton or a line segment. Now suppose P is full dimensional. Let $r = (r_1, r_2)^\top$ be the extreme point of P with the largest x_2-coordinate; if there are two such extreme points, let r be the one with a larger x_1-coordinate. Similarly, let $s = (s_1, s_2)^\top$ be the extreme point of P with the smallest x_2-coordinate; let s be the one with a larger x_1-coordinate if there are two such extreme points. Divide the extreme points of P other than r and s into two sets: (1) Set Q_1 consisting of extreme points visited if we walk clockwise along the boundary of P from s to r; (2) Set Q_2 consisting of extreme points visited if we walk clockwise along the boundary of P from r to s. Let $R = \{r, s, A_2 r, A_2 s\}$. Since $\mathrm{ext}(P) = Q_1 \cup Q_2 \cup \{r, s\}$, the possible extreme points of $\mathrm{conv}(P \cup A_2 P)$ are among Q_1, Q_2, $A_2 Q_1$, $A_2 Q_2$, and R.

We claim that any point in Q_2 can be represented as a convex combination of points in $Q_1 \cup A_2 Q_2 \cup R$. To see this, first consider any point $p = (p_1, p_2)^\top \in Q_2$. By the definition of Q_2, we have $p_2 > 0$ and there exists a point $h = (h_1, h_2)^\top$ on the line segment connecting r and s such that $h_1 < p_1$ and $h_2 = p_2$. See the illustration in Figure 4.5. We can verify that $p = \lambda A_2 p + (1 - \lambda) h$ with $\lambda = \frac{p_1 - h_1}{p_1 + p_2 - h_1} \in (0, 1)$. Thus p can be represented as a convex combination of $A_2 p$ and h. Since h can also be represented by a convex combination of r and

s, p can be represented as a convex combination of A_2p, r, and s. Therefore, we show that any point in Q_2 is a convex combination of points in $Q_1 \cup A_2Q_2 \cup R$.

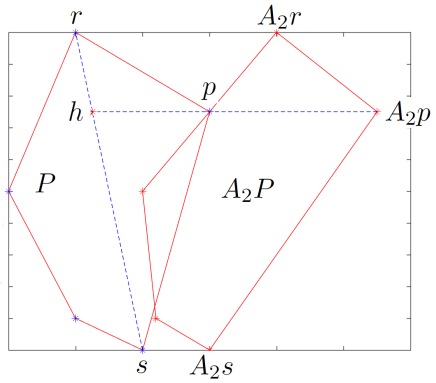


Figure 4.5: Point p is a convex combination of r, s, and A_2p.

Similarly, we can show that any point in A_2Q_1 is a convex combination of points in $Q_1 \cup A_2Q_2 \cup R$. Then we have $\text{conv}(P \cup A_2P) = \text{conv}(Q_1 \cup A_2Q_2 \cup R)$. Thus $|\text{ext}(\text{conv}(P \cup A_2P))| = |\text{ext}(\text{conv}(Q_1 \cup A_2Q_2 \cup R))| \leq |Q_1| + |A_2Q_2| + |R| \leq (|Q_1| + |Q_2| + |\{r, s\}|) + 2 \leq |\text{ext}(P)| + 2$. The result for any $P \subseteq \mathbb{R} \times \mathbb{R}_-$ can be proved similarly. □

Proposition 21. *The pair Σ_2 has the oligo-vertex property and $N_k(\Sigma_2) = O(k)$.*

Proof. To simplify the notation, we omit the dependence of Σ_2 in $N_k(\Sigma_2, a)$ and $P_k(\Sigma_2, a)$ in the rest of this proof. We claim that $N_{k+1}(a) \leq N_k(a) + 8$ for any $a \in \mathbb{R}^2$ and integer $k \geq 2$. Then $N_k(a) \leq N_{k-1}(a) + 8 \leq \cdots \leq N_2(a) + 8(k-2) \leq 4 + 8(k-2) = 8k - 12$. Thus $N_k = O(k)$.

To prove the claim, first observe that $P_{k+1}(a) = \text{conv}(A_1P_k(a) \cup A_4P_k(a)) = \text{conv}(A_1P_k(a) \cup A_2A_1P_k(a))$. Define $P^+ = A_1P_k(a) \cap \{x \mid x_2 \geq 0\}$ and $P^- = A_1P_k(a) \cap \{x \mid x_2 \leq 0\}$. Notice that P^+ is a polytope in $\mathbb{R} \times \mathbb{R}_+$ and P^- is polytope in $\mathbb{R} \times \mathbb{R}_-$, and $|\text{ext}(P^+)| + |\text{ext}(P^-)| \leq |\text{ext}(A_1P_k(a))| + 4 = N_k(a) + 4$. The first inequality above follows from the fact the line $x_2 = 0$ may introduce two new extreme points for both P^+ and P^-. On the other hand,

$$\begin{aligned} P_{k+1}(a) &= \text{conv}(A_1P_k(a) \cup A_2A_1P_k(a)) \\ &= \text{conv}(P^+ \cup P^- \cup A_2(P^+ \cup P^-)) \\ &= \text{conv}(\text{conv}(P^+ \cup A_2P^+) \cup \text{conv}(P^- \cup A_2P^-)). \end{aligned}$$

Thus $N_{k+1}(a) \leq |\text{ext}(\text{conv}(\text{conv}(P^+ \cup A_2P^+)))| + |\text{ext}(\text{conv}(P^- \cup A_2P^-))|$. By Lemma 6, $|\text{ext}(\text{conv}(P^+ \cup A_2P^+))| \leq |\text{ext}(P^+)| + 2$ and $|\text{ext}(\text{conv}(P^- \cup A_2P^-))| \leq |\text{ext}(P^-)| + 2$. Thus we have $N_{k+1}(a) \leq |\text{ext}(\text{conv}(P^+))| + 2 + |\text{ext}(\text{conv}(P^-))| + 2 \leq N_k(a) + 8$. □

4.5.3 $\Sigma_3 = \{A_2, A_3\}$

We first prove the following result when the initial vector a is in the first quadrant.

Proposition 22. *For any $a \in \mathcal{Q}_1$, $N_k(\Sigma_3, a) = O(k)$.*

Proof. The proof is similar to the proof of Proposition 18 for Σ_1. We first bound the cardinality of $E_k^i(\Sigma_3, a)$ for each i. Similar to the proofs of Lemmas 2, 3, and 4, we can show that for any $a \in \mathcal{Q}_1$, $|E_k^1(\Sigma_3, a)| \leq 4$ when $k \geq 3$, $E_k^3(\Sigma_3, a) \subseteq \{A_2^k a, A_3^k a\}$ when $k \geq 1$, $|E_k^4(\Sigma_3, a)| \leq |E_{k-1}^4(\Sigma_3, a)| + |E_{k-1}^1(\Sigma_3, a)| + 2$ and $|E_k^2(\Sigma_3, a)| \leq |E_{k-1}^2(\Sigma_3, a)| + |E_{k-1}^1(\Sigma_3, a)| + 2$ when $k \geq 1$, respectively. Then $|E_k^4(\Sigma_3, a)| \leq |E_{k-1}^4(\Sigma_3, a)| + 6 \leq |E_3^4(\Sigma_3, a)| + 6(k-3) \leq 6k - 10$. Similarly, $|E_k^2(\Sigma_3, a)| \leq 6k - 10$. Finally, for any $a \in \mathcal{Q}_1$ and integer $k \geq 3$, $N_k(\Sigma_3, a) \leq \sum_{i=0}^4 |E_k^i(\Sigma_3, a)| \leq 8 + 4 + (6k - 10) + 2 + (6k - 10) + 8 = 12k - 6$. □

Proposition 23. *The pair Σ_3 has the oligo-vertex property and $N_k(\Sigma_3) = O(k^2)$.*

Proof. We only need to prove that $N_k(\Sigma_3, a) = O(k^2)$ for any $a \in \text{int}(\mathcal{Q}_4)$. Define $f_k = \sup\{N_k(\Sigma_3, a) \mid a \in \text{int}(\mathcal{Q}_4)\}$ for any integer $k \geq 1$. Note that $f_k = \sup\{N_k(\Sigma_3, a) \mid a \in \text{int}(\mathcal{Q}_2)\}$ for $k \geq 1$ as well. Since $P_k(\Sigma_3, a) = \text{conv}(P_{k-1}(\Sigma_3, A_2 a) \cup P_{k-1}(\Sigma_3, A_3 a))$, we have $N_k(\Sigma_3, a) \leq N_{k-1}(\Sigma_3, A_2 a) + N_{k-1}(\Sigma_3, A_3 a)$. Consider a vector $a = (a_1, a_2)^\top \in \text{int}(\mathcal{Q}_4)$ with $a_1 > 0$ and $a_2 < 0$.

1. If $a_1 = -a_2$, we have $A_2 a = (0, a_2)^\top \in \mathcal{Q}_3$ and $A_3 a = (a_1, 0)^\top \in \mathcal{Q}_1$. Then there exists $\alpha > 0$ and integer k_0 such that for any integer $l \geq k_0$, $N_l(\Sigma_3, A_2 a) \leq \alpha l$ and $N_l(\Sigma_3, A_3 a) \leq \alpha l$. Thus for any integer $k \geq k_0 + 1$, $N_k(\Sigma_3, a) \leq N_{k-1}(\Sigma_3, A_2 a) + N_{k-1}(\Sigma_3, A_3 a) \leq \alpha(k-1) + \alpha(k-1) \leq 2\alpha k$. Therefore, $N_k(\Sigma_3, a) = O(k)$.

2. If $a_1 < -a_2$, we have $A_2 a = (a_1 + a_2, a_2)^\top \in \mathcal{Q}_3$ and $A_3 a = (a_1, a_1 + a_2)^\top \in \text{int}(\mathcal{Q}_4)$. Then there exists $\alpha > 0$ and integer k_0 such that for any integer $l \geq k_0$, $N_l(\Sigma_3, A_2 a) \leq \alpha l$. For any $k \geq k_0 + 1$, $N_k(\Sigma_3, a) \leq N_{k-1}(\Sigma_3, A_2 a) + N_{k-1}(\Sigma_3, A_3 a) \leq \alpha(k-1) + f_{k-1}$. Then for any $k \geq k_0 + 1$, $f_k \leq \alpha(k-1) + f_{k-1}$. Thus for any $k \geq 2k_0$,

$$\begin{aligned} f_k &\leq \alpha(k-1) + f_{k-1} \leq \alpha(k-1) + \alpha(k-2) + f_{k-2} \\ &\cdots \leq \alpha(k-1) + \alpha(k-2) + \cdots + \alpha k_0 + f_{k_0} \\ &\leq \alpha \frac{(k-1+k_0)(k-k_0)}{2} + 2^{k_0} \leq \beta k^2, \end{aligned}$$

   for some $\beta > 0$. Therefore, $f_k = O(k^2)$.

3. If $a_1 > -a_2$, it can be proved that $f_k = O(k^2)$ with a similar argument as in the case $a_1 < -a_2$.

□

4.5.4 $\Sigma_4 = \{A_4, A_5\}$

Proposition 24. *The pair Σ_4 has the oligo-vertex property and $N_k(\Sigma_4) = O(k^2)$.*

Proof. First observe that $A_4A_5 = A_2A_2$, $A_4A_4 = A_2A_3$, $A_5A_5 = A_3A_2$, and $A_5A_4 = A_3A_3$. When k is an even integer, every product of k matrices with A_2 and A_3 can be represented by a product of k matrices with A_4 and A_5 and vice versa. Therefore, for any given $a \in \mathbb{R}^2$, $P_k(\Sigma_4, a) = P_k(\Sigma_3, a)$ and $N_k(\Sigma_4, a) = N_k(\Sigma_3, a)$. When k is an odd integer, $P_k(\Sigma_4, a) = \text{conv}(A_4P_{k-1}(\Sigma_4, a) \cup A_5P_{k-1}(\Sigma_4, a))$ and $N_k(\Sigma_4, a) \leq 2N_{k-1}(\Sigma_4, a) = 2N_{k-1}(\Sigma_3, a)$. Since there exists $\alpha > 0$ and integer k_0 such that $N_k(\Sigma_3, a) \leq \alpha k^2$ for any integer $k \geq k_0$, we have $N_k(\Sigma_4, a) \leq 2\alpha k^2$ for any integer $k \geq k_0$. Therefore, $N_k(\Sigma_4) = O(k^2)$. □

4.5.5 $\Sigma_5 = \{A_2, A_4\}$

Proposition 25. *For any $a \in \mathcal{Q}_1$ with $a_1 \geq a_2$, $N_k(\Sigma_5, a) = O(k)$.*

Proof. First similar to the proofs of Lemmas 2, 3, and 4, we can show by induction that for any $a \in \mathcal{Q}_1$ with $a_1 \geq a_2$, $E^1_k(\Sigma_5, k) = \{A^k_4 a\}$ and $E^3_k(\Sigma_5, a) = \{A^k_2 a\}$ when $k \geq 0$, and $|E^4_k(\Sigma_5, a)| \leq |E^4_{k-1}(\Sigma_5, a)| + |E^1_{k-1}(\Sigma_5, a)| + 2$ and $|E^2_k(\Sigma_5, a)| \leq |E^2_{k-1}(\Sigma_5, a)| + |E^1_{k-1}(\Sigma_5, a)| + 2$ when $k \geq 1$, respectively. Then $|E^4_k(\Sigma_5, a)| \leq |E^4_{k-1}(\Sigma_5, a)| + 3 \leq |E^4_1(\Sigma_5, a)| + 3(k-1) \leq 3k - 1$. Similarly, $|E^2_k(\Sigma_5, a)| \leq 3k - 1$. Finally, for any integer $k \geq 1$, $N_k(\Sigma_5, a) \leq \sum_{i=0}^{4} |E^i_k(\Sigma_5, a)| \leq 6k + 8$. □

We now extend Proposition 25 to the case where a is in the first quadrant.

Proposition 26. *For any $a \in \mathcal{Q}_1$, $N_k(\Sigma_5, a) = O(k)$.*

Proof. For any $a = (a_1, a_2)^\top$ with $a_1 \geq 0$ and $a_2 \geq 0$, both A_2a and A_4a are contained in $\{x \in \mathbb{R}^2_+ \mid x_1 \geq x_2\}$. By Proposition 25, there exists $\alpha > 0$ and integer k_0 such that for any integer $l \geq k_0$, $N_l(\Sigma_5, A_2a) \leq \alpha l$ and $N_l(\Sigma_5, A_4a) \leq \alpha l$. Thus for any integer $k \geq k_0 + 1$, $N_k(\Sigma_5, a) \leq N_{k-1}(\Sigma_5, A_2a) + N_{k-1}(\Sigma_5, A_4a) \leq \alpha(k-1) + \alpha(k-1) \leq 2\alpha k$. Therefore, $N_k(\Sigma_5, a) = O(k)$. □

Finally, we extend the result to $a \in \mathbb{R}^2$, similar to Proposition 23 for the case Σ_3.

Proposition 27. *The pair Σ_5 has the oligo-vertex property and $N_k(\Sigma_5) = O(k^2)$.*

4.6 Computational results

In this section, we compare the performance of our algorithm with one state-of-the-art global optimization solver Baron [123]. We randomly generate 10 instances for each of the 10 sets of parameters (n, m, K) for (P3), with 100 instances in total. The parameters are summarized in

Table 4.2. The entries of each matrix are randomly drawn from a uniform distribution over $[-1, 1]$, and the entries of the initial vector a are randomly drawn from a uniform distribution over $[0, 1]$. Note that our algorithm does not rely on any additional property of f other than convexity. In order for Baron to gain a better performance, we choose a simple smooth objective function $f(x) = \|x\|_2^2$. All test instances can be downloaded at `https://github.com/qqqhe`. The mixed-integer nonlinear programming (MINLP) formulation of (P3) is given in (4.12) and solved by Baron, where A_{lij} denotes the (i, j)-th entry of the l-th matrix for $l \in [m]$. Note that we also tried to linearize the constraints in the MINLP formulation by introducing big-M constants, but we observed that Baron easily run into numerical issues with many big-M constants in the constraints, even for a small-sized instance.

$$\begin{aligned} \max_{x,z} \quad & \sum_{i=1}^{n} x_i^2(K) \\ \text{s.t.} \quad & x_i(k) = \sum_{l=1}^{m} \sum_{j=1}^{n} A_{lij} x_j(k-1) z_{k,l}, i \in [n], k \in [K], \\ & \sum_{l=1}^{m} z_{k,l} = 1, k \in [K], \\ & z_{k,l} \in \{0, 1\}, l \in [m], k \in [K], \\ & x(0) = a. \end{aligned} \tag{4.12}$$

Our algorithm is coded in Matlab. Computational experiments are conducted on a Laptop with Intel i7-6560U 2.20 GHz and 8 GB of RAM memory, under Windows 10 Operating System. The MINLP formulation is coded in AMPL and solved by Baron 18.5.8. The time limit for each instance is set to 600s. When $n \leq 5$, our algorithm employs Matlab's build-in function *convhulln* to construct the set of extreme points directly. When $n \geq 6$, our algorithm solves a linear program with the commercial solver Gurobi [16] to identify each extreme point. The computational results are summarized in Table 4.2. All test instances are solved to optimality by our algorithm within the time limit. The average solution time of our algorithm is reported in the rows "Our algorithm (s)". On the other hand, Baron cannot solve most instances to optimality, and has a variety of output for instances of different sizes. Instead of reporting the solution time, we report the number of instances with different outputs by Baron in three categories that were described in [124]: The symbol G (G!) denotes that Baron finds a global optimal solution and proves (cannot prove) its optimality within the time limit; The symbol Limit denotes that Baron finds some feasible solution within the time limit; The symbol Wrong denotes that Baron reports infeasibility or failure.

Our proposed algorithm has a clear advantage over Baron in solving (P3). Our algorithm is

(n, m, K)		(2,2,20)	(2,2,50)	(2,2,500)	(2,5,500)	(2,10,500)
Our algorithm (s)		0.013	0.031	0.300	0.298	0.289
Baron	G/G!	4/6	2/5	4/2	0/0	0/0
	Limit	0	2	1	7	7
	Wrong	0	1	3	3	3
(n, m, K)		(5,2,100)	(5,5,100)	(5,10,100)	(8,2,50)	(10,2,20)
Our algorithm (s)		1.094	2.456	2.405	59.457	58.357
Baron	G/G!	0/0	0/0	0/0	0/0	0/0
	Limit	0	0	1	0	10
	Wrong	10	10	9	10	0

Table 4.2: The average running time of our algorithm and solution statistics of Baron

very efficient in solving instances with $n = 2$ and large m and K, requiring less than one second. When n increases to 8 and 10, our algorithm is able to solve instances with $K = 50$ and $K = 20$ respectively in less than one minute. On the other hand, Baron is only able to solve several instances with a pair of 2×2 matrices to optimality. When n or m is larger than 2, it either cannot find the optimal solution within the time limit or runs into numerical issues. Finally, we observe that when the problem dimension $n \geq 8$, our algorithm is not able to solve instances with $K = 100$ within the time limit, since the running time grows rapidly with K. We suspect the reason to be that the set of randomly generated matrices no longer has the oligo-vertex property for larger n. This observation is also consistent with the fact that (P3) is NP-hard for general n.

4.7 Open Problems and Conclusions

The problem (P3) has many applications in operations research and control, and can also be seen as an approximation to the dynamics of more general continuous-time nonlinear switched systems. In this chapter, we preset an efficient exact algorithm to solve large-sized instances of (P3) that cannot be handled by state-of-the-art optimization software. We introduce an interesting property—the oligo-vertex property—for a set of matrices. We now present several open questions on the oligo-vertex property, which we believe may be of independent interest.

1. Does any set of 2×2 rational matrices have the oligo-vertex property?
2. Does any set of 2×2 real matrices have the oligo-vertex property?
3. Is there an "easy-to-check" necessary condition for a set of matrices to have the oligo-vertex property? Is there a finite-time algorithm to test the oligo-vertex property for a

given set of matrices with rational entries? If so, is deciding whether a set of matrices has the oligo-vertex property in P or NP?

4. Does the finiteness property imply the oligo-vertex property, and vice versa?
5. Is $N_k(\Sigma) = O(k)$ for any pair of 2×2 binary matrices?

The last question comes from our observation that $N_k(\Sigma, a)$ grows linearly with k for any 2×2 binary matrices in the computational experiment. We believe answers to any of the above questions will lead to a faster exact algorithm for (P3).

References

[1] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[2] Eugene L Lawler, Jan Karel Lenstra, and Alexander HG Rinnooy Kan. The traveling salesman problem. 1985.

[3] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66, 1997.

[4] Gregory Gutin and Abraham P Punnen. *The traveling salesman problem and its variations*, volume 12. Springer Science & Business Media, 2006.

[5] William J Cook. *In pursuit of the traveling salesman: mathematics at the limits of computation*. Princeton University Press, 2011.

[6] John Gunnar Carlsson and Siyuan Song. Coordinated logistics with a truck and a drone. *Management Science*, 2017.

[7] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

[8] George B Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 1(1):486–490, 1896.

[9] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Multidimensional knapsack problems. In *Knapsack problems*, pages 235–283. Springer, 2004.

[10] Yvan Dumas, François Soumis, and Jacques Desrosiers. Optimizing the schedule for a fixed vehicle path with convex inconvenience costs. *Transportation Science*, 24(2):145–152, 1990.

[11] Kjetil Fagerholt, Gilbert Laporte, and Inge Norstad. Reducing fuel emissions by optimizing speed on shipping routes. *Journal of the Operational Research Society*, 61(3):523–529, 2010.

[12] Qie He, Xiaochen Zhang, and Kameng Nip. Speed optimization over a path with heterogeneous arc costs. *Transportation Research Part B: Methodological*, 104:198–214, 2017.

[13] M Selim Aktürk, Alper Atamtürk, and Sinan Gürel. Aircraft rescheduling with cruise speed control. *Operations Research*, 62(4):829–845, 2014.

[14] Angela Nuic. User manual for the base of aircraft data (bada) revision 3.10. *Atmosphere*, 2010:001, 2010.

[15] Toshihide Ibaraki and Naoki Katoh. *Resource allocation problems: algorithmic approaches*. MIT press, 1988.

[16] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2019.

[17] IBM Data Science. Ibm ilog cplex optimization studio, 2019.

[18] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. *Integer programming*, volume 271. Springer, 2014.

[19] Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 2014.

[20] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[21] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[22] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[23] Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*, page 2430, 1965.

[24] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

[25] David L Applegate, William J Cook, Sanjeeb Dash, and David S Johnson. A practical guide to discrete optimization, 2014.

[26] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[27] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[28] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[29] Lenore Blum, Mike Shub, and Steve Smale. On a theory of computation over the real numbers; np completeness, recursive functions and universal machines. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 387–397. IEEE, 1988.

[30] Richard Ernest Bellman. Dynamic programming treatment of the traveling salesman problem. 1961.

[31] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[32] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100, 1991.

[33] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[34] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[35] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[36] Emile Aarts and Jan Korst. Simulated annealing and boltzmann machines. 1988.

[37] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[38] Zbigniew Michalewicz. *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media, 2013.

[39] César Rego, Dorabela Gamboa, Fred Glover, and Colin Osterman. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal of Operational Research*, 211(3):427–441, 2011.

[40] Michael Patriksson. A survey on the continuous nonlinear resource allocation problem. *European Journal of Operational Research*, 185(1):1–46, 2008.

[41] Michael Patriksson and Christoffer Strömberg. Algorithms for the continuous nonlinear resource allocation problemnew implementations and numerical studies. *European Journal of Operational Research*, 243(3):703–722, 2015.

[42] Thijs van der Klauw, Marco ET Gerards, and Johann L Hurink. Resource allocation problems in decentralized energy management. *OR Spectrum*, 39(3):749–773, 2017.

[43] Dorit S Hochbaum. Lower and upper bounds for the allocation problem and other nonlinear optimization problems. *Mathematics of Operations Research*, 19(2):390–409, 1994.

[44] Thibaut Vidal, Daniel Gribel, and Patrick Jaillet. Separable convex optimization with nested lower and upper constraints. *INFORMS Journal on Optimization*, 2018.

[45] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.

[46] Zhendong Sun. *Switched linear systems: control and design*. Springer Science & Business Media, 2006.

[47] Hai Lin and Panos J Antsaklis. Stability and stabilizability of switched linear systems: a survey of recent results. *IEEE Transactions on Automatic control*, 54(2):308–322, 2009.

[48] Daniel Liberzon. *Switching in systems and control*. Springer Science & Business Media, 2012.

[49] Qie He, Junfeng Zhu, David Dingli, Jasmine Foo, and Kevin Zox Leder. Optimized treatment schedules for chronic myeloid leukemia. *PLoS computational biology*, 12(10):e1005129, 2016.

[50] Martin Grötschel, László Lovász, and Alexander Schrijver. Complexity, oracles, and numerical computation. In *Geometric Algorithms and Combinatorial Optimization*, pages 21–45. Springer, 1988.

[51] Alexander Schrijver. Preliminaries on algorithms and complexity. *Combinatorial Optimization: Polyhedra and Efficiency*, 1, 2003.

[52] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[53] Hans Rock. Scaling techniques for minimal cost network flows. *Discrete structures and algorithms*, 1980.

[54] Robert G Bland and David L Jensen. On the computational behavior of a polynomial-time network flow algorithm. *Mathematical Programming*, 54(1-3):1–39, 1992.

[55] Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[56] Ravindra K Ahuja, Andrew V Goldberg, James B Orlin, and Robert E Tarjan. Finding minimum-cost flows by double scaling. *Mathematical programming*, 53(1-3):243–266, 1992.

[57] Dimitri P Bertsekas, Patrick A Hosein, and Paul Tseng. Relaxation methods for network flow problems with convex arc costs. *SIAM Journal on Control and Optimization*, 25(5):1219–1243, 1987.

[58] Eva Tardos. A strongly polynomial algorithm to solve combinatorial linear programs. *Operations Research*, 34(2):250–256, 1986.

[59] James B Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations research*, 41(2):338–350, 1993.

[60] László A Végh. A strongly polynomial algorithm for a class of minimum-cost flow problems with separable convex objectives. *SIAM Journal on Computing*, 45(5):1729–1761, 2016.

[61] PV Kamesam and Robert R Meyer. Multipoint methods for separable nonlinear networks. In *Mathematical Programming at Oberwolfach II*, pages 185–205. Springer, 1984.

[62] Dimitri P Bertsekas, Lazaros C Polymenakos, and Paul Tseng. An ϵ-relaxation method for separable convex cost network flow problems. *SIAM Journal on Optimization*, 7(3):853–870, 1997.

[63] Michel Minoux. A polynomial algorithm for minimum quadratic cost flow problems. *European Journal of Operational Research*, 18(3):377–387, 1984.

[64] Michel Minoux. Solving integer minimum cost flows with separable convex cost objective polynomially. In *Netflow at Pisa*, pages 237–239. Springer, 1986.

[65] Andrew V Goldberg and Robert E Tarjan. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[66] Alexander V Karzanov and S Thomas McCormick. Polynomial methods for separable convex optimization in unimodular linear spaces with applications. *SIAM Journal on Computing*, 26(4):1245–1275, 1997.

[67] Dorit S Hochbaum and J George Shanthikumar. Convex separable optimization is not much harder than linear optimization. *Journal of the ACM (JACM)*, 37(4):843–862, 1990.

[68] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM, 1984.

[69] Arun Padakandla and Rajesh Sundaresan. Separable convex optimization problems with linear ascending constraints. *SIAM Journal on Optimization*, 20(3):1185–1204, 2009.

[70] Zizhuo Wang. On solving convex optimization problems with linear ascending constraints. *Optimization Letters*, 9(5):819–838, 2015.

[71] Thibaut Vidal, Patrick Jaillet, and Nelson Maculan. A decomposition algorithm for nested resource allocation problems. *SIAM Journal on Optimization*, 26(2):1322–1340, 2016.

[72] O. Gross. *A Class of Discrete Type Minimization Problems*. RM-1644, Rand Corporation, 1956.

[73] Bennett Fox. Discrete optimization via marginal analysis. *Management Science*, 13(3):210–216, 1966.

[74] Peter Brucker. An O(n) algorithm for quadratic knapsack problems. *Operations Research Letters*, 3(3):163–166, 1984.

[75] P. T. Akhil and Rajesh Sundaresan. A survey of algorithms for separable convex optimization with linear ascending constraints. arXiv preprint arXiv:1608.08000, 2016.

[76] Thibaut Vidal. Private communication, 2018.

[77] Ralph Tyrell Rockafellar. *Convex analysis*. Princeton university press, 1970.

[78] David Ronen. The effect of oil price on the optimal speed of ships. *Journal of the Operational Research Society*, 33(11):1035–1040, 1982.

[79] Wei Chu and S Sathiya Keerthi. Support vector ordinal regression. *Neural computation*, 19(3):792–815, 2007.

[80] Luis Torgo. Liacc regresion dataset, 2019.

[81] Ravindra K Ahuja and Dorit S Hochbaum. Solving linear cost dynamic lot-sizing problems in o (n log n) time. *Operations research*, 56(1):255–261, 2008.

[82] Harilaos N Psaraftis and Christos A Kontovas. Ship speed optimization: Concepts, models and combined speed-routing scenarios. *Transportation Research Part C: Emerging Technologies*, 44:52–69, 2014.

[83] Balachandran Vaidyanathan and Ravindra K Ahuja. Fast algorithms for specially structured minimum cost flow problems with applications. *Operations research*, 58(6):1681–1696, 2010.

[84] James B Orlin and Balachandran Vaidyanathan. Fast algorithms for convex cost flow problems on circles, lines, and trees. *Networks*, 62(4):288–296, 2013.

[85] William S Jewell. Optimal flow through networks. In *Operation Research*, volume 6, pages 633–633, 1958.

[86] Masao Iri. A new method of solving transportation-network problems. *Journal of the Operations Research Society of Japan*, 3(1):2, 1960.

[87] Robert G. Busaker and Paul J. Gowen. A procedure for determining minimal-cost flow network patterns. Technical report, Tech. Rep. ORO-15, Operational Research Office, Johns Hopkins University, Baltimore, MD, 1961.

[88] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 1997.

[89] Portia M Mira, Kristina Crona, Devin Greene, Juan C Meza, Bernd Sturmfels, and Miriam Barlow. Rational design of antibiotic treatment plans: a treatment strategy for managing evolution and reversing resistance. *PloS One*, 10(5):e0122283, 2015.

[90] Daniel Nichol, Peter Jeavons, Alexander G Fletcher, Robert A Bonomo, Philip K Maini, Jerome L Paul, Robert A Gatenby, Alexander RA Anderson, and Jacob G Scott. Steering evolution with sequential therapy to prevent the emergence of bacterial antibiotic resistance. *PLoS computational biology*, 11(9):e1004493, 2015.

[91] Vincent D Blondel and John N Tsitsiklis. When is a pair of matrices mortal? *Information Processing Letters*, 63(5):283–286, 1997.

[92] Olivier Bournez and Michael Branicky. The mortality problem for matrices of low dimensions. *Theory of Computing Systems*, 35(4):433–448, 2002.

[93] Raphaël Jungers. *The joint spectral radius: theory and applications*, volume 385. Springer Science & Business Media, 2009.

[94] Gian-Carlo Rota and W. Gilbert Strang. A note on the joint spectral radius. *Proceedings of the Netherlands Academy*, 22:379–381, 1960.

[95] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[96] Jeffrey C Lagarias and Yang Wang. The finiteness conjecture for the generalized spectral radius of a set of matrices. *Linear Algebra and its Applications*, 214:17–42, 1995.

[97] Raphaël Jungers. On the finiteness property for rational matrices. In *The Joint Spectral Radius*, pages 63–74. Springer, 2009.

[98] Jun Liu and Mingqing Xiao. Rank-one characterization of joint spectral radius of finite matrix family. *Linear Algebra and its Applications*, 438(8):3258–3277, 2013.

[99] Raphaël M Jungers and Vincent D Blondel. On the finiteness property for rational matrices. *Linear Algebra and its Applications*, 428(10):2283–2295, 2008.

[100] Chengzhi Yuan and Fen Wu. Hybrid control for switched linear systems with average dwell time. *IEEE Transactions on Automatic Control*, 60(1):240–245, 2015.

[101] Magnus Egerstedt, Yorai Wardi, and Florent Delmotte. Optimal control of switching times in switched dynamical systems. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, volume 3, pages 2138–2143. IEEE, 2003.

[102] Duarte Antunes and WP Maurice Heemels. Linear quadratic regulation of switched systems using informed policies. *IEEE Transactions on Automatic Control*, 62(6):2675–2688, 2017.

[103] Jianghai Hu, Jinglai Shen, and Wei Zhang. Generating functions of switched linear systems: analysis, computation, and stability applications. *IEEE Transactions on Automatic Control*, 56(5):1059–1074, 2011.

[104] Wei Zhang, Jianghai Hu, and Alessandro Abate. On the value functions of the discrete-time switched lqr problem. *IEEE Transactions on Automatic Control*, 54(11):2669–2674, 2009.

[105] Zhendong Sun and Shuzhi Sam Ge. Analysis and synthesis of switched linear control systems. *Automatica*, 41(2):181–195, 2005.

[106] Feng Zhu and Panos J Antsaklis. Optimal control of hybrid switched systems: A brief survey. *Discrete Event Dynamic Systems*, 25(3):345–364, 2015.

[107] Sebastian Sager. *Numerical methods for mixed-integer optimal control problems.* PhD thesis, University of Heidelberg, 2005.

[108] Sebastian Sager, Hans Georg Bock, and Moritz Diehl. The integer approximation error in mixed-integer optimal control. *Mathematical programming*, 133(1-2):1–23, 2012.

[109] Panos J Antsaklis. A brief introduction to the theory and applications of hybrid systems. In *Proceedings of the IEEE, Special Issue on Hybrid Systems: Theory and Applications*, pages 879–887, 2000.

[110] Ngoc Mai Tran and Jed Yang. Antibiotics time machine is NP-hard. *Notices of the American Mathematical Society*, 64:1136–1140, 2017.

[111] Christos H Papadimitriou and John N Tsitsiklis. The complexity of markov decision processes. *Mathematics of operations research*, 12(3):441–450, 1987.

[112] John N Tsitsiklis and Vincent D Blondel. The lyapunov exponent and joint spectral radius of pairs of matrices are hard–when not impossible–to compute and to approximate. *Mathematics of Control, Signals and Systems*, 10(1):31–40, 1997.

[113] Vincent D Blondel and Yurii Nesterov. Computationally efficient approximations of the joint spectral radius. *SIAM Journal on Matrix Analysis and Applications*, 27(1):256–272, 2005.

[114] Pablo A Parrilo and Ali Jadbabaie. Approximation of the joint spectral radius using sum of squares. *Linear Algebra and its Applications*, 428(10):2385–2402, 2008.

[115] Amir Ali Ahmadi, Raphaël M Jungers, Pablo A Parrilo, and Mardavij Roozbehani. Joint spectral radius and path-complete graph lyapunov functions. *SIAM Journal on Control and Optimization*, 52(1):687–717, 2014.

[116] Thierry Bousch and Jean Mairesse. Asymptotic height optimization for topical ifs, tetris heaps, and the finiteness conjecture. *Journal of the American Mathematical Society*, 15(1):77–111, 2002.

[117] Vincent D Blondel, Jacques Theys, and Alexander A Vladimirov. An elementary counterexample to the finiteness conjecture. *SIAM Journal on Matrix Analysis and Applications*, 24(4):963–970, 2003.

[118] Victor Kozyakin. A dynamical systems construction of a counterexample to the finiteness conjecture. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 2338–2343. IEEE, 2005.

[119] Kevin G Hare, Ian D Morris, Nikita Sidorov, and Jacques Theys. An explicit counterexample to the lagarias–wang finiteness conjecture. *Advances in Mathematics*, 226(6):4667–4701, 2011.

[120] Michael R Garey and David S Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman, 1979.

[121] Ralph Tyrell Rockafellar. *Convex analysis*. Princeton University Press, 2015.

[122] Ronald L Graham. An efficient algorith for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[123] Mustafa R Kılınç and Nikolaos V Sahinidis. Exploiting integrality in the global optimization of mixed-integer nonlinear programming problems with baron. *Optimization Methods and Software*, 33(3):540–562, 2018.

[124] Arnold Neumaier, Oleg Shcherbina, Waltraud Huyer, and Tamás Vinkó. A comparison of complete global optimization solvers. *Mathematical programming*, 103(2):335–356, 2005.