Implementation of a Fast Approximation Algorithm for Precedence Constrained Scheduling

Department of Mathematics, Linköping University

Måns Alskog

LiTH-MAT-EX–2022/07–SE

Credits: **30 hp**

Level: **A**

Supervisors: **Hannes Uppman**,
Saab AB, Linköping

Christiane Schmidt,
Department of Science and Technology, Linköping University

Examiner: **Elina Rönnberg**,
Department of Mathematics, Linköping University

Linköping: **November 2022**

Abstract

We present an implementation of a very recent approximation algorithm for scheduling jobs on a single machine with precedence constraints, minimising the total weighted completion time. We also evaluate the performance of this implementation. The algorithm was published by Shi Li in 2021 and is a $(6+\varepsilon)$-approximation algorithm for the multiprocessor problem $P|\text{prec}|\sum_j w_j C_j$. We have implemented a version which is a $(2+\varepsilon)$-approximation algorithm for the single processor problem $1|\text{prec}|\sum_j w_j C_j$. This special case can easily be generalised to the multiprocessor case, as the two algorithms are based on the same LP relaxation of the problem. Unlike other approximation algorithms for this and similar problems, for example, those published by Hall, Schulz, Shmoys and Wein in 1997, and by Li in 2020, this algorithm has been developed with a focus on obtaining a good asymptotic run time guarantee, rather than obtaining the best possible guarantee on the quality of solutions. Li's algorithm has run time

$$O\Big((n+\kappa)\cdot \text{polylog}(n+\kappa)\cdot \log^3 p_{\max}\cdot \frac{1}{\varepsilon^2}\Big),$$

where n is the number of jobs, κ is the number of precedence constraints and $p_{\max}$ is the largest of the processing times of the jobs. We also present a detailed explanation of the algorithm aimed at readers who do not necessarily have a background in scheduling and/or approximation algorithms, based on the paper by Li. Finally, we empirically evaluate how well (our implementation of) this algorithm performs in practice. The performance was measured on a set of 96 randomly generated instances, with the largest instance having 1024 jobs and 32 768 precedence constraints. We can find a solution for an instance with 512 jobs and 11 585 precedence constraints in 25 minutes.

Keywords:

Optimization, scheduling, approximation algorithms, linear programming, multiplicative weight update

URL for electronic version:

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-190178

Sammanfattning

Vi presenterar en praktisk implementation av en ny approximationsalgoritm för schemaläggning av jobb på en maskin med ordningsbivillkor, under minimering av den viktade summan av sluttider. Algoritmen, som publicerades av Shi Li år 2021, är en $(6 + \varepsilon)$-approximationsalgoritm för multiprocessorproblemet $P|\text{prec}|\sum_j w_j C_j$. Vi har implementerat en version som är en $(2 + \varepsilon)$-approximationsalgoritm för enprocessorproblemet $1|\text{prec}|\sum_j w_j C_j$. Detta specialfall kan enkelt generaliseras till multiprocessorfallet, eftersom de två algoritmerna baseras på samma LP-relaxation av problemet. Till skillnad från andra approximationsalgoritmer för detta och liknande problem, exempelvis de från Hall, Schulz, Shmoys och Wein år 1997, och från Li år 2020, har denna algoritm utvecklats med fokus på att uppnå en bra garanti på asymptotisk körtid, istället för att försöka uppnå den bästa möjliga garantin på lösningarnas kvalité. Lis algoritm har körtid

$$O\Big((n + \kappa) \cdot \text{polylog}(n + \kappa) \cdot \log^3 p_{\max} \cdot \frac{1}{\varepsilon^2}\Big),$$

där n är antalet jobb, κ antalet ordningsbivillkor och $p_{\max}$ är den största körtiden bland jobben. En detaljerad beskrivning av algoritmen riktad till personer som inte nödvändigtvis har förkunskaper inom schemaläggning och/eller approximationsalgoritmer, baserad på artikeln, ges också. Slutligen utvärderar vi empiriskt hur väl (vår implementation av) denna algoritm presterar i praktiken. Implementationens egenskaper mättes på en uppsättning av 96 slumplässigt genererade instanser, där den största instansen har 1024 jobb och 32768 ordningsbivillkor. Med vår implementation kan vi hitta en lösning för en instans med 512 jobb och 11 585 precedencensbivillkor på 25 minuter.

Nyckelord:

Optimering, schemaläggning, approximationsalgoritmer, linjärprogrammering

URL för elektronisk version:

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-190178

Acknowledgements

I would like to thank my supervisors, Hannes Uppman and Christiane Schmidt, for helping me understand the subject and giving invaluable feedback on my work. Further, I want to thank Hannes for providing a detailed explanation of, and suggestions on how to implement, the more difficult parts of the algorithm, upon which much of my code is based, as well as his outstanding help with debugging the final implementation.

I want to thank Saab AB in Linköping for giving me the opportunity to work on this thesis. I also want to thank my examiner Elina Rönnberg and my opponents Mikaela Lindberg and Lina Larsson for their invaluable help, and for their patience with me.

Finally, I would like to thank my family and friends, for their support and encouragement during my studies and especially during my work on this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this work, we implement an efficient algorithm for solving a certain scheduling problem. This problem has many similarities to scheduling problems that appear in practical applications, in that it has precedence constraints and a quite complex objective function. Many practical problems that are interesting to solve in the industry, for example at Saab AB (where this work was carried out), are of a similar nature. We are thus interested in being able to solve this problem, and more complex problems like it, quickly.

One candidate for a quick method for finding solutions is an approximation algorithm. This is a type of algorithm that is guaranteed to run in polynomial time, and produce a solution that is not worse than some guarantee. For the algorithm that we implement, where the approximation ratio is $2+\varepsilon$, the solutions never have more than $2+\varepsilon$ times as large objective value as an optimal solution. The algorithm we implement is an approximation algorithm that runs in almost nearly linear time. This algorithm is very recent (from 2021), and unlike most other approximation algorithms for this kind of problem, it has been developed with a focus on obtaining a low asymptotic run time, rather than obtaining the best possible approximation ratio.

However, while this approximation algorithm is fast in theory, this is only an asymptotic guarantee, meaning that the algorithm is not necessarily faster than, for example, a good heuristic. Therefore it is interesting to evaluate empirically how well an actual implementation of the approximation algorithm in question performs. We do this using a set of randomly generated instances.

1.1 Background

Here we introduce the main subjects that we will work with, namely scheduling and approximation algorithms. We begin by giving a short overview of the mathematical fields of scheduling and approximation algorithms, and end with a few examples of applications of scheduling.

Scheduling In scheduling, we typically have a set of jobs, which could, for example, be small computer programs or manufacturing steps in a factory. Each job has a processing time, which is the time it takes to complete after starting it. The jobs may also be related by a number of precedence constraints, which represent dependencies between jobs. For example, if a program requires as input the output of another program or if a manufacturing step requires some part manufactured by another step, then there is a precedence constraint between the corresponding jobs, i.e., the former job can only be started after the latter job has been completed.

We may have one or more machines, which could be for example processor cores or manufacturing tools. Processor cores are a typical example of what is called identical machines while manufacturing tools are an example of unrelated machines. For unrelated machines, the processing time of a job is allowed to depend on which machine the job is scheduled on. We are interested in the case of identical machines, where the processing time only depends on the job and not on the machine.

In some cases, the jobs are allowed to be paused and resumed at some later time, possibly on a different machine. This is known as preemption. However, we are only interested in the case when jobs complete in their entirety after starting.

There are several different properties which one may want to optimise. The time at which a job j completes is known as its completion time, usually denoted C_j. A natural objective function is to minimise the total time before all jobs are completed, that is, to minimise $\max_j C_j$. This is known as the makespan and is denoted $C_{\max}$. Another reasonable objective is to minimise the average completion time $\sum_j C_j/n$, where n is the total number of jobs. This is equivalent to minimising the total completion time, $\sum_j C_j$, as n is constant for a given set of jobs.

If certain jobs are deemed more important than others, we might want to ensure that these complete quicker. This could be achieved by introducing a weight w_j for each job j, with a higher value representing that we are more interested in scheduling the job earlier. In this case, we minimise the total weighted completion time $\sum_j w_j C_j$.

It is also possible to have additional constraints on the schedule. For example, a scheduling problem might have release times and/or deadlines. A release time r_j for a job constrains it to start no earlier than some given point in time, $r_j \leq C_j - p_j$. Likewise, a deadline d_j constrains a job to end no later than some given point in time, $C_j \leq d_j$.

Approximation Algorithms Solving these scheduling problems is computationally difficult, as many other combinatorial optimisation problems are. However, in many applications of scheduling, we are often not interested in finding an optimal solution, but simply a solution that is "good enough". For solving such a problem in practice, an approximation algorithm can be a useful tool. An approximation algorithm is an algorithm which provides an approximate solution to the problem in question, meaning a solution that is feasible, and has an objective value that is within some given bound of the optimal objective value.

For a minimisation problem, this is formulated as the criterion that the algorithm provides a solution with objective value OBJ such that $\text{OBJ} \leq \alpha \cdot \text{OPT}$, where OPT is the optimal objective value. The factor α is called the *approximation ratio* and is at least 1 for minimisation problems. For a maximisation problem, the criterion is that $\text{OBJ} \geq \alpha \cdot \text{OPT}$, and the approximation ratio is at most 1. Note that if $\alpha = 1$, the approximation algorithm gives an optimal solution.

If the approximate solution has an objective value close to the optimal objective value, then the solution could possibly be used directly. If not, it could be used as a starting point when trying to find better solutions by another method.

An approximation algorithm should complete within a run time that does not grow too fast, as a function of the size of the input. In general, it should be a polynomial-time algorithm. Some problems may be difficult, and we may not be able to find an optimal solution in any reasonable time. If we have an approximation algorithm for the problem, we can be certain that its output will always have an objective value that is, for example, not more than twice as large as the optimal objective value, if $\alpha = 2$ and the problem in question is a minimisation problem.

There are many different methods for finding an approximation algorithm, but one straightforward approach is to *relax* the problem formulation, solve this relaxed version, and then *round* the obtained solution. For example, if we have some integer program (IP) to solve, we could relax this to a linear program (LP). Then, if we can find a rounding rule that guarantees that any feasible solution of the LP is rounded to a feasible solution of the IP, we are done. To formally call this an approximation algorithm, we would also need to prove that the

objective value is within some bounds and that the LP can indeed be solved in polynomial time. An illustrative example of this approach, the *relax-and-round* method, for the *set cover problem* is given in [37, Section 1.2–1.7].

It has been shown that if an efficient algorithm can be found for the scheduling problem we are interested in (see Section 1.2), then an efficient algorithm can also be found for many other computationally hard problems (and vice versa), such as the *boolean satisfiability problem*, the *travelling salesman problem* and the *graph colouring problem*. Problems in this class are called *NP-complete*. Most researchers believe that the NP-complete problems cannot be solved in polynomial time, but proving this is a long-standing open problem.

Another use for approximation algorithms, is as a way to study the "difficulty" of a given problem as compared to other NP-complete problems. Problems for which a good (in some sense; usually with respect to the approximation ratio) approximation algorithm exists, can be said to be "easier" problems than those for which no good approximation algorithm exists. It is possible, in some cases, to prove that any approximation algorithm must necessarily have worse approximation ratio than some bound (given that, e.g., the $P \neq NP$ *conjecture* or the *exponential time hypothesis* hold, depending on the problem). In such a case, we have a concrete measure of the "hardness" of the problem.

Applications of Scheduling Scheduling problems are a class of problems which arise in a wide range of practical applications. Some examples of areas where scheduling can be applied are [26, Chapters 45–51]:

- University timetabling
- Batch production scheduling
- Scheduling in the airline industry
- Bus and train driver scheduling

Another example of an area where scheduling can be applied is in avionics. Avionics (aviation electronics) refers to electronic systems used in aircraft. Modern avionic systems usually consist of a large number of processors working together to perform tasks such as measuring important data (altitude, airspeed, etc.) or controlling components, e.g., as fly by wire systems do. It is important that these processors work together efficiently and reliably, to ensure the safety of the aircraft.

In avionics, it is thus useful to find a schedule for all computer tasks before the application is run, to guarantee that the system works as intended, and has the required performance. An example of an application of scheduling to avionic

scheduling is [18], wherein a *matheuristic* approach is used. This is a combination of *metaheuristics* (a class of heuristics which are not problem-specific) and mathematical programming (a.k.a. optimisation). Matheuristics have some similarities to approximation algorithms, but are slightly more practically oriented. For example, a matheuristic solution method usually gives no formal guarantee on the quality of the solution, as it is a heuristic method.

Our work was carried out at Saab AB in Linköping. As Saab works with the design of avionics, there is an interest in developing efficient methods for solving scheduling problems. Algorithms with good asymptotic performance are interesting because they are useful to be able to solve large-scale problems. Approximation algorithms are more often studied in a theoretical setting, as compared to, e.g., heuristics, which are more closely associated with practical applications. As such, it is interesting for us to see if the algorithm we implement performs well in practice.

1.2 The Scheduling Problem

We are interested in the problem of scheduling to minimise the *total weighted completion time* on *identical machines* with *job precedence constraints*, written $P|\text{prec}|\sum_j w_j C_j$ in the classic three-field notation of [13]. The first field of this shorthand gives the type and number of machines and the middle field contains any extra information, such as the use of precedence constraints here. The third field contains the objective function, in this case using C_j for the completion time and w_j for the weight of a job j.

This is a quite basic scheduling problem, with the only constraint, besides non-overlapping placement of jobs, being precedence constraints. The objective function, minimising weighted completion time, is however slightly more difficult to optimise compared to, for example, minimising makespan, as we will discuss in Section 1.4.

Although we present the identical-machine problem, in much of this thesis we focus on the simpler problem of minimising total weighted completion time on a *single machine* with job precedence constraints, $1|\text{prec}|\sum_j w_j C_j$. The two problems are identical, except that the latter has the number of machines fixed to 1. Our implementation of the algorithm is restricted to the simpler case of the single machine problem. The former problem is defined as follows.

Definition 1.1. A *number of machines* m and a set J of n *jobs* are given. Each job $j \in J$ has a given *processing time* $p_j \in \mathbb{Z}_{>0}$ and *weight* $w_j \in \mathbb{Z}_{\geq 0}$. A binary relation $\prec$ over J is also given, called the *precedence relation*. The precedence constraint $j \prec j'$ indicates that job j must complete before job j' can start.

Completion times $(C_j)_{j\in J}$ are to be found, which minimise the objective function

$$\sum_{j\in J} w_j C_j, \tag{1.1}$$

i.e., the *weighted sum of completion times*. The solution $(C_j)_{j\in J}$ should satisfy the constraints

$$C_j \geq p_j \qquad \forall j \in J, \tag{1.2}$$

$$C_j \leq C_{j'} - p_{j'} \qquad \forall j, j' \in J : j \prec j', \tag{1.3}$$

$$t \in (C_j - p_j, C_j] \text{ for at most } m \text{ jobs } j \in J \qquad \forall t \geq 0. \tag{1.4}$$

A solution $(C_j)_{j\in J}$ can be interpreted as meaning that the job j is scheduled in the time interval $(C_j - p_j, C_j]$. The interval is open at one end to allow a job j' to start at the same time point another job j ends, that is, to allow $C_{j'} - p_{j'} = C_j$. We can remark that if this was not allowed, there would be a problem, as $C_{j'} - p_{j'} = C_j + \varepsilon$ would only be a valid solution for $\varepsilon > 0$, but $w_j C_j + w_{j'} C_{j'}$ decreases as $\varepsilon \to 0^+$, so there would exist no solution minimising the objective function. We will always let the interval be open at the start, but this is simply a convention and either end can be chosen.

The first two constraints are relatively straightforward. Constraint (1.2) ensures that no jobs complete before their processing time has elapsed. Constraint (1.3) ensures that if there is a precedence constraint $j \prec j'$, then j' does not start before j has been completed.

Constraint (1.4) can be understood by visualising the timeline of jobs as in Figure 1.1. It ensures that for each time t, the number of jobs c_t that are "running" at the moment is at most m.

It is not obvious that the intervals $(C_j - p_j, C_j]$ for $j \in J$ can each be assigned to a machine without overlap, given that $c_t \leq m$ for every $t \geq 0$. There could hypothetically be a situation in which a job has to run on one machine at time t and another machine at time t', which is not allowed. A folklore result, which we prove in Section 2.2.2, says that it is in fact possible to assign jobs to machines without issue, given that we have a schedule that fulfills (1.2)–(1.4).

We can also note that there is no constraint ensuring that the completion times are integer valued. However, if the processing times are integer valued, then there is an optimal solution with integer completion times. See Section 2.2.1 for a proof. This proof further shows that if $(C_j)_{j\in J}$ fulfill the constraints of the problem, then $(\lfloor C_j \rfloor)_{j\in J}$ also fulfill the constraints. The objective value for the latter solution is not larger than that of the former, also according to Section 2.2.1.

The single-machine scheduling problem can also be expressed in a slightly simpler way.

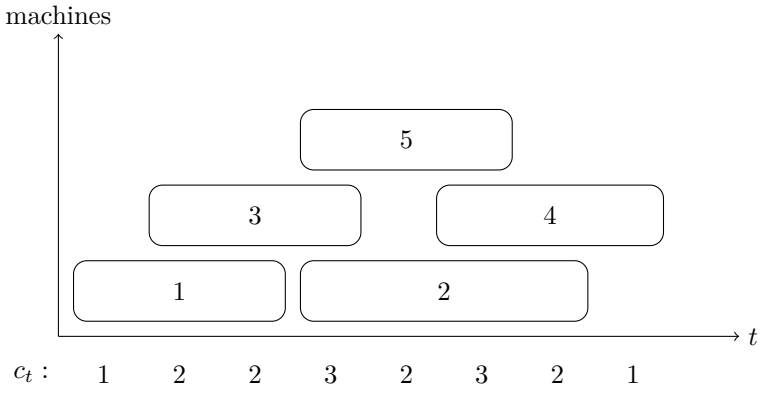


Figure 1.1: A visualisation of a schedule for five jobs on three machines. The rectangles each represent a scheduled job. Constraint (1.4) means that $c_t \leq 3 = m$ for all $t \geq 0$.

Definition 1.2. A set J of n *jobs* is given. Each job $j \in J$ has a given *processing time* $p_j \in \mathbb{Z}_{\geq 0}$ and *weight* $w_j \in \mathbb{Z}_{\geq 0}$. A binary relation $\prec$ over J of *precedence constraints* is given.

Completion times $(C_j)_{j \in J}$ are to be found, which minimise the objective function

$$\sum_{j \in J} w_j C_j. \tag{1.5}$$

The solution $(C_j)_{j \in J}$ should satisfy the constraints (1.2), (1.3) and

$$C_j \geq C_{j'} + p_{j'} \quad \text{or} \quad C_{j'} \geq C_j + p_j \quad \forall j, j' \in J : j \neq j', \tag{1.4'}$$

called the "disjunctive" constraints.

Finding an optimal solution to the scheduling problem (for both single-machine and multiple-machine variants) is NP-hard [22] [5].

1.3 Overview of the Implemented Algorithm

The algorithm which our implementation is based on was devised by Li in 2021 [28] and is a $(6 + \varepsilon)$-approximation algorithm for the multiple machine problem $P|\text{prec}|\sum_j w_j C_j$. We implement a simpler version which is a $(2 + \varepsilon)$-approximation algorithm for the single machine scheduling problem $1|\text{prec}|\sum_j w_j C_j$. Both algorithms are of the *relax and round* type and are based on the same LP

relaxation of the problem. Thus the only change needed to extend our implementation to the complete algorithm, is a slight modification to the final step (the *rounding algorithm*, as we shall see later). This is described in [28, Algorithm 2, p. 10].

Note that, technically, our implementation has approximation ratio $2+O(\varepsilon)$. However, by using a smaller ε, we can make the term $O(\varepsilon)$ arbitrarily small. Thus, it is justified to say that the algorithm has approximation ratio $2+\varepsilon$, by simply modifying the value of ε at the start of the algorithm. See Section 3.3.1 for more details on the approximation ratio.

The algorithm has a run time of

$$\tilde{O}_\varepsilon((n+\kappa)\log p_{\max}), \tag{1.6}$$

where $p_{\max} := \max_{j\in J} p_j$. See Section 2.3.2 for further explanation of the notation $\tilde{O}_\varepsilon(\cdot)$.

The algorithm that is implemented works by solving a linear programming relaxation of the problem. In essence, we introduce binary decision variables over a set of "allowed" completion times, for each job. To keep the number of variables low, the gap between the "allowed" completion times is larger for later completion times (scaling exponentially).

The linear program is then solved efficiently using a multiplicative weight update (MWU) algorithm. This is a method by which a linear program with many constraints can be solved, by iteratively solving a linear program with only one "difficult" constraint. For more details, see Section 2.7. The algorithm which solves this simpler LP is called the *oracle* $\mathcal{O}$.

The simpler LP can be solved by considering its dual (see Section 2.4.3), which can be considered as a set of network flow problems. See Section 4.1.1. This set also needs to be small enough to give a good run time. To solve one of these flow problems, which has a maximum flow objective, a graph-based algorithm is used to find a minimum cut. Then the maximum-flow minimum-cut theorem is used to show that this can be used to find a solution for the primal linear program.

The algorithm for solving one of these network flow problems uses a so-called *shortcut graph*, which has an edge for each path (in the original graph) from a source to a sink, and an edge for each path (in the support of the current flow) from a sink to a source. It also makes use of a novel data structure called a *handled graph* [28, Definition D.1, p. 28]. We find a *blocking flow* (i.e., a s-t-flow, such that all s-t-paths contain at least one edge saturated by the flow) in this graph, as in Dinitz' algorithm. We do this using a *dynamic tree* (also known as a *link/cut tree*) data structure [34], in order to obtain the required run time guarantee. By repeatedly doing this, we can obtain a maximal flow, or equivalently a minimal cut.

Finally, we solve an LP over the set of results (i.e., flows) obtained from the flow problems, to get a solution to the LP with one "difficult" constraint. This concludes the description of the inner workings of the oracle $\mathcal{O}$. Running the MWU loop over this process, we obtain a solution to our LP relaxation.

All that remains is now to *round* this solution to a solution to the original problem. This is done by a so-called *rounding algorithm*, which in the case of the single machine scheduling problem, is a special case of *list scheduling* [12]. In Figure 1.2, an overview of the process is given as a flowchart.

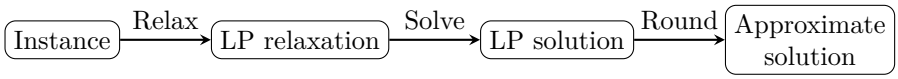


Figure 1.2: Conceptual sketch of the basic parts of the algorithm.

Our implementation is available online at `https://github.com/mansalskog/AlmostLinearApproximateScheduling`. The version described in this report is tagged with `v1.0` in the version history.

1.4 Related Work

Scheduling problems in various forms have been studied for a long time. The special case with the objective being minimisation of the total weighted completion time, $\sum_j w_j C_j$, is a particularly difficult problem compared to for example the problem of minimising the makespan, $C_{\max}$ [16, p. 513]. Some details of this difficulty will be discussed in this section.

Precedence constraints are also a difficult aspect of scheduling problems, especially when designing approximation algorithms. In [31, p. 367], five examples of difficult scheduling problems with precedence constraints are given.

Minimising the Makespan The first example is that of $P|\text{prec}|C_{\max}$, for which the first approximation algorithm is essentially given by Graham in 1966 [12]. It has approximation ratio $\alpha = 2 - \frac{1}{n}$. This method was thirty years old at the time of [31] and even after close to sixty years, it remains the best approximation algorithm for the problem. There can be no approximation algorithm for this problem with an approximation ratio better than 4/3, unless P = NP [25].

Graham's algorithm is based on having a list of jobs L. Whenever a machine completes a job (or at the very start), it starts the next job in the list L which has not been scheduled already, and does not have any precedence constraints preventing it from being scheduled at this time. If there is no such job, the

machine is idle until another machine completes a job, at which point we check again for an available job.

The total weighted completion time is a more general objective than the makespan, i.e., each problem instance for the latter can be represented as an instance for the former. To show this we add a "dummy" job j^* with $p_{j^*} = 1$, $w_{j^*} = 1$ and $j \prec j^*$ for all $j \in J$. Then we define the weights for $j \in J$ as $w_j = 0$, which gives us

$$\sum_{j \in J \cup \{j^*\}} w_j C_j = w_{j^*} C_{j^*} = 1 \cdot (C_{\max} + 1),$$

as the job j^* must end exactly one time unit after all other jobs have completed.

In 2010, it was shown by Svensson that it is NP-hard to approximate $P|\text{prec}|C_{\max}$ within a factor of $2 - \varepsilon$ for any $\varepsilon > 0$, assuming a variant of the *unique games conjecture* [35], a conjecture about hardness of approximation [20]. This was done by showing that $P|\text{prec}|C_{\max}$ is at least as hard as $1|\text{prec}|\sum_j w_j C_j$, for which Bansal and Khot have shown that there exists no $(2-\varepsilon)$-approximation algorithm, assuming the previously mentioned variant of the unique games conjecture [3].

By the reasoning above, this also holds for $P|\text{prec}|\sum_j w_j C_j$. In fact, no better hardness results are known for $P|\text{prec}|\sum_j w_j C_j$ than for $P|\text{prec}|C_{\max}$, i.e., there might exist a greater lower bound on the approximation ratio for the former, but no such bound has been found yet [27, p. 284].

Linear-ordering formulation The best known approximation ratio for $1|\text{prec}|\sum_j w_j C_j$ is 2, presented by Hall, Schulz, Shmoys and Wein in 1997 [16]. A number of approximation algorithms are presented by Hall, Schulz, Shmoys and Wein, for different variations on the scheduling problems. The basis of their scheduling algorithm for $1|\text{prec}|\sum_j w_j C_j$ is replacing the constraint (1.4′) by

$$\sum_{j \in J'} p_j C_j \geq \frac{1}{2}\Big(\sum_{j \in J'} p_j^2 + \big(\sum_{j \in J'} p_j \big)^2 \Big) \quad \forall J' \subseteq J, \tag{1.7}$$

which (it is shown) implies that

$$C_j \geq (1/2) \sum_{j' \in J : C_{j'} \leq C_j} p_{j'},$$

for all $j \in J$. This does not necessarily give a feasible schedule, as the intervals in which the jobs are scheduled are not constrained to be disjoint, which would require

$$C_j \geq \sum_{j' \in J : C_{j'} \leq C_j} p_{j'} \quad \text{for all } j \in J.$$

Hall et al. then solve a linear relaxation of $1|\text{prec}|\sum_j w_j C_j$ (cf. Definition 1.2) in which $\sum_{j\in J} w_j C_j$ is minimised subject to (1.3) and (1.7), to obtain fractional completion times $(\bar{C}_j)_{j\in J}$. This can be done in polynomial time using the *ellipsoid algorithm* for convex linear programs [30], even though the number of constraints is exponential in the number of jobs. By scheduling the jobs in order of nondecreasing $\bar{C}_j$ and breaking ties by using the precedence relation, a schedule that is within a factor 2 of the optimal schedule is obtained [16, p. 519].

This approximation scheme can be used in conjunction with other LP formulations of the scheduling problem. One such formulation is based on variables $(\delta_{i,j})_{i,j\in J:i\neq j}$, which symbolise that job i precedes job j in the schedule if $\delta_{i,j} = 1$. This is called a linear-ordering formulation. By using this, the fractional completion times $(\bar{C}_j)_{j\in J}$ can be found while avoiding the ellipsoid algorithm. Hall et al. also consider a time indexed formulation [9], which has variables $(x_{j,t})_{j\in J,t\in[0,T]}$ where $T = \sum_{j\in J} p_j$. If a job j completes at a point in time t, then $x_{j,t} = 1$.

Graph-based formulation The earliest approximation algorithm for $P|\text{prec}|\sum_j w_j C_j$ was published by Ravi, Agrawal and Klein in 1991 and had an approximation ratio of $O(\log n \log \sum w_j)$ [32]. The approach here was to represent the problem instance as a (directed acyclic) graph $G = (J, E_J)$ where $E_J = \{(j, j') : j \prec j'\}$. Then the scheduling problem is solved by finding a topological ordering $j_1, \ldots, j_n$ of G such that the minimum of $\sum_{j\in J} w_j C_j$, where

$$C_{j_k} = \sum_{l=1}^{k} p_l,$$

over all topological orderings of G, is attained. The problem is reduced to another scheduling problem, that of minimising the *storage-time product*, by adding some nodes and edges. This problem is then solved using a recursive algorithm on the graph [32, pp. 755–759].

The method was expanded on by Even, Naor, Rao and Schieber in 1995 [10] to obtain a $O(\log n \log\log \sum w_j)$-approximation algorithm. This was done by improving the graph algorithm for minimising the storage-time product [10, pp. 607–608].

Time indexed formulation For $P|\text{prec}|\sum_j w_j C_j$, the best known approximation ratio is $2 + 2\log 2 + \varepsilon \approx 3.4 + \varepsilon$, which was shown by Li in 2020 using a time indexed LP relaxation of the problem [27]. This improved on the earlier best, a 4-approximation algorithm by Munier, Queyranne and Schulz [31] [29].

Asymptotic time complexity All the results mentioned above have a focus on finding the smallest possible approximation ratio in polynomial time. The article by Li [28] differs in that the focus when developing the algorithm was obtaining a low time complexity. The time complexity of Li's algorithm is almost nearly linear (see Definition 2.4), which is quite low in comparison with the other algorithms mentioned. Most of them do not specify the run time further than that it is polynomial. There is a trade-off however, as the approximation ratio, $6+\varepsilon$, is not particularly good when compared with the best known, $2+2\log 2+\varepsilon$.

Linear programming Since many approximation algorithms involve solving a linear relaxation, it is necessary to have a fast algorithm for solving LPs. It has been shown by Klee and Minty in 1972 that the simplex algorithm in its original formulation has exponential time complexity, as it needs to visit all 2^m vertices of a problem with m constraints in the worst case [21]. By using a different pivoting rule it is possible to improve the performance of the simplex algorithm. A deterministic pivot rule leading to sub-exponential time complexity is known, but it is an open question whether there exists a deterministic pivot rule leading to polynomial time complexity [17].

The ellipsoid method was introduced in 1979 by Khachiyan [19], as the first polynomial-time algorithm for solving linear programs. It built on the work of Shor [33], as well as Nemirovski and Yudin [38]. The interior point method is another polynomial time algorithm for solving linear programs.

Some recent fast algorithms for solving general LPs include one by Lee and Sidford with run time $\tilde{O}((N+m^2)\sqrt{m}\log(1/\varepsilon))$ [23] and one by Lee, Song and Zhang with run time (approximately) $\tilde{O}(\bar{n}^{2.373}\log(1/\varepsilon))$ [24], for a LP with a constraint matrix of size $m \times n$ with N nonzero elements.

Multiplicative weight update Another way to solve linear programs is by using the multiplicative weight update (MWU) algorithm. MWU is an algorithm which has applications in many different areas. Examples include machine learning, game playing problems in economics, and convex optimisation [2, p. 122]. We are interested in its application to solving packing linear programs. Packing problems can be loosely described as problems of choosing a limited number of items to maximise their total value.

There has been much research into using the MWU algorithm to solve linear packing feasibility problems, such as deciding if there exists an $\mathbf{x} \in P$ such that $\mathbf{Ax} \leq \mathbf{b}$ for some given $P \subseteq \mathbb{R}^n_{\geq 0}$, $\mathbf{A} \in \mathbb{R}^{m\times n}$ with nonnegative elements and $\mathbf{b} \in \mathbb{R}^m$ [2, p. 135]. Usually, we are interested in approximate solutions, i.e., finding an $\mathbf{x} \in P$ such that $\mathbf{Ax} \leq (1+\varepsilon)\mathbf{b}$ for some $\varepsilon > 0$.

1.5 Outline of the Report

In Chapter 2 we give an overview of the mathematical concepts needed to understand the algorithm and our implementation of it. We leave out most proofs, as they are readily found in other literature. The main parts of the report are Chapters 3 and 4. In Chapter 3 we explain the overarching structure of the algorithm, giving a complete description of how the problem is solved, but leaving out the description of a helper function for solving a certain network flow problem. This helper function is the subject of Chapter 4, wherein its implementation is described. Many of the more technical details are left out in the interest of brevity. In Chapter 5, we present the results of the evaluation of our implementation. Finally, in Chapter 6, we discuss the results and consider possible extensions and future work in the area. Appendix A contains some small technical details which are not essential to understanding Chapters 3–4, but which nonetheless are interesting to discuss.

Chapter 2

Preliminaries

In this chapter we describe some basic results in scheduling and linear programming, which will be used in the implementation. We explain the basics of the multiplicative weight update method and parts of the simplex method. We also prove two facts about the scheduling problem in question which we have mentioned in the introduction. Finally, we give some definitions relating to graphs and show how linear programming duality gives the the max-flow min-cut theorem.

2.1 Notation

The set of real numbers is denoted $\mathbb{R}$, and the set of integers $\mathbb{Z}$. We use the notation $A_{\geq x_0} := \{x \in A : x \geq x_0\}$, for example, $\mathbb{R}_{\geq 0} = \{x \in \mathbb{R} : x \geq 0\}$. $A_{<x_0}, A_{\leq x_0}, A_{>x_0}$ are defined analogously.

The interval notation $[a, b]$ is used to mean either

$$\{x \in \mathbb{R} : a \leq x \leq b\}$$

or

$$\{k \in \mathbb{Z} : a \leq k \leq b\},$$

when the intended meaning is clear from the context. The notation $[n] := \{1, \ldots, n\}$ is used for integers only. Open and half-open intervals such as

$$(a, b] := \{x \in \mathbb{R} : a < x \leq b\}$$

are used for real numbers only.

Vectors are denoted with boldface letters, for example, $\mathbf{a}$. For a vector $\mathbf{a} \in \mathbb{R}^A$ for some finite set A, coordinates are denoted a_i for $i \in A$. A vector is assumed to be a column vector, unless otherwise stated. The vector space $\mathbb{R}^n$ is the same as $\mathbb{R}^{[n]}$, meaning the coordinates of $\mathbf{a} \in \mathbb{R}^n$ are $a_1, a_2, \ldots, a_n$. If $\mathbf{a} \in \mathbb{R}^A_{\geq 0}$ is a vector over some finite set A, then we define

$$a(B) := \sum_{i \in B} a_i,$$

for any $B \subseteq A$. For two vectors $\mathbf{a}$ and $\mathbf{b}$ over the same space $\mathbb{R}^A$, $\mathbf{a} \leq \mathbf{b}$ means

$$a_i \leq b_i \quad \text{for all } i \in A,$$

and likewise for the other inequalities $<, \geq, >$.

Matrices are written with uppercase boldface letters, e.g., $\mathbf{A} \in \mathbb{R}^{m \times n}$. $\mathbf{A}_i \in \mathbb{R}^n$ denotes row $i \in [m]$ of $\mathbf{A}$ as a row vector.

We write standard norms with single vertical bars, for example

$$|\mathbf{x}| := |\mathbf{x}|_2 := \sqrt{\sum_{k=1}^{n} x_k^2}$$

and

$$|\mathbf{x}|_1 := \sum_{k=1}^{n} |x_k|,$$

for any vector $\mathbf{x} \in \mathbb{R}^n$.

2.2 Scheduling

In this section we prove that we can, without loss of generality, assume that there exists integer-valued completion times for our problem (Definition 1.1) with optimal weighted completion time. We also prove the fact that the constraint $c_t \leq m$ (see (1.4)) is enough to guarantee that each job can be run without interruption.

2.2.1 Integer Completion Times

Here we prove that the optimal completion times for our scheduling problem can be chosen to be integer-valued given that the processing times are. Consider some solution $(C_j)_{j \in J}$. If $p_j \in \mathbb{Z}_{>0}$ then

$$C_j \geq p_j \quad \Longrightarrow \quad \lfloor C_j \rfloor \geq \lfloor p_j \rfloor = p_j$$

and

$$\begin{aligned} C_j \le C_{j'} - p_{j'} \quad &\iff \quad C_{j'} - C_j \ge p_{j'} \\ &\implies \quad \lfloor C_{j'} \rfloor - \lfloor C_j \rfloor \ge \lfloor C_{j'} - C_j \rfloor \ge p_{j'} \\ &\implies \quad \lfloor C_j \rfloor \le \lfloor C_{j'} \rfloor - p_{j'}. \end{aligned}$$

Finally, if for all $t \ge 0$,

$$t \in (C_j - p_j, C_j] \text{ for at most } m \text{ jobs } j \in J$$

then for all $t \ge 0$,

$$t \in (\lfloor C_j \rfloor - p_j, \lfloor C_j \rfloor] \text{ for at most } m \text{ jobs } j \in J.$$

This can be proven by assuming, towards contradiction, that there exists a $t \ge 0$ and jobs $j_1, \ldots, j_{m+1}$ such that

$$t \in (\lfloor C_{j_k} \rfloor - p_{j_k}, \lfloor C_{j_k} \rfloor] \quad \forall k \in [m+1],$$

which is the same as assuming that the intersection

$$\bigcap_{k=1}^{m+1} (\lfloor C_{j_k} \rfloor - p_{j_k}, \lfloor C_{j_k} \rfloor]$$

is nonempty, or in other words its length is positive,

$$\min_k \lfloor C_{j_k} \rfloor - \max_k (\lfloor C_{j_k} \rfloor - p_{j_k}) > 0, \tag{2.1}$$

and because the length is an integer, it must be at least 1. Now let

$$\delta_k = C_{j_k} - \lfloor C_{j_k} \rfloor \in [0, 1) \quad \text{for } k \in [m+1].$$

Then, we know that

$$\min_k C_{j_k} = \min_k (\lfloor C_{j_k} \rfloor + \delta_k) = \lfloor C_{j_{k^*}} \rfloor + \delta_{k^*},$$

for some k^* such that $\lfloor C_{j_{k^*}} \rfloor = \min_k \lfloor C_{j_k} \rfloor$. Likewise, we have

$$\begin{aligned} \max_k (C_{j_k} - p_{j_k}) &= \max_k (\lfloor C_{j_k} \rfloor - p_{j_k} + \delta_k) \\ &= \lfloor C_{j_{k^\circ}} \rfloor - p_{j_{k^\circ}} + \delta_{k^\circ}, \end{aligned}$$

for some k° such that $\lfloor C_{j_{k^\circ}} - p_{j_{k^\circ}} \rfloor = \max_k(\lfloor C_{j_k} \rfloor - p_{j_k})$. Thus, by (2.1), we get

$$\begin{aligned}\min_k C_{j_k} - \max_k (C_{j_k} - p_{j_k}) &= (\lfloor C_{j_{k^*}} \rfloor + \delta_{k^*}) - (\lfloor C_{j_{k^\circ}} \rfloor - p_{j_{k^\circ}} + \delta_{k^\circ}) \\ &= \underbrace{\lfloor C_{j_{k^*}} \rfloor - \lfloor C_{j_{k^\circ}} \rfloor + p_{j_{k^\circ}}}_{\geq 1} + \underbrace{\delta_{k^*} - \delta_{k^\circ}}_{\in(-1,1)} > 0,\end{aligned}$$

which finally gives us (by the same reasoning as above) that

$$t \in (C_{j_k} - p_{j_k}, C_{j_k}] \quad \forall k \in [m+1],$$

which is a contradiction.

In conclusion, if $(C_j)_{j\in J}$ is a solution, then $(\lfloor C_j \rfloor)_{j\in J}$ is a solution, with objective value $\sum_{j\in J} w_j \lfloor C_j \rfloor \leq \sum_{j\in J} w_j C_j$. Thus, we can deduce that there exists an optimal solution that has integer completion times, if there exists any optimal solution.

2.2.2 Congestion

We claimed in Section 1.2 that if we have a set of completion times $(C_j)_{j\in J}$ such that c_t, the number of jobs running at time t, is not greater than m for each $t \geq 0$, then it is possible to schedule the jobs J on m machines without overlap, such that the jobs complete at the given completion times.

We prove this by finding the mapping $M\colon J \times \mathbb{R}_{\geq 0} \to [m] \cup \{\perp\}$, giving the machine to run job $j \in J$ on at time $t \geq 0$, such that

$$\begin{aligned} M(j,t_1) = M(j,t_2) \in [m] \qquad & \forall j \in J, t_1, t_2 \in (C_j - p_j, C_j], \\ M(j,t) = \perp \qquad & \forall j \in J, t \notin (C_j - p_j, C_j]. \end{aligned}$$

The value $\perp$ represents that the job is not running at the given time. To find this, first let J_t denote the set of jobs running at time $t \geq 0$. Let $t_1, \ldots, t_K$ be the completion times together with the start times, sorted and with duplicates removed, i.e.,

$$t_1 < t_2 < \ldots t_{K-1} < t_K$$

and

$$\{t_k : k \in [K]\} = \{C_j : j \in J\} \cup \{C_j - p_j : j \in J\}.$$

Let $t_0 := 0$. For any time interval $(t_{k-1}, t_k]_{k\in[K]}$, we know that

$$J_t = J_{t'} \quad \forall t, t' \in (t_{k-1}, t_k],$$

because no jobs can complete or start "in the middle of" the interval. We have also assumed that

$$|J_t| = c_t \leq m \quad \forall t \geq 0.$$

Now we start by setting

$$M(j,t) := \perp \quad \forall j \in J, t \in \mathbb{R}_{\geq 0} : t \notin (C_j - p_j, C_j].$$

We proceed by induction on $k = 1, 2, \ldots$. For $k = 1$, we can arbitrarily assign the jobs J_{t_1} to the machines $[m]$, giving us $M(j, t_1)$ for all $j \in J$. For a $k > 1$, first set

$$M(j, t_k) := M(j, t_{k-1}) \quad \text{for all } j \in J_{t_k} \cap J_{t_{k-1}}.$$

Now we have assigned $|J_{t_k} \cap J_{t_{k-1}}|$ jobs, leaving us with $m - |J_{t_k} \cap J_{t_{k-1}}|$ machines available and $|J_{t_k} \setminus J_{t_{k-1}}|$ jobs to assign. Basic set theory tells us that

$$c_{t_k} = |J_{t_k}| = |J_{t_k} \cap J_{t_{k-1}}| + |J_{t_k} \setminus J_{t_{k-1}}|,$$

so

$$|J_{t_k} \setminus J_{t_{k-1}}| = c_{t_k} - |J_{t_k} \cap J_{t_{k-1}}| \leq m - |J_{t_k} \cap J_{t_{k-1}}|,$$

meaning that we have at least as many machines available as we have jobs left. Thus we can arbitrarily assign the jobs $J_{t_k} \setminus J_{t_{k-1}}$, i.e., the jobs which start running during the interval $(t_{k-1}, t_k]$, to the available machines.

2.3 Asymptotic Run Time

When we say that a certain algorithm has *time complexity* $O(g(n))$ for some function g, we mean that for the asymptotic running time of the algorithm for an input of size n, $T(n)$, the relation $T(n) = O(g(n))$ holds.

Definition 2.1 (Big O notation)**.** The notation $f(n) = O(g(n))$ for some functions $f\colon \mathbb{Z}_{\geq 0} \to \mathbb{R}_{\geq 0}$ and $g\colon \mathbb{Z}_{\geq 0} \to \mathbb{R}_{\geq 0}$ means that there exists some M and n_0 such that

$$f(n) \leq Mg(n) \quad \text{for all } n \geq n_0.$$

Further, the notation $f(n) = \text{poly}(n)$ means that $f(n) = n^{O(1)}$, i.e., there exists some $N > 0$ such that $f(n) = O(n^N)$. In this case, we say that f is *polynomial* in n.

If $f(n) = \text{poly}(\log n)$ we say that f is *polylogarithmic* in n and write $f(n) = \text{polylog}(n)$.

There are numerous results relating to these concepts, for example, the following lemma which simplifies some calculations.

Lemma 2.2. *If* $f(n) = \text{poly}(n)$ *then* $\log f(n) = O(\log n)$.

Proof. By definition, $f(n) = \text{poly}(n)$ states that there are some M, n_0 and N such that $f(n) \leq Mn^N$ for all $n \geq n_0$. We assume w.l.o.g. that $n_0 \geq 3 > e$ and $M \geq 1$. As the logarithm function is strictly increasing,

$$\log f(n) \leq \log(Mn^N) = \log M + N \log n \leq M' \log n,$$

where $M' := \log M + N$, for all $n \geq n_0$ which is the definition of $\log f(n) = O(\log n)$. □

In fact, polylogarithmic functions grow slowly compared to polynomial functions, as we can see in the following theorem.

Theorem 2.3. *Assume that* $f(n) = \text{polylog}(n)$. *Then, for any* $\varepsilon > 0$ *and any* $M > 0$, *there exists an* n_0 *such that*

$$|f(n)| \leq Mn^\varepsilon \quad \textit{for all } n \geq n_0.$$

In particular, $f(n) = O(n^\varepsilon)$.

For a proof see [7, pp. 53–54]. This can be described as the fact that any polylogarithmic function grows slower than any polynomial function, and motivates the introduction of the concept of nearly linear time, which we will discuss in Section 2.3.2.

2.3.1 Machine Models

The running time of an algorithm can be measured in various different ways. For instance, the number of seconds for an actual program to complete is one way to measure it. Another way to measure the run time is as the number of steps, or instructions, an *abstract machine* has to perform to run the algorithm. This is useful because the number of seconds to execute is very machine-dependent, but the number of steps needed is more machine-independent. As the performance of the physical machines we use is constantly changing, it is valuable to be able to evaluate an algorithm in a way that will also be valid in the future. If the abstract machine we use is similar enough to the actual physical machines, then the theoretical performance of an algorithm can relatively easily be compared to, and used to predict, the performance in practice.

The well-known *Turing machine* [36, pp. 231–246] was the first abstract machine to be introduced. We will however primarily work with another abstract machine called the *random-access machine (RAM)*. This model is more similar to the physical computers which are used today, compared to the Turing machine. One definition of a random access machine is found in [1, Appendix 1, pp.

189–190]. In essence, the machine consists of a *store* which can hold a number of *words*, which are sequences of some number of bits, together with a set of *instructions*. The number of bits in a word is limited by some function of the input size, typically $O(\log n)$. The machine runs *programs*, which are finite sequences of instructions. The instructions available are typically basic mathematical operations, e.g., plus, minus, and multiplication, memory access instructions, e.g., load and store, and conditional branching or some other control-flow instructions. All of these instructions take one time step to perform, by definition of the RAM model.

2.3.2 Nearly Linear Time

One interesting class of algorithms is that consisting of algorithms that have linear time complexity if a polylogarithmic factor in the input size is disregarded.

Definition 2.4. The notation $f(n) = \tilde{O}(g(n))$ means that

$$f(n) = O(g(n) \cdot \text{poly}(\log n)).$$

We say that an algorithm runs in *nearly linear time* if its time complexity is $\tilde{O}(n)$ for an input of size n.

This notion of nearly linear time is *robust* in the sense that a RAM can compute a function in nearly linear time if and only if a number of other machine models, namely a frugal nonerasing RAM, a random-access Turing machine (RTM), a frugal nonerasing RTM, a bisecting-and-jumping Turing machine (BTM), a Kolmogorov machine, and a Schönhage machine, can all compute the function in nearly linear time [15, Theorem 1, p. 112]. It is said that these machines can be efficiently simulated by each other. Note that a regular Turing machine is not powerful enough to efficiently simulate the above machines. This means that there exists at least one algorithm which runs in nearly linear time on a RAM (and the other six models), but not in nearly linear time on a Turing machine.

2.4 Linear Programming

We start by giving a general definition of a linear program and an integer (linear) program.

Definition 2.5. A *linear program (LP)* consists of a vector of *variables* $\mathbf{x} \in \mathbb{R}^n$, a list of *constraints* on the form

$$\begin{aligned}\mathbf{A}^{(1)}\mathbf{x} &\leq \mathbf{b}_1,\\ \mathbf{A}^{(2)}\mathbf{x} &= \mathbf{b}_2,\\ \mathbf{A}^{(3)}\mathbf{x} &\geq \mathbf{b}_3,\end{aligned}$$

where $\mathbf{A}^{(k)} \in \mathbb{R}^{m_k \times n}$ and $\mathbf{b}_k \in \mathbb{R}^{m_k}$ for $k = 1, 2, 3$, and an *objective function* $\mathbf{c}^T\mathbf{x}$ for some fixed $\mathbf{c} \in \mathbb{R}^n$. *Solving* the LP means finding a value for $\mathbf{x}$ such that all the constraints are satisfied, and the value of the objective function, the *objective value*, is minimised or maximised, depending on the problem.

Definition 2.6. An *integer program (IP)* is a LP with the additional constraint that all variables take integer values, i.e., $\mathbf{x} \in \mathbb{Z}^n$.

A linear program may have an *optimal solution* with a finite objective value, it may have an *unbounded solution* such that the objective value can get arbitrarily close to $-\infty$ (or to $+\infty$ for a maximisation problem), or it may be *infeasible*, meaning that there is no $\mathbf{x}$ which fulfills the constraints.

All linear programs can be written on a simpler form, called *standard form*.

Theorem 2.7. *Every linear program can be expressed as a problem of finding a vector* $\mathbf{x} \in \mathbb{R}^n$ *that minimises* $\mathbf{c}^T\mathbf{x}$ *and satisfies constraints* $\mathbf{x} \geq \mathbf{0}$ *and* $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ *for some given* $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$, *and* $\mathbf{c} \in \mathbb{R}^n$.

We note that all linear programs can be expressed in this form by some simple transformations. Constraints such as

$$\mathbf{A}_i\mathbf{x} \leq b_i$$

can be written

$$-\mathbf{A}_i\mathbf{x} \geq -b_i,$$

while constraints such as

$$\mathbf{A}_i\mathbf{x} = b_i$$

can be written

$$\begin{cases}\mathbf{A}_i\mathbf{x} \geq b_i,\\ -\mathbf{A}_i\mathbf{x} \geq -b_i.\end{cases}$$

Variables x_i can be replaced by

$$x_i = x_i^+ - x_i^-,$$

where $x_i^+ \geq 0$ and $x_i^- \geq 0$. Finally, an objective to maximise $\mathbf{c}^T\mathbf{x}$ can be replaced by the objective to minimise $-\mathbf{c}^T\mathbf{x}$.

2.4.1 The Simplex Method

The *simplex method* is an algorithm for solving linear programs. The basic idea is to visit vertices in the *polytope* of allowed values. A *polytope* is an n-dimensional generalisation of a *polyhedron*, i.e., a three-dimensional shape with flat faces and straight edges meeting at vertices. For example, the set of points where a finite set of linear inequalities all hold is a polytope (if it is nonempty).

At each step of the simplex algorithm, we move to a vertex that has a better objective value, or if there is no such vertex, declare the problem solved. This works because (as we will see) the optimal value is always attained at a vertex. Consider the example in Figure 2.1, where the polytope of allowed values has vertices A, B, C, D and the objective is to maximise $\mathbf{c}^T\mathbf{x}$. The direction of the vector $\mathbf{c}$ decides at which vertex the maximum is attained. In this case it is at A. If the vector had instead had the direction $\mathbf{c} = [0, -1]^T$, the maximum would be attained at both C and D.

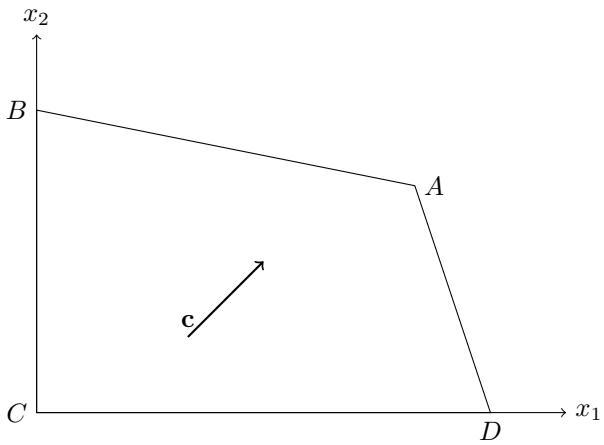


Figure 2.1: An example of a polytope of allowed values for an LP with the objective to maximise $\mathbf{c}^T\mathbf{x}$.

Now assume that we want to solve the general LP

$$\begin{aligned} &\text{maximise} && \mathbf{c}^T\mathbf{x} \\ &\text{subject to} && \mathbf{A}\mathbf{x} \leq \mathbf{b} && (2.2) \\ & && \mathbf{x} \geq \mathbf{0} && (2.3) \end{aligned}$$

for some given $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{c} \in \mathbb{R}^n$. First we introduce *slack variables*

s and rewrite (2.2) as

$$\mathbf{A}\mathbf{x} + \mathbf{s} = \mathbf{b}, \tag{2.4}$$

$$\mathbf{s} \geq \mathbf{0}. \tag{2.5}$$

We also add the unconstrained variable p, called the *profit*. It is set to be equal to the objective value by the constraint

$$p = \mathbf{c}^T\mathbf{x} \iff -\mathbf{c}^T\mathbf{x} + p = 0. \tag{2.6}$$

This gives us the *simplex tableau*:

$$\begin{array}{cccccccc|c} a_{1,1} & \dots & a_{1,n} & 1 & 0 & \dots & 0 & 0 & b_1 \\ a_{2,1} & \dots & a_{2,n} & 0 & 1 & \dots & 0 & 0 & b_2 \\ \vdots & & \vdots & & & \ddots & & & \vdots \\ a_{m,1} & \dots & a_{m,n} & 0 & 0 & \dots & 1 & 0 & b_n \\ \hline -c_1 & \dots & -c_n & 0 & 0 & \dots & 0 & 1 & 0 \end{array}$$

Here we use the notation $a_{i,j}$ for the element at row i and column j of the matrix $\mathbf{A}$. This is used by the simplex method to search among the so-called basic feasible solutions, which correspond to possible solutions, i.e., candidates for the optimal solution.

Definition 2.8. Given an LP on the above form, a *basic solution* is a solution to (2.4) and (2.6), where n of the variables $x_1, \dots, x_n, s_1, \dots, s_m$ are equal to zero, and the columns of $\begin{bmatrix}\mathbf{A} & \mathbf{I}\end{bmatrix}$ corresponding to the nonzero variables are linearly independent. If $\mathbf{x} \geq \mathbf{0}$ and $\mathbf{s} \geq \mathbf{0}$, then the solution corresponds to a point in the feasible region of the LP, and it is called a *basic feasible solution*.

We note that basic feasible solutions correspond exactly to the *extreme points* of the set

$$\left\{ \begin{bmatrix}\mathbf{x} \\ \mathbf{s}\end{bmatrix} : \mathbf{A}\mathbf{x} + \mathbf{s} = \mathbf{b},\ \mathbf{x} \geq \mathbf{0},\ \mathbf{s} \geq \mathbf{0} \right\},$$

where an *extreme point* is a point in the set such that there is no open line segment between two other points in the set, which contains the extreme point. See for example [14, Theorem 4.4, pp. 110–111] for a proof of this property of basic feasible solutions.

Theorem 2.9. *If an LP has a (finite) optimal solution, then it must occur at one (or more) of the basic feasible solutions.*

See for example [14, Theorem 4.7, pp. 121–122] for a proof. This theorem lets us find an optimal solution by simply checking all basic solutions for feasibility, and then comparing their objective values. However, the number of basic

solutions for an LP with n variables and m constraints is, in the worst case (when all columns are linearly independent),

$$\binom{m+n}{n} = \frac{(m+n)!}{n!(m+n-n)!} = \frac{(m+n)!}{n!m!},$$

which may be very large.

Therefore, it is in many contexts interesting to consider a better way to search among the basic solutions. One such strategy is the simplex method. However, we will not explain it in detail here, as we will not need it. Instead we explain a simpler strategy for finding a solution to a special form of LP. The simplex method is a more efficient variant of this.

2.4.2 Solving a Simple LP

Suppose that we want to solve an LP with only two constraints:

$$\begin{aligned} \text{maximise} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{a}^T\mathbf{x} \leq 1, && (2.7) \\ & \mathbf{b}^T\mathbf{x} \leq 1, && (2.8) \\ & \mathbf{x} \geq \mathbf{0}, && (2.9) \end{aligned}$$

where $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^n_{\geq 0}$ are given. We can introduce *slack variables* $s_1, s_2 \geq 0$ and rewrite the constraints (2.7) and (2.8) as

$$\mathbf{a}^T\mathbf{x} + s_1 = 1, \quad (2.7')$$

$$\mathbf{b}^T\mathbf{x} + s_2 = 1, \quad (2.8')$$

which is an equation system with $n+2$ unknowns. Letting

$$\begin{aligned} \hat{\mathbf{x}} &= \begin{bmatrix} \mathbf{x} & s_1 & s_2 \end{bmatrix}^T, \\ \hat{\mathbf{a}} &= \begin{bmatrix} \mathbf{a}^T & 1 & 0 \end{bmatrix}, \\ \hat{\mathbf{b}} &= \begin{bmatrix} \mathbf{b}^T & 0 & 1 \end{bmatrix}, \end{aligned}$$

the equation system becomes

$$\begin{bmatrix} \hat{\mathbf{a}} \\ \hat{\mathbf{b}} \end{bmatrix} \hat{\mathbf{x}} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

However, we know from Theorem 2.9 that there exists an optimal solution for which n of the unknowns are zero. Thus we can choose two indices $k, l \in \{1, \ldots, n+2\}, k < l$ and add the equations

$$\hat{x}_i = 0 \quad \text{for } i = 1, \ldots, n+2,\ i \neq k,\ i \neq l$$

to the system. The solution is then given by solving the system

$$\begin{bmatrix} \hat{a}_k & \hat{a}_l \\ \hat{b}_k & \hat{b}_l \end{bmatrix} \begin{bmatrix} \hat{x}_k \\ \hat{x}_l \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

using Gaussian elimination (or alternatively, finding the inverse matrix). The objective value for some solution $\hat{x}_k, \hat{x}_l$ is given by

$$\mathbf{c}^T\mathbf{x} = \hat{c}_k\hat{x}_k + \hat{c}_l\hat{x}_l, \quad \text{where} \quad \hat{c}_i = \begin{cases} c_i, & 1 \leq i \leq n, \\ 0, & \text{otherwise.} \end{cases} \tag{2.10}$$

By simply iterating over all pairs of k, l and finding the solution with the largest objective value, we can maximise $\mathbf{c}^T\mathbf{x}$ in $O(n^2)$ time. As long as n is small, this performance is good enough for certain applications.

Note that we must take some care to not include solutions where either of $\hat{x}_k$ and $\hat{x}_l$ is negative, as we require $\mathbf{x} \geq \mathbf{0}$ and $\mathbf{s} \geq \mathbf{0}$.

If

$$\det \begin{bmatrix} \hat{a}_k & \hat{a}_l \\ \hat{b}_k & \hat{b}_l \end{bmatrix} = 0, \tag{2.11}$$

either the system has no solution, or the system has infinitely many solutions. However, if (2.11) holds, then the columns k and l of $\begin{bmatrix} \mathbf{A} & \mathbf{I} \end{bmatrix}$, where $\mathbf{A} = \begin{bmatrix} \mathbf{a}^T \\ \mathbf{b}^T \end{bmatrix}$, are linearly dependent, which is not allowed by Definition 2.8. Thus, we can never have a basic solution such that (2.11) holds, so we do not need to consider this case in our implementation. We can simply skip all k and l where the determinant is zero.

2.4.3 Duality

An important concept in linear programming is *duality*. Assume we have an LP on the form in Theorem 2.7, that is

$$\begin{aligned} \text{minimise} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned} \tag{P}$$

or writing the matrix multiplications and vector inequalities using indices

$$\text{minimise} \quad \sum_{i=1}^{n} c_i x_i \tag{2.12}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{i,j} x_j \geq b_i \quad \forall i \in [m], \tag{2.13}$$

$$x_i \geq 0 \quad \forall i \in [n]. \tag{2.14}$$

Multiply both sides of (2.13) with a new variable $y_i \geq 0$ for $i \in [m]$ to get

$$y_i \sum_{j=1}^{n} a_{i,j} x_j \geq y_i b_i \quad \forall i \in [m],$$

which can also be written

$$\mathbf{y}^T \mathbf{A}\mathbf{x} = (\mathbf{A}^T \mathbf{y})^T \mathbf{x} \geq \mathbf{y}^T \mathbf{b}.$$

Now suppose that we can show that $\mathbf{A}^T \mathbf{y} \leq \mathbf{c}$. Then we would have a lower bound $\mathbf{y}^T \mathbf{b}$ for the objective (2.12), and if we can increase $\mathbf{y}^T \mathbf{b}$ we get a better lower bound. This leads us to the *dual*

$$\begin{aligned} \text{maximise} \quad & \mathbf{b}^T \mathbf{y} \\ \text{subject to} \quad & \mathbf{A}^T \mathbf{y} \leq \mathbf{c}, \\ & \mathbf{y} \geq \mathbf{0}. \end{aligned} \tag{D}$$

In this context, the LP (P) is called the *primal*. One can show that the dual of (D) is again (P).

There are two important results relating to duality:

Theorem 2.10 (Weak duality)**.** *Any objective value of a feasible solution of* (P) *is at least as large as any objective value of a feasible solution of* (D)*.*

Proof. The definition of the dual as above directly gives this result. □

Theorem 2.11 (Strong duality)**.** *Given that* (P) *has a (finite) optimal solution,* (D) *has an optimal solution and the optimal objective value for* (D) *is equal to the optimal objective value of* (P)*.*

See for example [14, Theorem 6.9, pp. 180–181] for a proof.

2.4.4 Constructing the Dual

When constructing the dual for a problem, it is tedious to first rewrite the primal on the standard form using Theorem 2.7 and then rewriting the dual on a more convenient form, by, e.g., replacing two inequalities with a corresponding equality. There is a "mechanical" method of constructing the dual, described for the case when the primal is a maximisation problem in Table 2.1. For a minimisation problem, there is a corresponding method but with slightly different details.

Table 2.1: A method for constructing the dual of an LP.

Primal objective	Dual objective
maximise $\mathbf{c}^T\mathbf{x}$	minimise $\mathbf{b}^T\mathbf{y}$
Primal variable	**Dual constraint**
$x_k \geq 0$	$(\mathbf{A}^T)_k\mathbf{y} \geq c_k$
$x_k \leq 0$	$(\mathbf{A}^T)_k\mathbf{y} \leq c_k$
x_k unrestricted	$(\mathbf{A}^T)_k\mathbf{y} = c_k$
Primal constraint	**Dual variable**
$\mathbf{A}_k\mathbf{x} \geq b_k$	$y_k \leq 0$
$\mathbf{A}_k\mathbf{x} \leq b_k$	$y_k \geq 0$
$\mathbf{A}_k\mathbf{x} = b_k$	y_k unrestricted

2.5 Graph Theory

Here we give some basic definitions from graph theory.

Definition 2.12. An *undirected graph* is a tuple $G = (V, E)$ of sets V, called the *vertices*, and $E \subseteq \{\{v, u\} : v, u \in V\}$, called the *undirected edges*.

Definition 2.13. A *(directed) graph* is a tuple $G = (V, E)$ of *vertices* V and *(directed) edges* $E \subseteq V \times V = \{(v, u) : v, u \in V\}$.

An example of a directed graph can be found in Figure 2.2. The vertices are represented by circles, and the edges are represented by arrows between the

two vertices they connect. The graph in the figure has the edges

$$E = \{(1,2),(1,4),(1,5),(2,3),(2,4),(3,6),(4,3),(4,5),(4,6),(5,6)\}.$$

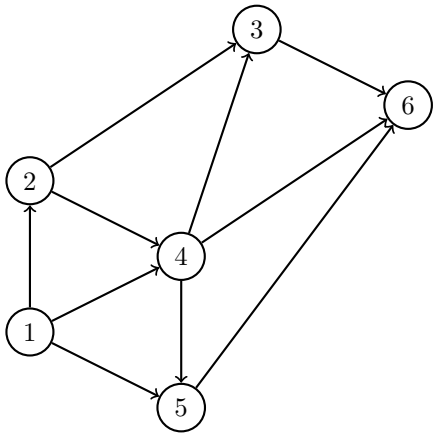


Figure 2.2: An example of a directed graph with $V = \{1,\ldots,6\}$ and $|E| = 10$.

Definition 2.14. Let $G = (V, E)$ be a directed graph and $v \in V$ a vertex. The set of *outgoing edges* of v is defined as

$$\delta_G^+(v) = \{(v', u) \in E : v' = v\}.$$

We define the *outgoing neighborhood* of v as

$$\Delta_G^+(v) = \{u \in V : (v, u) \in E\},$$

The vertices $u \in \Delta_G^+(v)$ above are called *direct successors* of v. Likewise, the set of *incoming edges* of v is

$$\delta_G^-(v) = \{(u, v') \in E : v' = v\}.$$

and we define the *incoming neighborhood* as

$$\Delta_G^-(v) = \{u \in V : (u, v) \in E\}.$$

The vertices $u \in \Delta_G^-(v)$ above are called *direct predecessors* of v.

We define the *neighbourhood* of v as

$$N_G(v) := \Delta^+(v) \cup \Delta^-(v).$$

Definition 2.15. Let $G = (V, E)$ be an undirected graph. The *neighbourhood* of a vertex v is defined as

$$N_G(v) := \{u \in V : \{v, u\} \in E\}.$$

In the example in Figure 2.2, we have for instance

$$\begin{aligned}
\delta_G^+(4) &= \{(4,3),(4,5),(4,6)\},\\
\delta_G^-(4) &= \{(1,4),(2,4)\},\\
\Delta_G^+(4) &= \{3,5,6\},\\
\Delta_G^-(4) &= \{1,2\},\\
N_G(4) &= \{1,2,3,5,6\}.
\end{aligned}$$

Definition 2.16. For a directed graph $G = (V, E)$ and a set of vertices $V' \subseteq V$, we define

$$\delta_G^+(V') := \{(v, u) \in E : v \in V', u \in V \setminus V'\}$$

and

$$\delta_G^-(V') := \{(v, u) \in E : v \in V \setminus V', u \in V'\}.$$

We also define

$$\Delta_G^+(V') := \{u \in V : (v, u) \in \delta^+(V') \text{ for some } v \in V'\}$$

and

$$\Delta_G^-(V') := \{v \in V : (v, u) \in \delta^-(V') \text{ for some } u \in V'\}.$$

Note that, for example, $\delta_G^+(\{v\}) = \delta_G^+(v)$ for any $v \in V$.

Definition 2.17. Let $G = (V, E)$ be a graph and $V' \subseteq V$. If G is undirected, then we let

$$\delta_G(V') = \{\{v, u\} : \{v, u\} \in E \text{ for some } v \in V', u \in V\}$$

If G is directed, then

$$\delta_G(V') = \delta_G^+(V') \cup \delta_G^-(V').$$

The subscript G in $\delta_G^+(\cdot)$, $\delta_G^-(\cdot)$, etc., will be omitted if the graph in question is clear from context.

Definition 2.18. Let $G = (V, E)$ be a directed graph. A *path* in G is a finite sequence of vertices $v_1, \ldots, v_k \subseteq V$ such that

$$(v_j, v_{j+1}) \in E \quad \text{for all } j \in [k-1].$$

In this case, v_k is said to be *reachable* from, or a *successor* of, v_1, while v_1 is said to be a *predecessor* of v_k. We write $v_1 \rightsquigarrow_G v_k$ to denote this relation, omitting the graph G if it is clear from context.

If $v_1 = v_k$ we say that the path is a *cycle*. If instead all vertices $v_1, \ldots, v_k$ are distinct, we say that it is a *simple path*.

Definition 2.19. A set of vertices $V' \subseteq V$ is said to be reachable from $v \in V$, written $v \rightsquigarrow V'$, if

$$\exists v' \in V' \text{ such that } v \rightsquigarrow v'.$$

Likewise, v is reachable from V', $V' \rightsquigarrow v$, if

$$\exists v' \in V' \text{ such that } v' \rightsquigarrow v.$$

Definition 2.20. A *directed acylic graph (DAG)* is a directed graph that has no cycles.

In the graph in Figure 2.2, we have the (simple) path $(1, 2, 4, 3, 6)$, among others. This graph contains no cycles, so it is a DAG.

Definition 2.21. Let $G = (V, E)$ be a directed graph. A graph $\hat{G} = (\hat{V}, \hat{E})$ is a *subgraph* of G if

$$\hat{V} \subseteq V \quad \text{and} \quad \hat{E} \subseteq E.$$

In this case, G is said to be a *supergraph* of $\hat{G}$.

Another class of graphs are bipartite graphs:

Definition 2.22. Let $G = (V, E)$ be a directed (undirected) graph. Assume that there exist sets V_1 and V_2, which form a partition of V, such that for all edges $(v, u) \in E$ (edges $\{v, u\} \in E$), either $v \in V_1$ and $u \in V_2$, or $v \in V_2$ and $u \in V_1$. Then we say that G is a *bipartite graph*.

See Figure 2.3 for an example of a directed bipartite graph.

2.6 Network Flow Problems

A *network flow problem* is a problem where we want to find a *flow* $\mathbf{f} \in \mathbb{R}^E$ for a given graph $G = (V, E)$, according to some specified objective and constraints. A flow is simply a vector over the edges of a graph, that satisfies the conservation of flow, which will be explained in Definition 2.23. It can be thought of as representing the quantity of something (e.g., water or electrical current) moving along the edges of the graph (which represent e.g., pipes or wires). One network flow problem that is of interest to us is the following:

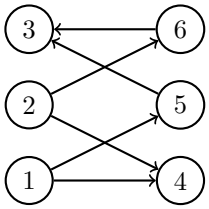


Figure 2.3: An example of a (directed) bipartite graph with $V = \{1, \ldots, 6\}$, $|E| = 6$, $V_1 = \{1, 2, 3\}$ and $V_2 = \{4, 5, 6\}$.

Definition 2.23 (Maximum flow problem)**.** Given a DAG $G = (V, E)$, a *source* $s \in V$, a *sink* $t \in V$ and *capacities* $c_e \in [0, \infty]$ for $e \in E$, we wish to find a *valid flow* $\mathbf{f} \in \mathbb{R}^E_{\geq 0}$, which is a vector satisfying the conservation of flow

$$f(\delta^-(v)) = f(\delta^+(v)) \quad \forall v \in V \setminus \{s, t\},$$

or equivalently

$$\sum_{u \in \Delta^+(v)} f_{v,u} - \sum_{u \in \Delta^-(v)} f_{u,v} = 0 \quad \forall v \in V \setminus \{s, t\}, \tag{2.15}$$

as well as the capacity constraints

$$f_{v,u} \leq c_{v,u} \quad \forall (v, u) \in E. \tag{2.16}$$

We define the *value* of a valid flow, which we wish to maximise, as

$$\text{val}(\mathbf{f}) := f(\delta^+(s)) = f(\delta^-(t)).$$

An intuition for this problem is that we have some roads (edges) between different cities (nodes), and want to maximise the amount of product (the flow) sent from one location (s) to another (t).

For the problems we are interested in, we can also assume that

$$\delta^-(s) = \delta^+(t) = \varnothing, \tag{2.17}$$

that is, that there are neither incoming edges to the source nor outgoing edges from the sink. The dual of this problem (given our assumption) has a variable $d_{v,u}$ for every $(v, u) \in E$ and a variable z_v for every $v \in V \setminus \{s, t\}$. The constraints

of the dual are

$$d_{v,u} - z_v + z_u \geq 0 \qquad \forall (v,u) \in E : v \neq s, u \neq t, \tag{2.18}$$

$$d_{s,u} + z_u \geq 1 \qquad \forall u \in \Delta^+(s) : u \neq t, \tag{2.19}$$

$$d_{v,t} - z_v \geq 0 \qquad \forall v \in \Delta^-(t) : v \neq s, \tag{2.20}$$

$$d_{s,t} \geq 1 \qquad \text{if } (s,t) \in E, \tag{2.21}$$

$$d_{v,u} \geq 0 \qquad \forall (v,u) \in E. \tag{2.22}$$

The objective is to minimise

$$\sum_{e \in E} c_e d_e. \tag{2.23}$$

If for any $e \in E$, we have $c_e = \infty$, then we say that the whole sum is ∞, unless $d_e = 0$.

This LP can be interpreted by first noting that there always exists an optimal integral solution, i.e., we can find values $d_e \in \{0,1\}$ for $e \in E$ and $z_v \in \{0,1\}$ for $v \in V \setminus \{s,t\}$, such that the objective value of the LP (2.18)–(2.23) is minimised. This fact is proved in [11, pp. 26–30].

We can assume we have such a solution and introduce the sets

$$V_s := \{v \in V \setminus \{s,t\} : z_v = 1\}$$

and

$$V_t := V \setminus (V_s \cup \{s,t\}) = \{v \in V \setminus \{s,t\} : z_v = 0\}.$$

as well as the set of edges

$$E_C := \{e \in E : d_e = 1\}.$$

We note that the constraint (2.18) ensures that if $v \in V_s$ and $u \in V_t$, then $d_{v,u} \geq z_v - z_u = 1 - 0$, so $(v,u) \in E_C$. The constraint (2.19) ensures that if $u \in V_t \cap \Delta^+(s)$, then $(s,u) \in E_C$, and likewise (2.20) ensures that if $v \in V_s \cap \Delta^-(t)$, then $(v,t) \in E_C$. The constraint (2.21) guarantees that if $(s,t) \in E$, then $(s,t) \in E_C$.

The objective (2.23) is to minimise $\mathrm{cap}(E_C)$, the sum of the capacity over the edges in E_C. Because this is minimised, we do not need any constraints to guarantee that an edge (v,u) is not in E_C if, e.g., both $v \in V_s$ and $u \in V_s$. We note that this is not true for the edge case of $c_{v,u} = 0$.

This dual can thus be interpreted as the problem:

Definition 2.24 (Minimum (s,t)-cut problem)**.** We are given a DAG $G = (V,E)$, a source $s \in V$, a sink $t \in V$, and capacities $c_e \in [0,\infty]$ for $e \in E$. A *cut* $C = (V_s, V_t)$ is a *partition* of V, i.e.,

$$V = V_s \cup V_t \quad \text{and} \quad V_s \cap V_t = \varnothing \tag{2.24}$$

such that

$$s \in V_s \quad \text{and} \quad t \in V_t. \tag{2.25}$$

The *cut set* of a cut is defined as the set of edges from V_s to V_t (not including edges from V_t to V_s),

$$E_C := \delta^+(V_s) = \delta^-(V_t) = (V_s \times V_t) \cap E$$

and the *capacity* of a cut, which we wish to minimise, is defined as

$$\text{cap}(C) := \sum_{e \in E_C} c_e.$$

Theorem 2.25 (Max-flow min-cut theorem)**.** *Given $G = (V, E)$, s, t and $\mathbf{c}$ as in Definition 2.23 and Definition 2.24, we have*

$$\max_{\mathbf{f} \text{ s.t. } (2.15),(2.16)} \text{val}(\mathbf{f}) = \min_{C \text{ s.t. } (2.24),(2.25)} \text{cap}(C).$$

Proof. As the min-cut problem is the dual of the max-flow problem, this follows directly from strong duality, Theorem 2.11. □

Note that here we have only shown the max-flow min-cut theorem for the special case where we assume (2.17), but it is true for the general problems described in Definition 2.23 and Definition 2.24. See [11, pp. 11–12, Theorem 5.1] for a proof.

2.7 The Multiplicative Weight Update Method

A method that is useful for solving a variety of problems is the multiplicative weight update method [2]. This method works for situations where we have a number of *experts* that can give advice about a situation, but we are not sure whose advice to follow.

Assume that we have n experts each giving some advice for a number of days T, which results in some reward or penalty $\mathbf{m}^{(t)} \in [-1, 1]^n$ for $t \in [T]$, called the *outcome*. The penalty $m_i^{(t)} = 1$ is the worst outcome and the reward $m_i^{(t)} = -1$ is the best. These are not known to the algorithm until after the decision for day $t \in [T]$ has been made.

The idea of the method is to assign a weight $w_i^{(t)} \in \mathbb{R}_{\geq 0}$ to each expert $i \in [n]$ at each time $t \in T$, starting with

$$\mathbf{w}^{(1)} = (1, \ldots, 1) \in \mathbb{R}^n_{\geq 0}$$

and updating the weights after each step according to

$$w_i^{(t+1)} = (1 - \eta m_i^{(t)}) w_i^{(t)},$$

where $\eta \in (0, 1/2)$ is a parameter chosen before the start of the algorithm. This parameter affects how much the weight is changed based on the outcome, and the performance guarantee (2.26) (which will be explained in Theorem 2.26) depends on the value chosen for η.

Define the *potential function*

$$\Phi^{(t)} = \sum_{i=1}^{n} w_i^{(t)}.$$

At each day, we choose the expert whose advice to follow by picking randomly, with probabilities according to the normalised weights

$$p_i^{(t)} = \frac{w_i^{(t)}}{\Phi^{(t)}}.$$

This gives us the expected cost $\mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)}$ at day $t \in T$.

The idea of this method is that the weight of an expert that gives bad advice (i.e., advice resulting in a high value for the outcome $m_i^{(t)}$) quickly decreases to near zero, while the cost incurred by mistakes (i.e., the sum $\sum_{t'=1}^{t} m_i^{(t')}$) grows slowly. More exactly, the weight decreases exponentially while the cost grows linearly.

Using this method, we obtain an upper bound on the total penalty (or reward) [2, p. 126]:

Theorem 2.26. *Running the MWU algorithm for n experts and T days, with penalties according to $\mathbf{m}^{(t)} \in [-1, 1]^n$ for each day $t \in [T]$, it holds that*

$$\sum_{t=1}^{T} \mathbf{p}^{(t)} \cdot \mathbf{m}^{(t)} \leq \sum_{t=1}^{T} m_i^{(t)} + \eta \sum_{t=1}^{T} \left| m_i^{(t)} \right| + \frac{\log n}{\eta} \tag{2.26}$$

for all $i \in [n]$.

In particular, (2.26) *holds for the expert $i^* \in [n]$ which had the lowest total number of mistakes.*

The following proof is from [2, pp. 127–128].

Proof. Rewrite the potential function

$$\begin{aligned}
\Phi^{(t+1)} &= \sum_{i\in[n]} w_i^{(t+1)} \\
&= \sum_{i\in[n]} w_i^{(t)}(1-\eta m_i^{(t)}) \\
&= \sum_{i\in[n]} w_i^{(t)} - \eta \sum_{i\in[n]} \overbrace{w_i^{(t)}}^{=\Phi^{(t)}p_i^{(t)}} m_i^{(t)} \\
&= \Phi^{(t)} - \eta\Phi^{(t)} \sum_{i\in[n]} m_i^{(t)} p_i^{(t)} \\
&= \Phi^{(t)}(1-\eta \mathbf{m}^{(t)}\cdot\mathbf{p}^{(t)}) \\
&\le \Phi^{(t)} \exp(-\eta \mathbf{m}^{(t)}\cdot\mathbf{p}^{(t)}).
\end{aligned}$$

The inequality at the end uses the fact that

$$e^{-x} \ge 1-x \quad \text{for } x\in\mathbb{R}, \tag{2.27}$$

which can be proven by noting that e^{-x} is convex, as $\frac{\mathrm{d}^2}{\mathrm{d}^2 x} e^{-x} = e^{-x} > 0$, and has the tangent $e^{-0} + (-e^{-0}x) = 1-x$ at 0. A convex function lies above any of its tangents, which gives (2.27). We also use the fact that $\Phi^{(t)} \ge 0$ in the last inequality, which can be proven by noting that $w_i^{(1)} = 1$ is positive, and for each iteration, we multiply it by

$$1 - \underbrace{\eta}_{<1/2}\underbrace{m_i^{(t)}}_{<1} \ge 0.$$

Now induction gives

$$\begin{aligned}
\Phi^{(t+1)} &\le \Phi^{(t)} \exp(-\eta \mathbf{m}^{(t)}\cdot\mathbf{p}^{(t)}) \\
&\le \Phi^{(t-1)} \exp(-\eta \mathbf{m}^{(t)}\cdot\mathbf{p}^{(t)} - \eta \mathbf{m}^{(t-1)}\cdot\mathbf{p}^{(t-1)}) \\
&\le \underbrace{\Phi^{(1)}}_{=\sum_i w_i^{(1)}} \exp\left(-\eta \sum_{t'=1}^{t} \mathbf{m}^{(t')}\cdot\mathbf{p}^{(t')}\right) \\
&= n \exp\left(-\eta \sum_{t'=1}^{t} \mathbf{m}^{(t')}\cdot\mathbf{p}^{(t')}\right).
\end{aligned}$$

We also have that

$$\Phi^{(t+1)} = \sum_{i\in[n]} w_i^{(t+1)} \geq w_i^{(t+1)} \quad \text{for any } i \in [n].$$

Bernoulli's inequality states that

$$1 - \eta x \geq (1-\eta)^x \quad \text{for } x \in [0,1], \eta \leq 1 \tag{2.28}$$

and replacing x by $-x$ and η by $-\eta$ gives us

$$1 - \eta x \geq (1+\eta)^{-x} \quad \text{for } x \in [-1,0], \eta \geq -1.$$

These inequalities together with induction on the definition of the weights gives

$$\begin{aligned} w_i^{(t+1)} &= \prod_{t'=1}^{t} \left(1 - \eta m_i^{(t')}\right) \\ &\geq \prod_{\geq 0} (1-\eta)^{m_i^{(t')}} \cdot \prod_{<0} (1+\eta)^{-m_i^{(t')}} \\ &= (1-\eta)^{\sum_{\geq 0} m_i^{(t')}} \cdot (1+\eta)^{-\sum_{<0} m_i^{(t')}}, \end{aligned} \tag{2.29}$$

where the subscript ≥ 0 on the product and sum means taking the product or sum, respectively, over $t' = 1, \ldots, t$ where $m_i^{(t')} \geq 0$. The subscript < 0 has the analogous meaning.

Combining these preceding equations gives

$$n \exp\left(-\eta \sum_{t'=1}^{t} \mathbf{m}^{(t')} \cdot \mathbf{p}^{(t')}\right) \geq \Phi^{(t+1)} \geq w_i^{(t+1)} \geq \left(1-\eta\right)^{\sum_{\geq 0} w_i^{(t')}} \left(1+\eta\right)^{-\sum_{<0} w_i^{(t')}}$$

and taking the logarithm of both sides, as well as using (2.29), results in

$$\log n - \eta \sum_{t'=1}^{t} \mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)} \geq \log(1-\eta) \sum_{\geq 0} m_i^{(t')} - \log(1+\eta) \sum_{<0} m_i^{(t')}$$

which can be rearranged to

$$\begin{aligned}
\sum_{t'=1}^{t} \mathbf{m}^{(t')} \cdot \mathbf{p}^{(t')} &\leq \frac{1}{\eta} \log \frac{1}{1-\eta} \sum_{\geq 0} m_i^{(t')} + \frac{1}{\eta} \log(1+\eta) \sum_{<0} m_i^{(t')} + \frac{\log n}{\eta} \\
&\leq \frac{\eta+\eta^2}{\eta} \sum_{\geq 0} m_i^{(t')} + \frac{\eta-\eta^2}{\eta} \sum_{<0} m_i^{(t')} + \frac{\log n}{\eta} \\
&= \sum_{\geq 0} m_i^{(t')} + \sum_{<0} m_i^{(t')} + \eta \sum_{\geq 0} m_i^{(t')} - \eta \sum_{<0} m_i^{(t')} + \frac{\log n}{\eta} \\
&= \sum_{t'=1}^{t} m_i^{(t')} + \eta \sum_{t'=1}^{t} \left| m_i^{(t')} \right| + \frac{\log n}{\eta},
\end{aligned}$$

where the second inequality in the above equation uses that

$$\log(1+\eta) \geq \eta - \eta^2 \quad \text{for } \eta \in (0,1)$$

and

$$\log \frac{1}{1-\eta} \leq \eta + \eta^2 \quad \text{for } \eta \in (0, 1/2). \qquad \square$$

2.7.1 Solving an LP using MWU

A variation of the basic MWU algorithm can be used to solve certain optimisation problems efficiently, given that we accept some approximations. The algorithm is described in [4] and [6]. We first define what a convex set is:

Definition 2.27. A set (of vectors) C is said to be *convex* if for all $x \in C$ and $y \in C$, it holds that

$$(1-t)x + ty \in C \quad \forall t \in [0,1].$$

The problem which the method can solve is the convex optimisation problem

$$\begin{aligned}
&\text{maximise} && f(\mathbf{x}), && && (2.30)\\
&\text{subject to} && g_i(\mathbf{x}) \leq 1 && \forall i \in [m], && (2.31)\\
& && \mathbf{x} \in \mathcal{Q}, && && (2.32)
\end{aligned}$$

where $g_i \colon \mathbb{R}^n \to \mathbb{R}$ are convex functions and $f \colon \mathbb{R}^n \to \mathbb{R}$ is a continuous function. The convex set $\mathcal{Q}$ is assumed to be "easy" or "simple", i.e., we can find an optimum for some given objective inside the set quickly. The constraints $g_i(\mathbf{x}) \leq 1$ are so-called "complicated" constraints.

The following theorem states that we have a "continous-time" algorithm finding an approximately feasible solution to the problem [4, Theorem 3, p. 204]. That is, a solution to the problem where the constraint (2.31) has been relaxed to (2.33).

Theorem 2.28. *The procedure given in Algorithm 1 solves the problem* (2.30)–(2.32) *approximately, by returning an* $\mathbf{x}_{out}$ *such that* $\mathbf{x}_{out} \in \mathcal{Q}$ *and*

$$g_i(\mathbf{x}_{out}) \leq 1 + \frac{\log m}{\eta} \quad \forall i \in [m], \tag{2.33}$$

where $\eta > 0$ *is a parameter analogous to the* η *described in Section 2.7. Note however that we no longer have an upper bound on* η*.*

In particular, if $\eta \geq \frac{\log m}{\varepsilon}$ *for some* $\varepsilon \in [0, 1/2)$*, then* $g_i(\mathbf{x}_{out}) \leq 1 + \varepsilon$ *for* $i \in [m]$*.*

Algorithm 1 MWU template from [4, p. 204]

procedure MWU($f, g_1, \ldots, g_m, \mathcal{Q}, \eta$)
  $\mathbf{w}^{(0)} \leftarrow$ all-1 vector of size m
  for $t \in [0, 1]$ **do**
    Find $\mathbf{v}^{(t)}$ that is good for $f(\mathbf{v}^{(t)})$, and such that
      $\mathbf{v}^{(t)} \in \mathcal{Q}$ and $\sum_{i=1}^{m} w_i^{(t)} g_i(\mathbf{v}^{(t)}) \leq \sum_{i=1}^{m} w_i^{(t)}$
    for $i \in [m]$ **do**
      $\frac{dw_i^{(t)}}{dt} \leftarrow \eta w_i^{(t)} g_i(\mathbf{v}^{(t)})$
    end for
  end for
  $\mathbf{x}_{\text{out}} \leftarrow \int_0^1 \mathbf{v}^{(t)} dt$
  return $\mathbf{x}_{\text{out}}$
end procedure

In Algorithm 1, note that the **for**-loop iterates over the set $[0, 1]$, which is uncountable. Clearly it is not possible to implement this algorithm in practice. However, we can replace the interval by some partition of it, for example $\{k/N : k = 0, \ldots, N\}$ for some positive integer N, and consider the algorithm as the limit as the number of points in the partition increase, in this case $N \to \infty$. This is similar to the idea of the Riemann integral.

Of course, this limit is also not possible to compute in finite time, but a discretisation of the algorithm can be found, based upon that idea. This is found in Algorithm 2 [4, Algorithm 2, p. 206]. Note that it is only valid for the case when $g_i(\mathbf{x}) \geq 0$ for $i \in [m]$ and $\mathbf{x} \in \mathcal{Q}$. In [4, Theorem 11, p. 207] it is

further proved that the discrete version of the algorithm, Algorithm 2, returns an $\mathbf{x}_{\text{out}}$ that satisfies $\mathbf{x}_{\text{out}} \in \mathcal{Q}$ and $g_i(\mathbf{x}_{\text{out}}) \leq 1+2\varepsilon$ where $\eta := \log m/\varepsilon$ for some $\varepsilon \in (0, 1/2)$. An important special case of (2.30)–(2.32) is when $f(\mathbf{x}) = \mathbf{c}^T\mathbf{x}$

Algorithm 2 Discretisation of Algorithm 1.

procedure MWU_DISCRETE($f, g_1, \ldots, g_m, \mathcal{Q}, \varepsilon$)
  $\mathbf{w}^{(0)} \leftarrow$ all-1 vector of size m
  $\mathbf{x} \leftarrow$ all-0 vector of size n
  $t \leftarrow 0$
  while $t < 1$ **do**
    Find $\mathbf{v}^{(t)}$ that is good for $f(\mathbf{v}^{(t)})$, and such that
      $\mathbf{v}^{(t)} \in \mathcal{Q}$ and $\sum_{i=1}^m w_i^{(t)} g_i(\mathbf{v}^{(t)}) \leq \sum_{i=1}^m w_i^{(t)}$
    $\delta \leftarrow \min\left\{\frac{\varepsilon}{\eta} \cdot \frac{1}{\max_{i\in[m]} g_i(\mathbf{v}^{(t)})}, 1-t\right\}$
    $\mathbf{x} \leftarrow \mathbf{x} + \delta\mathbf{v}^{(t)}$
    for $i \in [m]$ **do**
      $w_i(t+\delta) \leftarrow w_i^{(t)} \cdot \exp(\eta\delta \cdot g_i(\mathbf{v}^{(t)}))$
    end for
    $t \leftarrow t + \delta$
  end while
  $\mathbf{x}_{\text{out}} \leftarrow \mathbf{x}$
  return $\mathbf{x}_{\text{out}}$
end procedure

and $g_i(\mathbf{x}) = \mathbf{A}_i\mathbf{x}$ for $i \in [m]$, for a matrix $\mathbf{A}$ where all elements are nonnegative. The special case problem is thus

$$\text{maximise } \mathbf{c}^T\mathbf{x} \quad \text{subject to} \quad \mathbf{x} \in \mathcal{Q} \text{ and } \mathbf{A}\mathbf{x} \leq \mathbf{1}. \tag{2.34}$$

In this case Algorithm 2 simplifies to Algorithm 3 [28, Algorithm 1, p. 8]. This algorithm is the basis of the method which we use to solve the LP. However, we also let the $\mathbf{v}(t)$ which solves (2.34) be an approximate solution. This approximation will be explained later.

2.8 Dynamic Programming

Dynamic programming is a method in algorithm design that takes advantage of so-called *optimal substructure* of a problem. This is defined as the property that an optimal solution to a problem can be found, if an optimal solution to each of its subproblems is known. Another method in algorithm design that takes

Algorithm 3 Solving a linear packing problem, a special case of Algorithm 2.

procedure MWU_LINEAR_PACKING($\mathbf{c}, \mathbf{A}, \mathcal{Q}, \eta, \varepsilon$)
  $\mathbf{w}^{(0)} \leftarrow$ all-1 vector of size m
  $\mathbf{x} \leftarrow$ all-0 vector of size n
  $t \leftarrow 0$
  $\eta \leftarrow \frac{\log m}{\varepsilon}$
  while $t < 1$ **do**
    $\mathbf{b} \leftarrow \frac{\mathbf{w}^{(t)}}{|\mathbf{w}^{(t)}|}\mathbf{A}$
    Find $\mathbf{v}^{(t)}$ that is good for $\mathbf{c}^T\mathbf{v}^{(t)}$ such that
      $\mathbf{v}^{(t)} \in \mathcal{Q}$ and $\sum_{i=1}^m w_i^{(t)}\mathbf{A}_i\mathbf{v}^{(t)} \leq \sum_{i=1}^m w_i^{(t)}$
    $\delta \leftarrow \min\left\{\frac{\varepsilon}{\eta} \cdot \frac{1}{\max_{i\in[m]} \mathbf{A}_i\mathbf{v}^{(t)}}, 1-t\right\}$
    $\mathbf{x} \leftarrow \mathbf{x} + \delta\mathbf{v}^{(t)}$
    for $i \in [m]$ **do**
      $w_i^{(t+\delta)} \leftarrow w_i^{(t)} \cdot \exp(\eta\delta \cdot \mathbf{A}_i\mathbf{v}^{(t)})$
    end for
    $t \leftarrow t + \delta$
  end while
  $\mathbf{x}_{\text{out}} \leftarrow \mathbf{x}$
  return $\mathbf{x}_{\text{out}}$
end procedure

advantage of this property is *divide-and-conquer*. The difference between these two is that dynamic programming also makes use of the property of *overlapping subproblems*, i.e., the fact that a naive implementation would end up solving the same subproblem multiple times.

For example, the following recursive definition of the Fibonnacci numbers

$$f(1) = 1, f(2) = 1, f(n) = f(n-2) + f(n-1) \quad \forall n \geq 3,$$

requires evaluating f twice for each recursive step, so a straightforward implementation would require $O(2^n)$ evaluations to compute $f(n)$. However, we can instead start by computing $f(3) = f(1) + f(2)$ and store the result, then compute $f(4) = f(2) + f(3)$, and so on until we reach $f(n)$. This gives us the results $f(n)$ after n steps, each step consisting of only an addition (as well as reading $f(k-2), f(k-1)$ from memory and writing $f(k)$), so this implementation runs in $O(n)$ time.

Chapter 3

The Algorithm and its Implementation

In this chapter, we describe the algorithm. In essence, it consists of constructing a linear programming relaxation of (a restriction of) the original problem and solving this LP relaxation using an MWU algorithm, before finally rounding this to a solution of the original problem using a rounding algorithm. This is an application of the so-called *relax-and-round* method. See Figure 3.1 for an overview of how these parts interact.

We describe the details of this LP relaxation, how it is transformed into a packing LP that can be solved by the MWU algorithm, and our implementation of the MWU and rounding algorithms. The MWU algorithm uses a sub-algorithm for solving a certain network flow problem, which arises as the dual of a subproblem of the packing LP. This sub-algorithm will be explained in Chapter 4.

We start by explaining the high-level view of the algorithm. The details will be explained in the following sections. The LP relaxation is based on an integer programming (IP) model of (a restriction of) the problem, where we model the completion time of each job with a number of binary variables, one for each time point. This is called a *time indexed* model, and there exist a few variants in how the relation between the binary variables and the completion times is defined. We use the relation that

$$x_{j,t} = \begin{cases} 1, & \text{if } C_j \leq t, \\ 0, & \text{otherwise.} \end{cases}$$

See Figure 3.2 for a sketch. We see that this representation of C_j cannot rep-

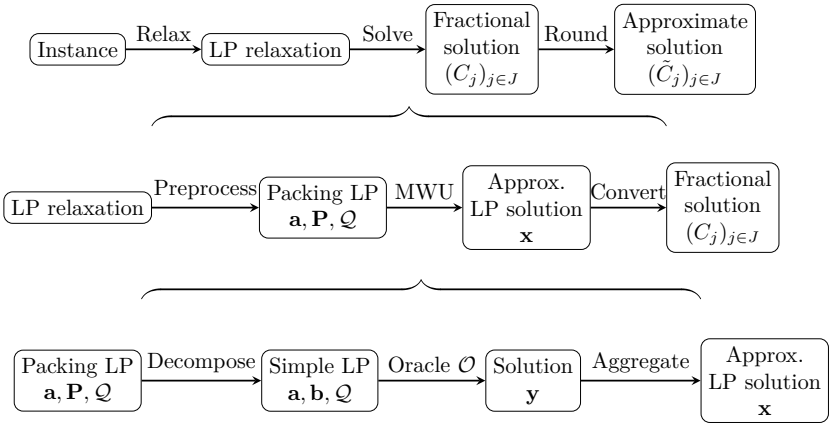


Figure 3.1: Detailed sketch of the "outer layer" of the algorithm, which is described in this chapter.

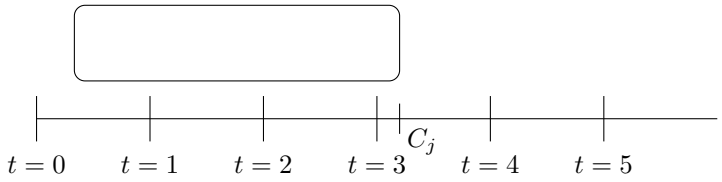


Figure 3.2: An example of how binary variables $x_{j,t}$ for $t = 0, 1, 2, \ldots$ are used to model the completion time C_j for some job j. Here we have $C_j = 3.2$, so $x_{j,0} = x_{j,1} = x_{j,2} = x_{j,3} = 0$ and $x_{j,4} = x_{j,5} = 1$.

resent every possible value for the completion time, but only a discrete set of values, for example $\{0, \ldots, 5\}$ in Figure 3.2. In Section 3.1.1, we explain how we chose this set of values for our model. Of note is that the set of values cannot be too large, because this would result in a large number of variables, which in turn affects the time it takes to find a solution to the LP. Therefore, we constrain our allowed solutions to those where the completion times are zero or a power of $(1 + \varepsilon)$.

This IP model can then be relaxed by allowing the binary variables to take any real value (in $[0, 1]$), which makes it an LP relaxation. Since we can solve the LP relaxation quickly with the method described in Section 2.7.1 (i.e., MWU), we can solve the scheduling problem by solving a number of simpler LPs. The LPs which we need to solve are simpler in the sense that they have only a single

constraint, where the original LP has many rows. This is because the MWU algorithm in each iteration handles a single linear inequality, which is computed from the original constraint matrix using the weights. We can transform one such simple LP into a set of network flow problems, which we will solve using an algorithm that we will explain in Chapter 4.

The rounding algorithm for a single machine, which is described in Section 3.3, simply works by sorting the jobs based on their *fractional completion times*, i.e., the completion times produced by solving the LP relaxation. By Theorem 3.2, which states that the fractional completion times obey the precedence constraints, we see that we only have to remove eventual overlaps. This can be done by simply setting the completion time of a job j to the total processing time of all jobs which come before j, when sorted by fractional completion times.

3.1 Integer Programming Formulation

The starting point of the algorithm is an IP formulation of the original problem in Section 1.2. As we saw in Section 1.2, if the processing times are integral, then there exists an optimal solution with integer-valued completion times. A natural starting point for a model will thus use a binary variable $x_{j,t}$ for every $j \in J, t = 0, 1, \ldots, T$, where $T := p(J)$, which indicate that j completes before or at time t if $x_{j,t} = 1$, and that j completes after t if not. This can be expressed using the completion time C_j as

$$\begin{aligned} C_j &\leq t, \quad \text{if } x_{j,t} = 1, \\ C_j &> t, \quad \text{otherwise.} \end{aligned}$$

We can then compute the completion times $(C_j)_{j \in J}$ from $(x_{j,t})_{j \in J, t \in [0,T]}$ by setting

$$C_j := \sum_{t=1}^{T} t(x_{j,t} - x_{j,t-1}) = p(J) - \sum_{t=0}^{T-1} x_{j,t},$$

so the objective of our model is to minimise

$$\begin{aligned} \sum_{j \in J} w_j C_j &= \sum_{j \in J} w_j \Big(p(J) - \sum_{t=0}^{T-1} x_{j,t} \Big) \\ &= w(J)p(J) - \sum_{j \in J} w_j \sum_{t=0}^{T-1} x_{j,t}. \end{aligned} \tag{3.1}$$

By definition, if $x_{j,t} = 1$ (which represents the job completing before or at time t), then it must be the case that $x_{j,t+1} = 1$ (which represents the job completing before or at time $t + 1$). This gives us the constraints

$$x_{j,t} \leq x_{j,t+1} \quad \forall j \in J, t \in [0, T-1]. \tag{3.2}$$

The constraints

$$x_{j,t} \geq x_{j',t} \quad \forall j \prec j', t \in [0, T] \tag{3.3}$$

ensures that for two jobs $j \prec j'$, if j' completes before or at time t, then j also completes before or at time t. We also have the constraints

$$\sum_{j \in J} p_j x_{j,t} \leq t \quad \forall t \in [0, T] \tag{3.4}$$

to ensure that the total time spent on jobs before time t does not exceed t time units.

Now we need to introduce a new parameter. Let q_j be the maximum of the total length of jobs in any precedence chain ending at $j \in J$, i.e.,

$$q_j := \max_{j_1 \prec j_2 \prec \cdots \prec j_K = j} \sum_{k=1}^{K} p_{j_k}.$$

This is computed as described in Section 3.1.4. Like all code listings in this thesis, it is written in the Julia language. To ensure that the job j does not complete before time q_j, we have the constraints

$$x_{j,t} = 0 \quad \forall j \in J, t \in [0, T] : t < q_j. \tag{3.5}$$

For example, a job j with processing time $p_j = 3$ for which there are no precedence constraints on the form $j' \prec j$, can complete at the earliest at time 3. Likewise, a job j with $p_j = 3$ and for which the only precedence constraint is $j' \prec j$, where $p_{j'} = 2$, can complete at the earliest at time 5. The constraints

$$x_{j,T} = 1 \quad \forall j \in J \tag{3.6}$$

ensures that all jobs complete before or at time T.

3.1.1 Discretisation of Completion Times

In the model we just described, we have discretised the completion times by requiring that they be integer valued. However, the actual model we solve uses a slightly different discretisation. We define the list of *discrete completion times*

by setting $\tau_1 = 0$ and $\tau_d = (1+\varepsilon)^{d-1}$ for every integer $d \geq 2$. Here we use a parameter $\varepsilon > 0$ that affects the quality of the solution, as well as certain aspects affecting the run time, such as the number of variables. We note that this is a method often used in approximation algorithms for scheduling problems, see for example [16] and [27]. Let D be the smallest integer such that

$$\tau_{D+1} \geq p(J) \tag{3.7}$$

and set $\eta_d := \tau_{d+1} - \tau_d$ for each $d \in [D]$. Now η_d is the length of the time interval between τ_d and τ_{d+1}, which is useful, for example, in computing the completion times in (3.10). Note that we use indexing from one, i.e., $(\tau_d)_{d\in[1,D+1]}$ as opposed to indexing from zero, i.e., $(\tau_d)_{d\in[0,D]}$ as in [28, p. 8].

It is necessary to use this discretisation of completion times, because the IP model is *time indexed*, meaning that it has one (binary) decision variable for each job and time point. Allowing any integer completion time results in

$$n \cdot T = n \cdot p(J) \tag{3.8}$$

variables. This would cause the algorithm to have a run time exceeding (1.6). Using the discretisation explained above results in nD variables. Solving for D in (3.7) gives

$$\begin{aligned} p(J) \leq \tau_{D+1} = (1+\varepsilon)^D &\iff \log p(J) \leq D \log(1+\varepsilon) \\ &\iff D = \left\lceil \frac{\log p(J)}{\log(1+\varepsilon)} \right\rceil, \end{aligned}$$

as D is the smallest integer which the inequality holds for. If we assume that $p_{\max} = \text{poly}(n)$ then we also have

$$p(J) \leq n p_{\max} = n \,\text{poly}(n) = \text{poly}(n).$$

Using Lemma 2.2 we get

$$\log p(J) = O(\log n),$$

which in turn gives

$$D = O\Big(\frac{\log n}{\varepsilon}\Big). \tag{3.9}$$

Here we also use that $\log(1+\varepsilon) \geq \varepsilon \log 2$ for $\varepsilon \in (0,1)$, which gives

$$\frac{1}{\log(1+\varepsilon)} \leq \frac{1}{\varepsilon \log 2}.$$

Thus the number of variables has been reduced from $n \cdot p(J)$ to

$$n\left\lceil \frac{\log p(J)}{\log(1+\varepsilon)} \right\rceil = O\Big(n\frac{\log n}{\varepsilon}\Big)$$

through the discretisation.

3.1.2 The Objective Function

In the IP model, the variable $x_{j,d} \in \{0,1\}$ indicates that job $j \in J$ has completion time at most τ_d, if $x_{j,d} = 1$. Thus, if $\tau_d < C_j \leq \tau_{d+1}$ then

$$x_{j,1} = \cdots = x_{j,d} = 0,$$

and

$$x_{j,d+1} = \cdots = x_{j,D+1} = 1.$$

We can then obtain completion times from values for $(x_{j,d})_{j\in J, d\in[D+1]}$ by setting

$$\begin{aligned} C_j &:= \sum_{d=2}^{D+1} \tau_d (x_{j,d} - x_{j,d-1}) \\ &= \tau_{D+1} - \sum_{d=2}^{D} (\tau_{d+1} - \tau_d) x_{j,d} \\ &= \tau_{D+1} - \sum_{d=1}^{D} \eta_d x_{j,d}. \end{aligned} \tag{3.10}$$

To show that this is a sensible definition, consider the following: Let $(C_j^*)_{j\in J}$ be optimal completion times for some instance and let $(x_{j,d})_{j\in J, d\in[D+1]}$ be the corresponding values of the decision variables for this solution, i.e.

$$\tau_d < C_j^* \leq \tau_{d+1} \quad \Longrightarrow \quad \begin{cases} x_{j,1} = \cdots = x_{j,d} = 0, \\ x_{j,d+1} = \cdots = x_{j,D+1} = 1. \end{cases}$$

The definition (3.10) sets C_j to the latest possible completion time, that is

$$\tau_d < C_j^* \leq \tau_{d+1} \quad \Longrightarrow \quad C_j = \tau_{d+1}.$$

This results in the guarantee that

$$C_j = \tau_{d+1} = (1+\varepsilon)\tau_d < (1+\varepsilon) C_j^*,$$

which in turn gives us the result that

$$\sum_{j\in J} w_j C_j < (1+\varepsilon) \sum_{j\in J} w_j C_j^*, \tag{3.11}$$

i.e., the objective value is not increased by more than a factor of $(1+\varepsilon)$.

It is then clear that the objective function can be written

$$\begin{aligned}\sum_{j\in J} w_j C_j &= \sum_{j\in J} w_j \left(\tau_{D+1} - \sum_{d=1}^{D} \eta_d x_{j,d}\right) \\ &= \left(\sum_{j\in J} w_j\right)\tau_{D+1} - \sum_{j\in J} w_j \sum_{d=1}^{D} \eta_d x_{j,d} \\ &= w(J)\tau_{D+1} - \sum_{j\in J} w_j \sum_{d=1}^{D} \eta_d x_{j,d}.\end{aligned}$$

3.1.3 The Complete IP model

Here we summarise the model which we have described in Sections 3.1–3.1.2 The objective of our IP model is

$$\text{minimise} \quad w(J)\tau_{D+1} - \sum_{j\in J} w_j \sum_{d=1}^{D} \eta_d x_{j,d} \tag{3.12}$$

subject to the constraints

$$x_{j,d} \le x_{j,d+1} \qquad \forall j \in J, d \in [D], \tag{3.13}$$

$$x_{j,d} \ge x_{j',d} \qquad \forall j, j' \in J, d \in [D+1] : j \prec j' \tag{3.14}$$

$$\sum_{j\in J} p_j x_{j,d} \le \tau_d \qquad \forall d \in [D+1], \tag{3.15}$$

$$x_{j,d} = 0 \qquad \forall j \in J, d \in [D+1] : \tau_d < q_j, \tag{3.16}$$

$$x_{j,D+1} = 1 \qquad \forall j \in J, \tag{3.17}$$

$$x_{j,d} \in \{0,1\} \qquad \forall j \in J, d \in [D+1]. \tag{3.18}$$

The constraint (3.13) ensures that the variables are consistent with their interpretation. If $x_{j,d} = 1$, interpreted as the job completing before τ_d, then $x_{j,d+1} = 1$, interpreted as the job completing before τ_{d+1}. For two jobs $j \prec j'$, (3.14) states that if $x_{j',d} = 1$, i.e., the job j' completes before τ_d, then $x_{j,d} = 1$, i.e., the job j also completes before τ_d. The constraint (3.15) ensures that the total time spent on jobs before time τ_d does not exceed τ_d time units. The constraints (3.16) ensures that a job $j \in J$ does not complete before q_j and (3.17) simply states that all jobs end at or before time τ_{D+1}.

Note that (3.16) and (3.17) together with (3.13) give us that

$$0 = x_{j,1} \le x_{j,2} \le \cdots \le x_{j,D} \le x_{j,D+1} = 1 \quad \forall j \in J$$

as $\tau_1 = 0 < p_j \leq q_j$ for any $j \in J$. This implies that $x_{j,d} \in [0,1]$ for all $j \in J, d \in [D+1]$, so constraint (3.18) can be replaced by $x_{j,d} \in \mathbb{Z}$, without changing the model.

3.1.4 Maximum Length of Precedence Chains

Here we explain the details of how we compute q_j as defined in Section 3.1. Algorithm 4 computes $(q_j)_{j \in J}$ from the DAG of precedence constraints. This is done using dynamic programming, by first setting q_j to the processing time for all jobs j where $\Delta^-(j) = \varnothing$ and then working forwards in the graph, setting q_j to

$$p_j + \max_{j' \in \Delta^-(j)} q_{j'}.$$

Making use of solutions to previous subproblems in this manner is the defining characteristic of dynamic programming.

As stated above, the subproblems are simply to maximize $q_{j'}$ for $j' \in \Delta^-(j)$. Because we process the jobs in the topological order, we know that we have already solved this subproblem for each $j' \in \Delta^-(j)$, by the time we get to j. For the base case, i.e., when $\Delta^-(j) = \varnothing$, we simply set q_j to p_j. This takes $O(1)$ time, while the other subproblems take $O(|\Delta^-(j)|)$ time if we find the maximum by simply checking all elements of the set. Thus, Algorithm 4 takes

$$\underbrace{\sum_{j:\Delta^-(j)=\varnothing} O(1)}_{\leq |J| = n} + \underbrace{\sum_{j:\Delta^-(j)\neq\varnothing} O(|\Delta^-(j)|)}_{= |E((J,\prec))| = \kappa} = O(n+\kappa).$$

The algorithm uses a so-called *topological order*, defined as follows.

Definition 3.1. Given a graph $G = (V, E)$, a *topological order* is a order of V such that if $v \rightsquigarrow_G u$ for some $v, u \in V$, then v comes before u in the order.

A topological order can, for example, be found using depth first search. We describe the algorithm in Appendix A.1. When we call the function `longest_prec_chains`, `prec_order` is assumed to be a topological order compatible with `prec_graph`, stored as a vector of vertices. The variable `prec_graph` contains the graph $(J, \prec)$.

Algorithm 4 Compute the total processing times for precedence chains ending at each job.

```
function longest_prec_chains(inst, prec_graph::Graph, prec_order)
    q = zeros(Int, inst.n)
    # prec_order is a topological order compatible with prec_graph
    for j in prec_order
        # inst.p[j] is the length of the job j
        q′ = inst.p[j] + maximum(q[j′] for j′ in Δ_inc(prec_graph, j), init=0)
        @assert q[j] == 0
        q[j] = q′
    end
    q
end
```

3.2 LP Relaxation

The LP relaxation of the model in Section 3.1, given in [28, p. 2], found by simply removing constraint (3.18), is:

$$\text{minimise} \quad w(J)\tau_{D+1} - \sum_{j \in J} w_j \sum_{d=1}^{D} \eta_d x_{j,d}, \tag{3.19}$$

subject to

$$x_{j,d} \leq x_{j,d+1} \qquad \forall j \in J, d \in [D], \tag{3.20}$$

$$x_{j,d} \geq x_{j',d} \qquad \forall j \prec j', d \in [D+1], \tag{3.21}$$

$$\sum_{j \in J} p_j x_{j,d} \leq \tau_d \qquad \forall d \in [D+1], \tag{3.22}$$

$$x_{j,d} = 0 \qquad \forall j \in J, d \in [D+1] : \tau_d < q_j, \tag{3.23}$$

$$x_{j,D+1} = 1 \qquad \forall j \in J. \tag{3.24}$$

These constraints are exactly the same as (3.12)–(3.17). Again, we have the implicit constraint that $x_{j,d} \in [0,1]$ for all $j \in J, d \in [D+1]$, due to constraints (3.20), (3.23), (3.24), and the fact that $p_j > 0$ for every $j \in J$. Removing the integrality constraint cannot make the optimum worse, so if we find an optimal solution to this LP, its objective value will be at worst $(1+\varepsilon)$ times the optimum for the original scheduling problem (see (3.11)). However, it is not immediately clear how to go from fractional completion times to actual completion times, as the fractional completion times do not necessarily fulfill the constraints of the original problem. For this we need a *rounding algorithm*.

3.3 Single Machine Rounding Algorithm

From the LP relaxation in Section 3.2 we obtain a set of *fractional completion times* (see [28, Definition of C_j, p. 9]). These are computed by the same expression as the integer completion times, (3.10), i.e.,

$$C_j = \tau_{D+1} - \sum_{d=1}^{D} \eta_d x_{j,d}, \tag{3.25}$$

which makes it intuitively reasonable that we should be able to use them to sort our jobs correctly. Here $(x_{j,d})_{j\in J, d\in[D+1]}$ is a solution to (3.19)–(3.24). The following property of the fractional completion times is useful.

Theorem 3.2 ([28, Claim 3.3])**.** *For a job $j \in J$, we have $C_j \geq q_j$. For jobs $j \prec j'$, we have $C_j \leq C_{j'}$.*

Proof. Let j be any job and let d' be the smallest integer such that $x_{j,d'} > 0$. The constraint (3.23) gives us that

$$x_{j,d} = 0 \quad \text{for all } d \in [D+1] \text{ such that } \tau_d < q_j,$$

thus $\tau_{d'} \geq q_j$, and we know that

$$\begin{aligned} C_j &= \tau_{D+1} - \sum_{d=1}^{D} \eta_d x_{j,d} = \sum_{d=1}^{D} \eta_d - \sum_{d=1}^{D} \eta_d x_{j,d} \\ &= \sum_{d=1}^{D} \eta_d (1 - x_{j,d}) \geq \sum_{d=1}^{d'-1} \eta_d = \tau_{d'} \geq q_j, \end{aligned}$$

where both the first and second inequality is due to (3.23).

Assume we have some jobs $j \prec j'$. Then we have

$$\begin{aligned} C_{j'} - C_j &= \tau_{D+1} - \sum_{d=1}^{D} \eta_d x_{j',d} - \tau_{D+1} + \sum_{d=1}^{D} \eta_d x_{j,d} \\ &= \sum_{d=1}^{D} \eta_d x_{j,d} - \sum_{d=1}^{D} \eta_d x_{j',d} \\ &= \sum_{d=1}^{D} \eta_d (x_{j,d} - x_{j',d}) \geq 0, \end{aligned}$$

as each term of the sum is greater than or equal to zero, by (3.21). □

Note that this proof assumes that we solve the LP exactly. The MWU algorithm only solves it approximately, as we will see in Definition 3.3 and later in Section 3.4. However, only the optimality of the objective value and the constraint (3.22) are approximated, and since the proof only uses constraints (3.23) and (3.21), this is not an issue.

A valid schedule can be computed by Algorithm 5, an application of the *list scheduling* algorithm [12]. We re-use the topological order, used in Algorithm 4,

Algorithm 5 Find completion times given a set of fractional completion times.

```
1  function round_completion_times(inst::Instance, C::Vector, prec_order::Vector)
2      ρ = ordering_to_rank(prec_order)
3      J_sorted = sort(collect(1:inst.n), by=j -> (C[j],ρ[j]), alg=MergeSort)
4
5      # inst.n is the number of jobs
6      # inst.p[j] is the completion time for job j
7      C = zeros(Int, inst.n)
8      for (j,C_j) in zip(J_sorted, cumsum(inst.p[J_sorted]))
9          C[j] = C_j
10     end
11     C
12 end
```

in order to find a *rank function* for J, the vertices of the precedence graph. A rank function is a function $\rho\colon J \to \mathbb{Z}$ such that $\rho(j) < \rho(j')$ if j comes before j' in the order. We use it in order to break ties. This is necessary if we have a situation where

$$C_{j_1} = \cdots = C_{j_k}$$

for some $j_1, \ldots, j_k$. The fractional completion times give us no information about the order of these jobs, but they cannot be scheduled arbitrarily because of precedence constraints. However, if $j \prec j'$ then $\rho(j) < \rho(j')$ and the jobs are scheduled in the correct order.

The run time of Algorithm 5 can be computed by noting that the merge sort on line 3 takes $O(n \log n)$ time, while computing the cumulative sum and setting `C[j]` on lines 8–10 takes $O(n)$ time. Thus the total run time of the rounding algorithm is $O(n \log n)$, where $n = |J|$.

3.3.1 Approximation Ratio of the Complete Algorithm

The following result proves that solving the LP relaxation and applying the rounding algorithm, is a $(2 + O(\varepsilon))$-approximation algorithm for the problem $1|\text{prec}|\sum_j w_j C_j$. In it, we use a solution provided by Algorithm 6, which is an algorithm that solves the problem (3.19)–(3.24) approximately. This will be explained in Section 3.4. It provides solutions to the following problem.

Definition 3.3. Let $\mathrm{LP}_{\mathrm{apx}}$ denote the problem of finding an $\mathbf{x}$ satisfying (3.20), (3.21), (3.23), (3.24) and

$$\sum_{j\in J} p_j x_{j,d} \leq (1+\varepsilon)\tau_d \quad \forall d \in [D+1] \tag{3.22$'$}$$

such that

$$w(J)\tau_{D+1} - \sum_{j\in J} w_j \sum_{d=1}^{D} \eta_d x_{j,d} \leq \mathrm{OPT}_{\mathrm{LP}} + \phi, \tag{3.19$'$}$$

where $\mathrm{OPT}_{\mathrm{LP}}$ is the optimal objective value for (3.19)–(3.24) and $\phi \leq \varepsilon \cdot \mathrm{OPT}$ is a parameter, where OPT is the weighted completion time of an optimal schedule.

Theorem 3.4 ([28, Lemma 3.4])**.** *Let $C' \geq 0$ be a time point and let $(C_j)_{j\in J}$ be the fractional completion times obtained from a solution to LP_{apx}. Then we have*

$$p(\{j \in J : C_j \leq C'\}) \leq (2 + O(\varepsilon))mC'.$$

For a proof of this see [28, Proof of Theorem 3.4, p. 21].

In fact, from this proof together with [28, Proof of Theorem 2.4, p. 20], follows a more detailed bound, namely:

Theorem 3.5. *Let C' be a time point and let $(C_j)_{j\in J}$ be fractional completion times as in Theorem 3.4. Then it holds that*

$$p(\{j \in J : C_j \leq C'\}) \leq 2 \cdot \frac{(1+\varepsilon)^2 + \varepsilon}{1-\varepsilon} mC'. \tag{3.26}$$

Given that Theorem 3.5 holds, Algorithm 5 schedules jobs such that for a job j,

$$\tilde{C}_j \leq p(\{j' \in J : C_{j'} \leq C_j\}),$$

where $(\tilde{C}_j)_{j\in J}$ is the output of Algorithm 5 given the fractional completion times $(C_j)_{j\in J}$ (which come from a solution to $\mathrm{LP}_{\mathrm{apx}}$). This is because the job j cannot be scheduled after any jobs having a larger fractional completion time

than C_j. Thus,

$$\begin{aligned}
\sum_{j\in J} w_j \tilde{C}_j &\le \sum_{j\in J} w_j p(\{j' \in J : C_{j'} \le C_j\}) \\
&\le 2\frac{(1+\varepsilon)^2+\varepsilon}{1-\varepsilon} \cdot \sum_{j\in J} w_j C_j \\
&\le 2\frac{(1+\varepsilon)^2+\varepsilon}{1-\varepsilon} \big(\mathrm{OPT}_{\mathrm{LP}} + \phi \big) \\
&\le 2\frac{(1+\varepsilon)^2+\varepsilon}{1-\varepsilon} \big(\mathrm{OPT}_{\mathrm{IP}} + \varepsilon \cdot \mathrm{OPT} \big) \\
&\le 2\frac{(1+\varepsilon)^2+\varepsilon}{1-\varepsilon} \big((1+\varepsilon) \cdot \mathrm{OPT} + \varepsilon \cdot \mathrm{OPT} \big) \\
&= 2\frac{(1+\varepsilon)^2+\varepsilon}{1-\varepsilon} (1+2\varepsilon) \cdot \mathrm{OPT}
\end{aligned}$$

where $\mathrm{OPT}_{\mathrm{LP}}$ is the optimal objective value for (3.19)–(3.24), $\mathrm{OPT}_{\mathrm{IP}}$ is the optimal objective value for (3.12)–(3.18) and OPT is the weighted completion time of an optimal schedule. Here we use (3.11), together with (3.19′), with the parameter ϕ set to its maximum value. We can rewrite

$$\begin{aligned}
2\frac{(1+\varepsilon)^2+\varepsilon}{1-\varepsilon}(1+2\varepsilon) &= \frac{2+10\varepsilon+14\varepsilon^2+4\varepsilon^3}{1-\varepsilon} \\
&= 2+12\varepsilon+\frac{26\varepsilon^2+4\varepsilon^3}{1-\varepsilon} \\
&= 2+12\varepsilon+\frac{2\varepsilon^2(13+2\varepsilon)}{1-\varepsilon},
\end{aligned}$$

which shows that:

Theorem 3.6. *Let* OPT *be the weighted completion time of an optimal schedule. Let* OBJ_{round} *be the objective value for the solution generated by Algorithm 5 run on the completion times given by a solution to* LP_{apx}*.*

Then $\mathrm{OBJ}_{round} \le \alpha(\varepsilon) \cdot \mathrm{OPT}$*, where*

$$\alpha(\varepsilon) := 2+12\varepsilon+\frac{2\varepsilon^2(13+2\varepsilon)}{1-\varepsilon} = 2+O(\varepsilon),$$

for the parameter $\varepsilon \in (0,1)$*.*

See Figure 3.3 for an illustration of the approximation ratio given different values of ε.

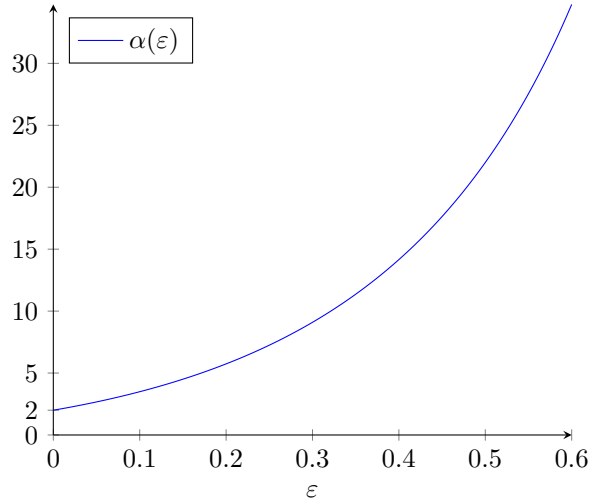


Figure 3.3: Plot of the approximation ratio $\alpha(\varepsilon)$ as a function of ε.

3.4 MWU Based Solver

We now need to solve the LP relaxation, which can be done quickly (i.e., in almost nearly linear time) using an MWU based solver for linear programs of a certain form [28, pp. 7–8].

This subset of LPs, called packing LPs, are in general described as follows: Let $\mathbf{P} \in \mathbb{R}_{\geq 0}^{m \times n}$ be a nonnegative matrix. Let $\mathbf{a} \in \mathbb{R}_{\geq 0}^{n}$ be a row vector. Finally, let $\mathcal{Q} \subseteq \mathbb{R}_{\geq 0}^{n}$ be a polytope which is defined by "easy" constraints, i.e., constraints which we do not need to "remove" using the MWU algorithm, but which can be left to be handled in the oracle for the simpler LP (3.32), as described in Section 3.4.2. The LP which is solved by the MWU-based solver, Algorithm 6, is

$$\text{maximise } \mathbf{ax} \quad \text{subject to} \quad \mathbf{x} \in \mathcal{Q} \quad \text{and} \quad \mathbf{Px} \leq \mathbf{1}. \tag{3.27}$$

The performance of our solver, Algorithm 6, is guaranteed by the following theorem:

Theorem 3.7 ([28, Theorem 2.4])**.** *Given parameters $\varepsilon \in (0, 1)$ and $\phi > 0$, Algorithm 6 will return an approximate solution $\mathbf{x}$ to (3.27) such that*

$$x \in \mathcal{Q}, \quad \mathbf{Px} \leq (1 + O(\varepsilon))\mathbf{1} \quad \textit{and} \quad \mathbf{ax} \geq \mathbf{ax}^* - \phi, \tag{3.28}$$

where $\mathbf{x}^*$ *is the optimal solution to* (3.27)*. This will be done within* $O\left(\frac{m \log m}{\varepsilon^2}\right)$ *iterations of the MWU loop (lines 14–31 of Algorithm 6), where* m *is the number of rows in the matrix* $\mathbf{P}$*.*

For a proof, see [28, p. 20]. From this proof a more precise guarantee on the constraint can be extracted, which avoids the $O(\varepsilon)$ term.

Theorem 3.8. *Given parameters* $\varepsilon \in (0, 1)$ *and* $\phi > 0$*, Algorithm 6 returns an approximate solution* $\mathbf{x}$ *to* (3.27) *such that*

$$\mathbf{Px} \leq ((1 + \varepsilon)^2 + \varepsilon)\mathbf{1}. \tag{3.29}$$

and

$$\mathbf{ax} \geq \mathbf{ax}^* - \phi. \tag{3.30}$$

Now we know that we can use MWU to solve packing LPs on the form (3.27). Before we examine the details of the MWU algorithm in Section 3.4.2, we must first reformulate our linear program, (3.19)–(3.24), as a packing LP.

3.4.1 Preprocessing of the LP Relaxation

Before we can apply Algorithm 6, we need to convert the linear program (3.19)–(3.24) to the form (3.27). To do this, we start by defining the set V as the set of all (j, d) such that $x_{j,d}$ is not fixed to 0 or 1 by (3.23) or (3.24), respectively. That is, set

$$V := \{(j, d) : j \in J, d \in [D], \tau_d \geq q_j\}.$$

Then we define E by adding an edge for each constraint (3.20) and (3.21), so that

$$\begin{aligned} E := &\{((j, d), (j, d + 1)) : (j, d) \in V, (j, d + 1) \in V\} \\ &\cup \{((j', d), (j, d)) : j \prec j', (j, d) \in V, (j', d) \in V\}. \end{aligned}$$

It is clear that the graph $G = (V, E)$ is acyclic, because there can be no cyclic precedence constraints. If there were any such, the instance would not have any solution. From the graph G we can derive the polytope

$$\mathcal{Q} := \{\mathbf{x} \in [0, 1]^V : x_v \leq x_u, \forall (v, u) \in E\},$$

consisting of the values $\mathbf{x}$ which fulfill the "easy" constraints (3.20) and (3.21). Because $\mathcal{Q}$ is derived from the graph G, the set itself is not stored in memory in the implementation, but only G.

Now let $\mathbf{P} \in \mathbb{R}_{\geq 0}^{[D+1]\times V}$ be such that the constraint (3.22) can be written $\mathbf{Px} \leq \mathbf{1}$. This can be done by setting

$$P_{d',(j,d)} = \begin{cases} p_j/\tau_d & \text{if } d = d', \\ 0, & \text{otherwise,} \end{cases}$$

so that each sum in (3.22) corresponds to one row of $\mathbf{P}$ multiplied by $\mathbf{x}$. This also gives us that $\mathbf{P}$ has $\bar{N} := |V|$ nonzero elements, as each vertex $v \in V$ corresponds to exactly one non-zero element of $\mathbf{P}$.

Finally, let $a_{j,d} := w_j\eta_d$ for $(j,d) \in V$. The objective (3.19) is then to minimise

$$\begin{aligned} w(J)\tau_{D+1} - \sum_{j\in J} w_j \sum_{d=1}^{D} \eta_d x_{j,d} &= w(J)\tau_{D+1} - \sum_{j\in J}\sum_{d=1}^{D} w_j\eta_d x_{j,d} \\ &= w(J)\tau_{D+1} - \mathbf{ax}, \end{aligned} \tag{3.31}$$

that is, to maximise $\mathbf{ax}$.

3.4.2 The MWU Algorithm

The MWU algorithm we have implemented is almost the same as Algorithm 3. It solves linear programs on the form (3.27) such that Theorem 3.8 holds. Our implementation is found in Algorithm 6.

It works by solving a simpler LP in each step, which is of the form

$$\text{maximise } \mathbf{ay} \quad \text{subject to} \quad \mathbf{y} \in \mathcal{Q} \quad \text{and} \quad \mathbf{by} \leq 1. \tag{3.32}$$

To solve this LP an *oracle*, i.e., a "black box" that can perform a specified function, is used.

Theorem 3.9 ([28, Theorem 3.2])**.** *Let $G = (V, E)$ be a DAG and $\mathcal{Q} := \{\mathbf{y} \in [0,1]^V : y_v \leq y_u \ \forall (v,u) \in E\}$. Let $\mathbf{a}, \mathbf{b} \in \mathbb{R}_{\geq 0}^V$ be two row vectors. Let*

$$\mathbf{y}^* := \operatorname*{arg\,max}_{\mathbf{y}\in\mathcal{Q}, \mathbf{by}\leq 1} \mathbf{ay}.$$

Given the two parameters $\varepsilon \in (0,1)$ and $\phi \in (0, |\mathbf{a}|_1/2)$, we can find an $\mathbf{y} \in \mathcal{Q}$ satisfying $\mathbf{by} \leq 1+\varepsilon$ and $\mathbf{ay} \geq \mathbf{ay}^* - \phi$ in $\tilde{O}_\varepsilon\Big(|E| \cdot \big(\log \frac{|\mathbf{a}|_1}{\phi}\big)^2\Big)$ time.

We will use $\mathcal{O}$ to represent an oracle that finds such a $\mathbf{y}$. How $\mathcal{O}$ works will be explained in more detail in the following section. Algorithm 6 is an adaptation of Algorithm 3, which we explained in Section 2.7.1. Note that we have replaced the parameter η with $\varepsilon\rho$, where $\rho = \log \hat{m}/\varepsilon^2$, which is equivalent to setting $\eta = \log \hat{m}/\varepsilon$. We set

$$\phi := \varepsilon \cdot w(J). \tag{3.33}$$

Algorithm 6 A MWU based approximate solver for the LP (3.27).

```
function mwu_solve_lp(a::Vector, P::Matrix, G::Graph, ϵ, ϕ)
    @assert 0 < ϵ < 1
    @assert ϕ > 0

    (m̂,n̂) = size(P)
    @assert length(a) == n̂

    R = typeof(ϵ) # float type used
    t = zero(R)
    ρ = log(m̂) / ϵ^2
    x = zeros(R, n̂)
    u = ones(R, m̂)' # row vector

    while t < 1
        b = (u/norm(u, 1) ∗ P)' # row vector but transposed for convenience

        ϵ′ = sqrt(1+ϵ) - 1
        y = oracle_solve_lp(a, b, G, ϵ′, ϕ)

        # note that we validate with the original epsilon as we want by ≤ (1+ϵ′)^2 ≤ 1+
    ϵ
        validate_oracle_solution(b, G, y, ϵ)

        δ = min(minimum(1/(ρ ∗ P[i,:]'∗y) for i in 1:m̂), 1-t)

        for i in 1:m̂
            # P[i,:]'∗y == P[i:i,:] ∗ y
            u[i] = u[i] ∗ exp(δ∗ϵ∗ρ ∗ P[i,:]' ∗ y)
        end
        x = x + δ∗y
        t = t + δ
    end
    x
end
```

3.5 The Approximate Oracle $\mathcal{O}$

To implement the oracle $\mathcal{O}$ we need a way to find solutions to the LP (3.32). However, as our instance of the LP originates from our scheduling problem, it can be shown that a few assumptions can be made, without loss of generality. These assumptions are described in Theorem 3.10 and Theorem 3.11.

This allows us to simplify the problem by applying some "preprocessing" and "postprocessing" steps, giving us the equivalent LP (3.41)–(3.45). Finally, we find the dual of this LP. This dual can be split up into a set of subproblems, which are network flow problems. It can be shown that by solving a finite number of such subproblems, we can (approximately) solve the original problem. This is the focus of Chapter 4.

The structure of our implementation of $\mathcal{O}$ is explained by Figure 3.4. In this figure, and in the rest of this section, we let $G = (V, E)$, $\mathcal{Q}, \mathbf{a}, \mathbf{b}, \varepsilon$ and ϕ be

as in Theorem 3.9. The sections Section 3.5.1, Section 3.5.2 and Section 3.5.3, correspond to the steps in the figure *Preprocess DAG*, *Postprocess*, and *Find Dual*, respectively. The step *Solve* will be described in Chapter 4.

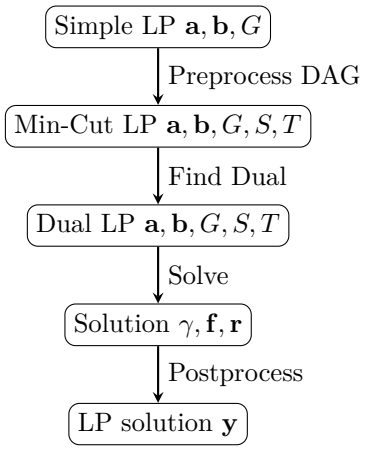


Figure 3.4: Overview of the implementation of the oracle $\mathcal{O}$.

3.5.1 Preprocessing the DAG

Define $S := \{s \in V : a_s > 0\}$ and $T := \{t \in V : b_t > 0\}$. The vertices in S can be thought of as sources of flow, and the vertices in T as sinks. The particular flow problem we consider will become clear later.

Recall that the LP which we wish to solve is

$$\begin{aligned} \text{maximise} \quad & \mathbf{ay}, && (3.34)\\ \text{subject to} \quad & \mathbf{by} \leq 1, && (3.35)\\ & y_v \leq y_u \quad \forall (v,u) \in E, && (3.36) \end{aligned}$$

given graph $G = (V, E)$ and the vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^V_{\geq 0}$. Note in particular that the constraint (3.36) implies that

$$y_v \leq y_u \quad \text{for all } v, u \in V \text{ such that } v \rightsquigarrow u.$$

Thus, if we fix the value of y_v, it imposes a minimum value on y_u for all nodes u which are reachable from v. Likewise, it imposes a maximum value on y_v for all nodes v from which u is reachable.

Before we continue, we prove some properties of these two sets, namely:

Theorem 3.10. *Without loss of generality, we can assume that*

- $S \cap T = \varnothing$,
- *there is no edge* $(s,t) \in E$ *from* $s \in S$ *to* $t \in T$,
- $\delta^-(s) = \varnothing$ *for every* $s \in S$,
- $\delta^+(t) = \varnothing$ *for every* $t \in T$,
- *for every* $v \in V$ *we have* $S \rightsquigarrow v$ *and* $v \rightsquigarrow T$.

Proof. This is done by a "preprocessing" step, which changes S and T, and fixes some y_v, to guarantee the properties. For any vertex v such that $S \not\rightsquigarrow v$, set $y_v := 0$ and remove it from V. The reason this is the best value for y_v is that if we have a $t \in T$ such that $v \rightsquigarrow t$, then we know $y_t \geq y_v$, and we want $b_t y_t$ to be as small as possible (due to (3.35)). Setting y_v to zero does not impose any restrictions on y_s for $s \in S$, as we assumed that $s \not\rightsquigarrow v$. Thus we do not constrain the value of our objective function

$$\mathbf{ay} = \sum_{s \in S} a_s y_s$$

by this preprocessing step.

For any vertex v such that $v \not\rightsquigarrow T$, set $y_v := 1$ and remove it from V. This is the best value for y_v because if we have $s \in S$ such that $s \rightsquigarrow v$, then $y_s \leq y_v$ and we want to maximise $a_s y_s$ (due to (3.34)). This does not impose any restrictions on the value of $\mathbf{by}$, by a similar reasoning as before.

For every $s \in S$, add a new vertex s' to V and a new edge (s', s) to E. Set $a_{s'} := a_s$, $b_{s'} := 0$ and change a_s to 0. Remove s from S and add s' instead. For every $t \in T$, add a new vertex t' to T and a new edge (t, t') to E. Set $a_{t'} := 0$, $b_{t'} := b_t$ and change b_t to 0. Remove t from T and add t' instead. See Figure 3.5 for an illustration of this.

This does not change the instance as the existence of the edge between s' and s implies that $y_{s'} \leq y_s$, and this is the only constraint on $y_{s'}$. It is thus best to set $y_{s'} = y_s$ in order to maximise the term $a_{s'} y_{s'}$. Likewise, we have $y_{t'} \geq y_t$ and it is best to set $y_{t'} = y_t$ in order to minimise the term $b_{t'} y_{t'}$.

It now clearly holds that $S \cap T = \varnothing$. We also have that

$$\delta^-(s') = \varnothing$$

and

$$\delta^+(t') = \varnothing$$

for any $s' \in S$ and $t' \in T$, respectively. Further, $\delta^+(s')$ contains only an edge to the vertex which was "replaced" by s'. This vertex is not in T, so it holds that we have no edge between S and T. □

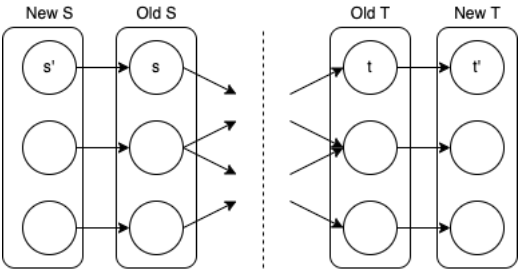


Figure 3.5: A sketch of the modifications of S and T in Theorem 3.10.

Our implementation of this procedure is found in Algorithm 7. A general theme of the implementation is the need to keep track of all transformations done. For instance, in Section 3.4.1 we need to record which variables $x_{j,d}$ are fixed and which are added to the set V. Here, in Algorithm 7, we also need to record which variables y_v are fixed to zero or one, and which are kept as variables. This is done using the vectors `fix_y` and `new_v`. In the code, the set V is a set of integers instead of a subset of $J \times [D+1]$. The vector `new_v` contains the new index corresponding to y_v. In the vector `old_v` we also maintain the "backwards" mapping.

3.5.2 Postprocessing the solution

We also prove that the constraints can be simplified slightly:

Theorem 3.11. *Without loss of generality, we can replace the constraint* $\mathbf{y} \in [0,1]^V$ *in Theorem 3.9 with*

$$\begin{aligned} y_s &\leq 1 \quad \forall s \in S, \\ y_t &\geq 0 \quad \forall t \in T. \end{aligned}$$

Proof. This is done through a "postprocessing" step which takes any solution to the weaker version of the problem and produces a solution to the stronger

version. Assume that we have a (approximate) solution $\mathbf{y}$ to

$$\begin{aligned} \text{maximise} \quad & \mathbf{a}^T\mathbf{y}, \\ \text{subject to} \quad & \mathbf{b}^T\mathbf{y} \leq 1, && (3.37) \\ & y_v \leq y_u \quad \forall (v,u) \in E, && (3.38) \\ & y_s \leq 1 \quad \forall s \in S, && (3.39) \\ & y_t \geq 0 \quad \forall t \in T, && (3.40) \end{aligned}$$

for some $\mathbf{a}, \mathbf{b} \in \mathbb{R}^V_{\geq 0}$, where

$$\begin{aligned} a_s > 0 \quad &\iff \quad s \in S, \\ b_t > 0 \quad &\iff \quad t \in T. \end{aligned}$$

Then, we can modify the solution by changing y_v to

$$\max_{s \in S : s \rightsquigarrow v} y_s \quad \text{for all } v \in V \setminus S.$$

This is the smallest value we can use for y_v (given these $(y_s)_{s \in S}$) and still satisfy constraint (3.38), which implies

$$y_v \geq y_s \quad \text{for all } v \in V \setminus S, s \in S \text{ such that } s \rightsquigarrow v.$$

As no y_s is greater than 1, this gives us a new solution such that $y_v \leq 1$ for all $v \in V$. Note in particular that this holds for $t \in T$. Because we assume (3.37), and we have not increased $\mathbf{y}$, this constraint still holds. The objective function $\mathbf{ay}$ is unchanged because we have not modified $(y_s)_{s \in S}$.

Now we modify the solution again, by changing y_v to

$$\min_{t \in T : v \rightsquigarrow t} y_t \quad \text{for all } t \in V \setminus T.$$

This is the largest value we can use for y_v and still satisfy the constraint (3.38), as

$$y_v \leq y_t \quad \text{for all } v \in V \setminus T, t \in T \text{ such that } v \rightsquigarrow t.$$

Because no y_t is less than 0 or (by the above) greater than 1, this gives us a new solution such that $y_v \in [0,1]$ for all $v \in V$. We assumed that $\mathbf{ay}$ was maximal, and because we have not decreased $\mathbf{y}$, this is still true. The constraint (3.37) holds as we have not modified any $(y_t)_{t \in T}$. □

Our implementation of this step is found in Algorithm 8.

3.5.3 The Dual of the LP as a Flow Problem

The LP (3.32) can now be rewritten as

$$\text{maximise} \quad \sum_{s\in S} a_s y_s, \tag{3.41}$$

$$\text{subject to} \quad \sum_{t\in T} b_t y_t \le 1, \tag{3.42}$$

$$y_v \le y_u \quad \forall (v,u) \in E, \tag{3.43}$$

$$y_s \le 1 \quad \forall s \in S, \tag{3.44}$$

$$y_t \ge 0 \quad \forall t \in T. \tag{3.45}$$

The objective (3.41) and the constraint (3.42) are simply maximise $\mathbf{ay}$ and $\mathbf{by} \le 1$, respectively. We note that

$$\mathbf{y} \in \mathcal{Q} \quad \Longleftrightarrow \quad \mathbf{y} \in [0,1]^V \quad \text{and} \quad y_v \le y_u \quad \forall (v,u) \in E,$$

so we get the constraint (3.43) and according to Theorem 3.11, $\mathbf{y} \in [0,1]^V$ can be written as (3.44) and (3.45).

This LP has the dual

$$\text{minimise} \quad \gamma + \sum_{s\in S} r_s, \tag{3.46}$$

$$\text{subject to} \quad f(\delta^+(s)) + r_s = a_s \quad \forall s \in S, \tag{3.47}$$

$$\gamma b_t - f(\delta^-(t)) \ge 0 \quad \forall t \in T, \tag{3.48}$$

$$f(\delta^+(v)) - f(\delta^-(v)) = 0 \quad \forall v \in V \setminus (S \cup T), \tag{3.49}$$

$$\gamma \ge 0, \tag{3.50}$$

$$f_{v,u} \ge 0 \quad \forall (v,u) \in E, \tag{3.51}$$

$$r_s \ge 0 \quad \forall s \in S. \tag{3.52}$$

The variables γ, $(f_{v,u})_{(v,u)\in E}$ and $(r_s)_{s\in S}$ correspond to the constraints (3.42), (3.43) and (3.44), respectively. The constraints (3.47), (3.48) and (3.49) correspond to variables $(y_s)_{s\in S}$, $(y_t)_{t\in T}$ and $(y_v)_{v\in V\setminus(S\cup T)}$, respectively. We only require that $y_t \ge 0$ for $t \in T$, so therefore (3.47) and (3.49) are equalities. In Section 2.4.4 we explain this mechanical method of constructing the dual.

The dual problem looks like a network flow problem, if we fix γ. The constraint (3.49) can be interpreted as conservation of flow, while (3.48) looks like a constraint on the total flow into the vertex $t \in T$. The constraint (3.47) can also be interpreted as a capacity constraint, as we will see later on.

What remains now is to find an approximate solution to (3.41)–(3.45). In Chapter 4 we will show that solving (3.46)–(3.52) for a finite set of fixed γ allows us to obtain such a solution.

Algorithm 7 A procedure for the transformations discussed in the proof of Theorem 3.10.

```
function transform_data(a::Vector, b::Vector, G::Graph)
    S = [s for s in 1:G.n if a[s] > 0]
    T = [t for t in 1:G.n if b[t] > 0]

    fix_y = Vector{Union{Nothing,Int}}(nothing, G.n)
    new_v = Vector{Union{Nothing,Int}}(nothing, G.n) # new_v[v] = v′
    old_v = Int[] # old_v[v′] = v
    reach_from_S = falses(G.n)
    for v in reach_from(G, S)
        reach_from_S[v] = true
    end
    reach_to_T = falses(G.n)
    for v in reach_to(G, T)
        reach_to_T[v] = true
    end
    for v in 1:G.n
        if !reach_from_S[v] # not reachable from S, remove and fix y[v] to 0
            fix_y[v] = 0
        elseif !reach_to_T[v] # not reachable from T, remove and fix y[v] to 1
            fix_y[v] = 1
        else # otherwise, keep the vertex
            push!(old_v, v)
            new_v[v] = length(old_v)
        end
    end

    new_edges = [
        (new_v[src_vertex(G, e)],new_v[dst_vertex(G, e)]) for e in edges(G)
            if !isnothing(new_v[src_vertex(G, e)])
                && !isnothing(new_v[dst_vertex(G, e)])]
    new_n = length(old_v)
    new_a = zeros(eltype(a), new_n)
    new_b = zeros(eltype(b), new_n)

    for s in S
        if new_v[s] === nothing
            continue
        end
        s′ = new_n + 1
        new_n += 1
        push!(new_a, a[s])
        push!(new_b, zero(eltype(b)))
        push!(new_edges, (s′,new_v[s]))
    end

    for t in T
        if new_v[t] === nothing
            continue
        end
        t′ = new_n + 1
        new_n += 1
        push!(new_a, zero(eltype(a)))
        push!(new_b, b[t])
        push!(new_edges, (new_v[t],t′))
    end

    new_G = Graph(new_n, new_edges)
    new_S = [s for s in 1:new_n if new_a[s] > 0]
    new_T = [t for t in 1:new_n if new_b[t] > 0]
    (new_a,new_b,new_G,new_S,new_T,fix_y,new_v)
end
```

Algorithm 8 Postprocessing to simplify the constraints as stated in Theorem 3.11.

```
function postprocess_y(y::Vector, G::Graph, a::Vector, b::Vector)
    G_order = topological_ordering(G)
    # set y[v] = max(y[s] for s in S if (v reachable from s) for v in V \ S
    for v in G_order
        if a[v] != 0 # faster way to check if a in S
            continue
        end
        y[v] = maximum(y[u] for u in Δ_inc(G, v))
    end
    # set y[v] = min(y[t] for t in T if (t reachable from v) for v in V \ T
    for v in Iterators.reverse(G_order)
        if b[v] != 0 # faster way to check if b in T
            continue
        end
        y[v] = minimum(y[u] for u in Δ_out(G, v))
    end
    y
end
```

Chapter 4

Network Flow Algorithm

Here we describe how to solve the network flow problem which needs to be solved as a part of the algorithm described in Chapter 3. We also explain our implementation of this solution method. The implementation consists of a number of procedures, each related to a lemma or algorithm in Li's paper [28, Appendix D, pp. 28–34].

See Figure 4.1 for an overview of how these procedures interact. The subproblem NFP_γ which is produced by the step *Split Into Subproblems*, and its dual are described in Section 4.1.1 and Section 4.1.2, respectively. The step *Construct Long Paths* is implemented in Algorithm 11 and described in Section 4.2. The final step *Reduce Over* γ is implemented in Algorithm 9, and is described in Section 4.1.4.

To start with, after the simplifications in Chapter 3, we know that an approximate solution to $1|\text{prec}|\sum_j w_j C_j$ can be found by solving the LP (3.46)–(3.52) with parameters given by the vectors $\mathbf{a}, \mathbf{b}$, the graph G and the sets S, T. As we will explain later on, this LP can be solved approximately by solving a number of maximum flow problems $(\text{NFP}_\gamma)_{\gamma\in\Gamma}$, in the graph G where the sources S have capacity $\mathbf{a}$ and the sinks T have capacity $\gamma\mathbf{b}$. Thus, we will need an efficient algorithm for solving NFP_γ.

An algorithm for solving a similar maximum flow problem (with a single source and sink) is Dinitz' algorithm [8]. This algorithm is based on finding a *blocking flow* (see Definition 4.12) in a graph G_L, called the *level graph*. The level graph is defined as $G_L = (V, E_L)$, where

$$E_L = \{(v, u) \in E_{\mathbf{f}} : \text{dist}(v) + 1 = \text{dist}(u)\},$$

$\text{dist}(v)$ gives the length of the shortest path from the source to v, and $E_{\mathbf{f}}$ is the set of edges in the *residual graph* for $\mathbf{f}$ and G, defined below. One can think of

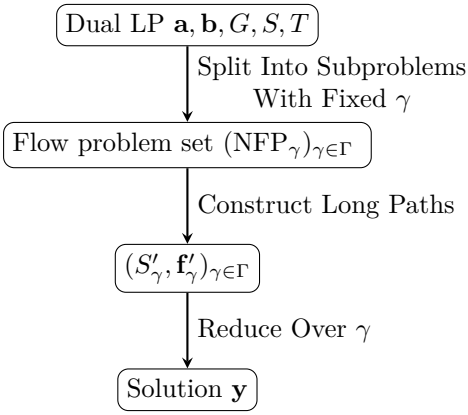


Figure 4.1: A sketch of the "inner layer" of the algorithm.

this as removing all edges from the residual graph which do not "increase the distance", i.e., make progress towards the sink. The residual graph $G_{\mathbf{f}} = (V, E_{\mathbf{f}})$ has the set of edges $E_{\mathbf{f}}$, which is the set of edges in E on which we can still add more flow, together with a "reverse" edge for each edge in E on which we can remove flow.

If we augment $\mathbf{f}$ by a blocking flow $\mathbf{f}'$ (that is, add $\mathbf{f}'$ to $\mathbf{f}$), this amounts to removing the edges that have reached their capacity from $G_{\mathbf{f}}$ (replacing them by edges going "the other way"). One can prove that the length of the shortest augmenting paths increases by one, after each iteration of the main loop of Dinitz' algorithm. Thus a blocking flow needs to be found at most $O(|V|)$ times.

Li's algorithm is similar to Dinitz' algorithm in that it is based around finding a blocking flow in the graph a number of times, in order to increase the length of the shortest augmenting paths.

The reason we cannot use Dinitz' algorithm directly, is that the run time would be too large. However, if G were a directed bipartite graph from S to T, then we could use Dinitz' algorithm. This is because it is possible to show that if the residual graph does not contain an augmenting path of length at most $2L + 1$, where L is "nearly constant" with respect to n and κ, then we can find an approximate solution to the problem in nearly linear time. See Theorem 4.9.

In each iteration, can find a blocking flow in the residual graph $G_{\mathbf{f}}$, restricted to shortest augmenting paths in nearly linear time, using *dynamic trees*. In Li's algorithm, augmenting $\mathbf{f}$ by this blocking flow increases the length of augmenting paths by 2. See Algorithm 15.

Given any G that fulfills Theorem 3.10, we can obtain such a directed bipartite graph H from S to T, by letting $H = (S \cup T, E_H)$ where $(s,t) \in E_H$ if and only if $s \rightsquigarrow_G t$ for some $s \in S, t \in T$. We call H the *shortcut graph*. However, the graph H is too big to create (the number of edges can be quadratic in $|V|$).

Therefore, H is "emulated" it using the original graph G, letting each path from S to T in G correspond to an edge from S to T in H, called a *forward shortcut edge*. We also let each path from S to T in $\operatorname{supp}(\mathbf{f})$ correspond to an edge from T to S in H, called a *backward shortcut edge*. A shortest augmenting path in H corresponds to an augmenting path in G with the minimum number of "switches" between following forward edges and following backward edges [28, p. 12].

In Li's algorithm, the level graph in Dinitz' algorithm is essentially represented by the sets $(S^i)_{i\in[1,\ell]}$ and $(T^i)_{i\in[1,\ell]}$, which are defined as described in Section 4.3.2. Briefly, they can be described as the sources and sinks that lie at a distance $2i-1$ and $2i-2$, respectively, from S in the graph H.

It is not straightforward to emulate H using G, as there will be "interference" between the forward and backward segments, see [28, p. 12–13] and also [28, p. 33]. Li solves this problem by introducing the concept of a *handle* (see Definition 4.5) in a graph, which is simply a copy of all the vertices and edges in the paths between some subset of the sources $S' \subseteq S$ and some subset of the sinks $T' \subseteq T$. See Definition 4.5. Such handles are added to represent each of the backward shortcut edges, before finding the blocking flow $\mathbf{g}$.

4.1 Solving the Simpler LP Approximatively

We can now solve the LP (3.41)–(3.45) approximatively, by iteratively fixing the variable γ to each value in a finite set Γ, and solving the problem (3.46)–(3.52), with this γ. We explain this in detail in Section 4.1.4. The set Γ is defined by

$$\Gamma := \left\{ \frac{\phi}{3}, \frac{\phi}{3}(1+\varepsilon), \frac{\phi}{3}(1+\varepsilon)^2, \ldots, \frac{\phi}{3}(1+\varepsilon)^K \right\},$$

where K is the smallest integer such that

$$\frac{\phi}{3}(1+\varepsilon)^K > |\mathbf{a}|_1.$$

As we will see in Theorem 4.6, we obtain a sufficiently good approximate solution by optimising over just $\gamma \in \Gamma$ instead of $\gamma \geq 0$. We start by considering the problem obtained by fixing γ in (3.46)–(3.52), which is a maximum flow problem. We also consider its dual, which is a minimum cut problem.

4.1.1 The Maximum Flow Problem

If γ is fixed, the LP (3.46)–(3.52) can be considered as a network flow problem:

Definition 4.1. For $\gamma \in \Gamma$, we use NFP_γ to denote the following network flow problem: We are given the network $G = (V, E)$ with sources $S \subseteq V$ and sinks $T \subseteq V$ such that

$$\begin{aligned} S \cap T &= \varnothing, \\ (s,t) &\notin E \quad \forall s \in S, t \in T, \\ \delta^-(s) &= \varnothing \quad \forall s \in S, \\ \delta^+(t) &= \varnothing \quad \forall t \in T, \\ S \rightsquigarrow v \text{ and } v &\rightsquigarrow T \quad \forall v \in V. \end{aligned}$$

The sources have supplies $\mathbf{a} \in \mathbb{R}^S_{>0}$ and the sinks have capacities $\gamma\mathbf{b}$ where $\gamma \geq 0$ and $\mathbf{b} \in \mathbb{R}^T_{>0}$. We want to find a *valid flow*, which is a vector $\mathbf{f} \in \mathbb{R}^E_{\geq 0}$ satisfying

$$f(\delta^+(s)) \leq a_s \quad \forall s \in S, \tag{4.1}$$

$$f(\delta^-(t)) \leq \gamma b_t \quad \forall t \in T, \tag{4.2}$$

$$f(\delta^+(v)) = f(\delta^-(v)) \quad \forall v \in V \setminus (S \cup T). \tag{4.3}$$

We let $\mathcal{F}_\gamma \subseteq \mathbb{R}^E_{\geq 0}$ denote the set of valid flows for some γ, and define the *value* of a flow $\mathbf{f}$ as

$$\mathrm{val}(\mathbf{f}) := f(\delta^+(S)) = f(\delta^-(T)).$$

Let $\mathbf{f}$ be the maximum flow for some NFP_γ, and let OPT_γ be the value of this flow. Let $r_s := a_s - f(\delta^+(s))$ for each $s \in S$. By (4.1), r_s is nonnegative and thus (3.47) and (3.52) are fulfilled. By (4.2) and (4.3), constraints (3.48) and (3.49), respectively, are fulfilled. The constraints (3.50) and (3.51) are also fulfilled, so γ, $(f_{v,u})_{(v,u)\in E}$ and $(r_s)_{s\in S}$ give a feasible solution to (3.46)–(3.51). The objective value for this solution is

$$\begin{aligned} \gamma + \sum_{s\in S} r_s &= \gamma + \sum_{s\in S} (a_s - f(\delta^+(s))) \\ &= \gamma + |\mathbf{a}|_1 - f(\delta^+(S)) \\ &= \gamma + |\mathbf{a}|_1 - \mathrm{OPT}_\gamma\,. \end{aligned}$$

Thus, by weak duality (see Theorem 2.10), we have proved the following (see also [28, p. 12]):

Lemma 4.2. *The objective value of any solution to* (3.41)–(3.45)*, i.e.,* $\sum_{s\in S} a_s y_s$*, is bounded above by* $\gamma + |\mathbf{a}|_1 - \mathrm{OPT}_\gamma$ *for any* $\gamma \geq 0$*.*

4.1.2 The Minimum Cut

We would like to express the problem NFP_γ on a more standard form, namely as in Definition 2.23. In order to do this we add a new vertex s^* to V, and add edges (s^*, s) for each $s \in S$ to E. Likewise, we add a new vertex t^* and edges (t, t^*) for each $t \in T$. Note that the constraint of conservation of flow, $f(\delta^+(v)) = f(\delta^-(v))$ is present in both NFP_γ and Definition 2.23, for $v \in V \setminus (S \cup T)$ respectively $v \in V \setminus \{s^*, t^*\}$. Because of this, the constraints (4.1) and (4.2) can be expressed as (2.16), if we set the edge capacities to

$$\begin{aligned} c_{s^*,s} &:= a_s \quad \text{for } s \in S, \\ c_{t,t^*} &:= \gamma b_t \quad \text{for } t \in T, \\ c_{v,u} &:= \infty \quad \text{for } v \in V \setminus \{s^*\}, u \in V \setminus \{t^*\}. \end{aligned} \tag{4.4}$$

This gives us

$$f(\delta^+(s)) = f(\delta^-(s)) = f_{s^*,s} \leq c_{s^*,s} = a_s \quad \forall s \in S$$

and

$$f(\delta^-(t)) = f(\delta^+(t)) = f_{t,t^*} \leq c_{t,t^*} = \gamma b_t \quad \forall t \in T,$$

as desired.

By Theorem 2.25, the dual of this problem consists of finding a cut $C = (V_s, V_t)$ such that

$$\begin{aligned} s^* &\in V_s, \\ t^* &\in V_t, \\ V_s \cap V_t &= \varnothing, \\ V &= V_s \cup V_t, \end{aligned}$$

and $\text{cap}(C) = \sum_{e \in E_C} c_e$ (see Definition 2.24 in Section 2.6) is minimised.

We will use the notation

$$T(\tilde{S}) := \{t \in T : \tilde{S} \rightsquigarrow_G t\}$$

for any $\tilde{S} \subseteq S$. Because of (4.4), only edges connected to s^* or t^* can be in the optimal cut set E_C, as otherwise the total capacity would become infinite. Clearly, this is larger than $|\mathbf{a}|_1$, which we get if we simply set $E_C := \{(s^*, s) : s \in S\}$. We can also see that if and only if $(s^*, s) \notin E_C$ for some $s \in S$, then we must have

$$(t, t^*) \in E_C \quad \forall t \in T(\{s\}),$$

in order to separate s^* from t^* with the cut C. It is easy to see this by considering a sketch such as Figure 4.2.

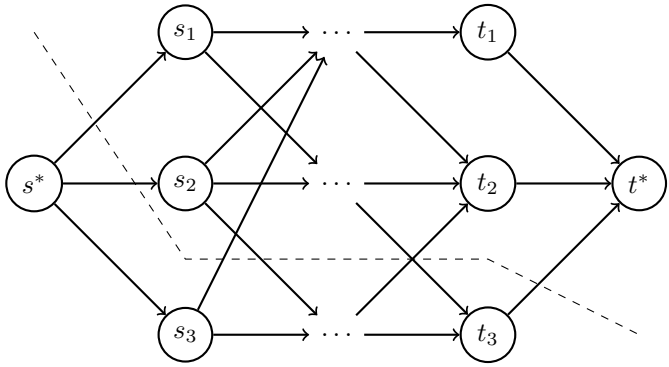


Figure 4.2: A sketch of the definition of s^* and t^*. The dashed line represents a cut with $E_C = \{(s^*, s_1), (s^*, s_2), (t_3, t^*), (s_3, \dots), (\dots, t_2)\}$.

Now we introduce the set

$$S' := S \cap V_s = \{s \in S : (s^*, s) \notin E_C\}.$$

We can then relate the cut set E_C to $S' \subseteq S$ by

$$E_C = \{(s^*, s) \in E : s \in S \setminus S'\} \cup \{(t, t^*) \in E : t \in T(S')\},$$

which gives us the total capacity

$$\begin{aligned} \operatorname{cap}(C) &= \sum_{e \in E_C} c_e \\ &= \sum_{s \in S \setminus S'} c_{s^*,s} + \sum_{t \in T(S')} c_{t,t^*} \\ &= \sum_{s \in S \setminus S'} a_s + \sum_{t \in T(S')} \gamma b_t \\ &= a(S \setminus S') + \gamma b(T(S')) \end{aligned}$$

and by the max-flow min-cut theorem (Theorem 2.25),

$$\mathrm{OPT}_\gamma = \min_{S' \subseteq S} a(S \setminus S') + \gamma b(T(S')).$$

In Figure 4.2 we have $V_s = \{s^*, s_3, \dots, t_3\}$ and $V_t = \{s_1, s_2, \dots, t_1, t_2, t^*\}$, where the ellipsis represents some unspecified sets of vertices. We also have $S' = \{s_3\}$.

4.1.3 The Shortcut Graph and Handled Graphs

In this section we let $G = (V, E)$ be a directed graph and $S \subseteq V$, $T \subseteq V$ such that the properties in Theorem 3.10 hold. We are only really interested in the sets S and T, and would like to ignore the structure of the graph "between" these two sets. Therefore we introduce a new concept:

Definition 4.3 ([28, Definition D.4])**.** Let $G' = (V', E')$ be a graph and $S, T \subseteq V'$ such that Theorem 3.10 holds. Let $\mathbf{f}'$ be a valid flow. We define the *shortcut graph* as $H = (S \cup T, E_H)$ where

$$E_H := \{(s,t) : s \in S, t \in T, s \rightsquigarrow t\} \\ \cup \{(t,s) : s \in S, t \in T, s \rightsquigarrow_{\mathrm{supp}(\mathbf{f}')} t\}.$$

The edges in the first set are called *forward shortcut edges*, and the edges in the second set are called *backward shortcut edges* (w.r.t. $\mathbf{f}'$).

This is a *bipartite graph*: there are two disjoint sets (S and T here) such that every edge connects a node in one set to a node in the other, and there are no edges from a node in S to a node in S, nor from a node in T to a node in T. The shortcut graph for G is too large to create within our bounds on run time, since the number of edges may be quadratic in n.

In order to "emulate" it, a novel data structure called a *handled graph* is used. The idea is to add copies of vertices and edges to the graph G, but keep the same S and T.

Definition 4.4 ([28, Definition D.1])**.** Let $\hat{G} = (\hat{V}, \hat{E})$ be a sub-graph of $G = (V, E)$ and let $S, T \subseteq V$. A directed graph $\tilde{G} = (\tilde{V}, \tilde{E})$ is a *copy of* $\hat{G}$ if

$$\tilde{V} \cap (S \cup T) = \hat{V} \cap (S \cup T),$$
$$(\tilde{V} \setminus (S \cup T)) \cap (\hat{V} \setminus (S \cup T)) = \varnothing,$$

and there is a bijection $\pi \colon \tilde{V} \to \hat{V}$ such that $\pi(v) = v$ for every $v \in \tilde{V} \cap (S \cup T)$ and $(v, u) \in \tilde{E}$ if and only if $(\pi(v), \pi(u)) \in \hat{E}$ for every $v, u \in \tilde{V}$.

Definition 4.5 ([28, Definition D.2])**.** Let $G^1 = (V^1, E^1), G^2 = (V^2, E^2), \ldots, G^k = (V^k, E^k)$ be k copies of sub-graphs of G for some integer $k \geq 0$ such that

$$V^i \cap V^j \subseteq S \cup T \quad \forall i, j \in [1, k].$$

We say that $G' = (V', E')$, where

$$V' = V \cup V^1 \cup V^2 \cup \cdots \cup V^k, E' = E \cup E^1 \cup E^2 \cup \cdots \cup E^k,$$

is a *handled graph*, and $G^1, G^2, \ldots, G^k$ are called the *handles of* G'.

4.1.4 Solution to the Simple Packing LP from Solutions to NFP_γ

Consider the following theorem:

Theorem 4.6 ([28, Theorem 3.10]). *Let $\gamma \geq 0$, $\varepsilon > 0$ and let $G = (V, E)$, $S, T \subseteq V$, $\mathbf{a}, \mathbf{b} \in \mathbb{R}^V_{\geq 0}$ be an instance of NFP_γ. It is possible to find a valid flow $\mathbf{f}$ and a set $S' \subseteq S$ such that*

$$a(S \setminus S') + \frac{\gamma}{1+\varepsilon} b(T(S')) \leq \mathrm{val}(\mathbf{f}) + \frac{\phi}{3}.$$

This can be done in

$$O\Big(\frac{1}{\varepsilon} \cdot |E| \cdot \log |V| \cdot \log \frac{|\mathbf{a}|_1}{\phi}\Big)$$

time.

Given such a solution $\mathbf{f}_\gamma, S'_\gamma$ for each $\gamma \in \Gamma$, we can obtain an approximate solution $\mathbf{y}$ to (3.41)–(3.45) by setting

$$y_v := \sum_{\gamma : S'_\gamma \rightsquigarrow v} z_\gamma \quad \text{for each } v \in V,$$

where $\mathbf{z}$ is the vector that solves (4.5)–(4.8) optimally. In the following LP we define the vectors $\tilde{\mathbf{a}}$ and $\tilde{\mathbf{b}}$ by

$$\begin{aligned} \tilde{a}_\gamma &:= a(S'_\gamma), \\ \tilde{b}_\gamma &:= b(T(S'_\gamma)), \end{aligned}$$

for $\gamma \in \Gamma$:

$$\begin{aligned} \text{maximise} \quad & \tilde{\mathbf{a}}\mathbf{z}, && (4.5) \\ \text{subject to} \quad & |\mathbf{z}|_1 \leq 1, && (4.6) \\ & \frac{1}{(1+\varepsilon)^2} \tilde{\mathbf{b}}\mathbf{z} \leq 1, && (4.7) \\ & \mathbf{z} \in \mathbb{R}^\gamma_{\geq 0}. && (4.8) \end{aligned}$$

To see that this $\mathbf{y}$ is a solution we check a few properties. For any $v \in V$, we have

$$y_v \leq \sum_{\gamma \in \Gamma} z_\gamma = |\mathbf{z}|_1 \leq 1,$$

that is, (3.44). Likewise, $\mathbf{z} \geq \mathbf{0}$ gives (3.45). For any $(v, u) \in E$, $S'_\gamma \rightsquigarrow v$ implies $S'_\gamma \rightsquigarrow u$, so $y_v \leq y_u$, i.e., (3.43) holds. Further, (4.7) gives us

$$\begin{aligned}(1+\varepsilon)^2 \geq \tilde{\mathbf{b}}\mathbf{z} &= \sum_{\gamma\in\Gamma} b(T(S'_\gamma))z_\gamma = \sum_{\gamma\in\Gamma}\sum_{t\in T, S'_\gamma \rightsquigarrow t} b_t z_\gamma \\ &= \sum_{t\in T} b_t y_t = \mathbf{b}\mathbf{y},\end{aligned} \tag{4.9}$$

which is (3.42), relaxed by a factor of $(1+\varepsilon)^2$. By instead using $\varepsilon' = \sqrt{1+\varepsilon}-1$, we get

$$\mathbf{b}\mathbf{y} \leq (1+\varepsilon')^2 = \sqrt{1+\varepsilon}^2 = 1+\varepsilon.$$

Note that $\varepsilon' \geq (\sqrt{2}-1)\varepsilon$, so this change of ε does not affect the expression for running time given by Theorem 4.6.

By a similar rewriting to (4.9), we obtain $\tilde{\mathbf{a}}\mathbf{z} = \mathbf{a}\mathbf{y}$.

Theorem 4.6 gives us

$$a(S \setminus S'_\gamma) + \frac{\gamma}{1+\varepsilon} b(T(S'_\gamma)) \leq \mathrm{val}(\mathbf{f}_\gamma) + \frac{\phi}{3} \leq \mathrm{OPT}_\gamma + \frac{\phi}{3}, \tag{4.10}$$

where OPT_γ is the optimum flow for NFP_γ. Now, by Lemma 4.2, we have

$$\mathbf{a}\mathbf{y}^* \leq \min_{\gamma\geq 0}(|\mathbf{a}|_1 - \mathrm{OPT}_\gamma + \gamma) \leq \min_{\gamma\in\Gamma}(|\mathbf{a}|_1 - \mathrm{OPT}_\gamma + \gamma),$$

where $\mathbf{y}^*$ is the optimal value for (3.41)–(3.45). Combining these two equations gives

$$\begin{aligned}\mathbf{a}\mathbf{y}^* &\leq \min_{\gamma\in\Gamma}\left(|\mathbf{a}|_1 - a(S\setminus S'_\gamma) - \frac{\gamma}{1+\varepsilon} b(T(S'_\gamma)) + \gamma\right) + \frac{\phi}{3} \\ &= \min_{\gamma\in\Gamma}\left(a(S'_\gamma) - \frac{\gamma}{1+\varepsilon} b(T(S'_\gamma)) + \gamma\right) + \frac{\phi}{3}.\end{aligned}$$

Further, one can prove [28, Lemma B.2, p. 23], that

$$\tilde{\mathbf{a}}\mathbf{z} \geq \min_{\gamma\in\Gamma}\left(a(S'_\gamma) - \frac{\gamma}{1+\varepsilon} b(T(S'_\gamma)) + \gamma\right) - \frac{2\phi}{3}.$$

This finally gives

$$\begin{aligned}\mathbf{a}\mathbf{y} &= \tilde{\mathbf{a}}\mathbf{z} \\ &\geq \min_{\gamma\in\Gamma}\left(a(S'_\gamma) - \frac{\gamma}{1+\varepsilon} b(T(S'_\gamma)) + \gamma\right) - \frac{2\phi}{3} \\ &\geq \mathbf{a}\mathbf{y}^* - \frac{\phi}{3} - \frac{2\phi}{3} \\ &= \mathbf{a}\mathbf{y}^* - \phi.\end{aligned}$$

In other words, we have proved the following:

Theorem 4.7 ([28, Theorem 3.2]). *Let $G = (V, E)$ be a directed acyclic graph and $\mathcal{Q} := \{\mathbf{y} \in [0,1]^V : y_v \leq y_u \ \forall (v,u) \in E\}$. Let $\mathbf{a}, \mathbf{b} \in \mathbb{R}^V_{\geq 0}$. Let $\mathbf{y}^* \in \mathcal{Q}$ satisfy $\mathbf{by}^* \leq 1$ and be such that $\mathbf{ay}^*$ is maximised. Let $\varepsilon \in (0,1), \phi \in (0, |\mathbf{a}|_1/2)$. Then in $\tilde{O}_\varepsilon\left(|E| \cdot \log^2 \frac{|\mathbf{a}|_1}{2}\right)$ time, we can find $\mathbf{y} \in \mathcal{Q}$ satisfying $\mathbf{by} \leq 1 + \varepsilon$ and $\mathbf{ay} \geq \mathbf{ay}^* - \phi$.*

The code for actually performing this step is very simple, as seen in Algorithm 9. Note that this calls the function `brute_force_lp`. Because the LP

Algorithm 9 For finding a solution $\mathbf{y}$ from solutions to $(\mathrm{NFP}_\gamma)_{\gamma \in \Gamma}$.

```
function y_from_nfp_gamma(G::Graph, ã::Vector, b̃::Vector, S′::Vector, ϵ)
    Γ_length = length(S′)

    A = [ones(1, Γ_length) ; reshape(b̃, 1, :)]
    b = [one(ϵ), (1+ϵ)^2]
    c = ã
    (z_value,_) = brute_force_lp(A, b, c)

    # this sets y[v] = sum(z[γ] where (v reachable from S′[γ]))
    y = zeros(typeof(ϵ), G.n)
    for i in 1:Γ_length
        for v in reach_from(G, S′[i])
            y[v] += z_value[i]
        end
    end
    y
end
```

(4.5)–(4.8) only has two non-trivial constraints, it can be solved by simply iterating over all of the vertices of the set of allowed $\mathbf{z}$, as described in Section 2.4.2. See Algorithm 10 for an implementation of this, for the LP:

$$\text{maximise } \mathbf{c}^T\mathbf{x} \text{ subject to } \mathbf{Ax} \leq \mathbf{b} \text{ and } \mathbf{x} \geq \mathbf{0}, \tag{4.11}$$

where $\mathbf{A} \in \mathbb{R}^{2 \times n}$, $\mathbf{b} \in \mathbb{R}^2$ and $\mathbf{c} \in \mathbb{R}^n$.

4.2 Solving NFP_γ

Here we give an overview of the proof of Theorem 4.6. Many of the details are skipped, in order to keep the description short.

We note that the concept of a shortcut graph (previously defined in Definition 4.3), can be extended to the case when G' is a handled graph and $\mathbf{f}'$ is a valid flow for G'.

Definition 4.8. Let $G' = (V', E')$ be a handled graph, $\mathbf{f}'$ be a valid flow for G', and H be the shortcut graph for $\mathbf{f}'$. Let $\mathbf{a} > \mathbf{0}$ and $\mathbf{b} > \mathbf{0}$ be vectors over S

and T, respectively. Let $\gamma \geq 0$. A source $s \in S$ is said to be *satisfied* (w.r.t. $\mathbf{f}'$) if $f(\delta^+(s)) = a_s$, and *unsatisfied* otherwise. Likewise, a sink $t \in T$ is said to be *saturated* (w.r.t. $\mathbf{f}'$) if $f(\delta^-(t) = \gamma b_t$, and *unsaturated* otherwise.

An *alternating shortcut path* in H is defined as a path which starts at an unsatisfied source $s \in S$. An *augmenting shortcut path* in H is defined as an alternating shortcut path in H which ends at an unsaturated sink $t \in T$.

The following lemma from [28, Section D.2] states that if all the augmenting paths in the shortcut graph for a flow $\mathbf{f}'$ are longer than some certain length, then we can find a good cut set S'.

Theorem 4.9 ([28, Lemma D.6])**.** *Let $G' = (V', E')$ be a handled graph and $\mathbf{f}'$ be a valid flow. Let*

$$L = \left\lfloor \log_{1+\varepsilon} \frac{3|\mathbf{a}|_1}{\phi} \right\rfloor = \tilde{O}_\varepsilon\Big(\log \frac{|\mathbf{a}|_1}{\phi} \Big).$$

Assume the shortcut graph H for $\mathbf{f}'$ does not contain an augmenting path of length at most $2L+1$. Then in time $O_\varepsilon(|E'|)$ we can find a set $S' \subseteq S$ such that

$$a(S \setminus S') + \frac{\gamma}{1+\varepsilon} b(T(S')) \leq \mathrm{val}(\mathbf{f}') + \frac{\phi}{3}.$$

For a proof of this lemma, see [28, p. 30]. So, now we simply need to find such a G' and $\mathbf{f}'$. This is handled by Algorithm 11 [28, Algorithm 4, p. 34]. This algorithm simply starts with the graph G and zero flow, and repeatedly calls the subroutine `inc_len` (see Algorithm 15). This subroutine takes a handled graph, which we call $G^{(\ell)}$, and a flow which we call $\mathbf{f}^{(\ell)}$. The shortest augmenting path in the shortcut graph for $G^{(\ell)}$ and $\mathbf{f}^{(\ell)}$ must have length at least $2\ell + 1$. It returns a new handled graph $G^{(\ell+1)}$ and a new flow $\mathbf{f}^{(\ell+1)}$, such that the shortest alternating shortcut path in this new graph, has length at least $2\ell + 3$.

Clearly the shortest augmenting shortcut path for $G^{(0)} = G$ and $\mathbf{f}^{(0)} = \mathbf{0}$ has length at least 1. Thus, after $L+1$ iterations, $G^{(L)}$ and $\mathbf{f}^{(L)}$ fulfill the conditions of Theorem 4.9. In Section 4.3, we will explain some of the implementation details of Algorithm 15).

4.3 Increasing the Length of the Shortest Alternating Shortcut Path

Here we give a quick overview of the details of the subroutine `inc_len`. It uses a few utility functions, among which are a function for finding particular sets of sources and sinks, as well as a function for projecting a flow to a (copy of a) sub-graph. Further, it uses a function for finding a blocking flow in a graph.

4.3.1 Projection of a Flow

We start by defining a projection of a flow:

Definition 4.10 ([28, Definition D.10])**.** Let $G = (V, E)$ be a handled graph and $\mathbf{f}$ a valid flow for G. Let $\tilde{G} = (\tilde{V}, \tilde{E})$ be a sub-graph or a copy of a sub-graph of G. If for every edge $(v, u) \in \operatorname{supp}(\mathbf{f})$, we have some $(\tilde{v}, \tilde{u}) \in \tilde{E}$ with $\pi(v) = \pi(\tilde{v})$ and $\pi(u) = \pi(\tilde{u})$, we say that $\mathbf{f}$ can be *projected* to $\tilde{G}$. The *projection* is defined as the flow $\tilde{\mathbf{f}}$ with

$$\tilde{f}_{\tilde{v},\tilde{u}} := f\big(\{(v, u) \in E : \pi(v) = \pi(\tilde{v}), \pi(u) = \pi(\tilde{u})\}\big).$$

Note that $\tilde{\mathbf{f}}$ is a valid flow for $\tilde{G}$, and that $\operatorname{val}(\tilde{\mathbf{f}}) = \operatorname{val}(\mathbf{f})$.

See Figure 4.3 for a sketch of a projection of a flow $\mathbf{f}$ to the sub-graph $\tilde{G} = (\{s, \tilde{v}, \tilde{u}, t\}, \tilde{E})$ of $G = (\{s, v, u, v', u', \tilde{v}, \tilde{u}, t\}, E)$ with $S = \{s\}$ and $T = \{t\}$, where

$$\pi(\tilde{v}) = \pi(v') = \pi(v) = v,$$
$$\pi(\tilde{u}) = \pi(u') = \pi(u) = u.$$

We see that this flow must be valid, because the total flow out of s is the same for $\mathbf{f}$ and $\tilde{\mathbf{f}}$, and likewise the total flow into t is unchanged. This holds for all vertices in S and T. Note in this case that the sets S and T consist of a single element each, but the preceding statement is true in general. This also shows us that the value, which is simply defined as the sum of flow out of all vertices in S, is the same for $\mathbf{f}$ and $\tilde{\mathbf{f}}$.

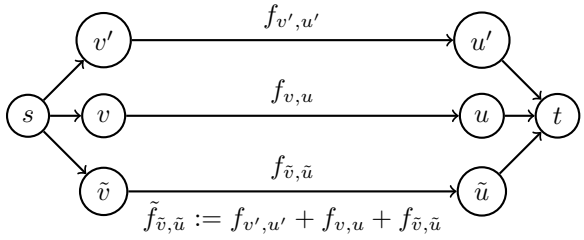


Figure 4.3: Sketch of a projection of a flow to a sub-graph.

4.3.2 Finding Certain Sets of Sources and Sinks

In Algorithm 15 we also use the sets $S^i \subseteq S$, which are defined as the sources to which the shortest alternating shortcut path in H has length $2i - 2$ for all

$i \in [1, \ell]$, and the sets $T^i \subseteq T$, which are defined as the sinks to which the shortest alternating shortcut path in H has length $2i - 1$, for all $i \in [1, \ell]$. The set $S^{\ell+1}$ is defined as the sources to which there is no alternating shortcut path of length at most $2\ell - 2$, i.e.,

$$S^{\ell+1} := S \setminus \bigcup_{i \in [1,\ell]} S^i.$$

Likewise, $T^{\ell+1}$ is defined as the sinks to which there is no alternating shortcut path of length at most $2\ell - 1$, i.e.,

$$T^{\ell+1} := T \setminus \bigcup_{i \in [1,\ell]} T^i.$$

Here H is the shortcut graph for the graph G° and the flow $\mathbf{f}^\circ$ which are passed as parameters to Algorithm 15. The procedure we use for finding such S^i and T^i is Algorithm 12.

It works by constructing a graph, which we call F, containing two vertices for each vertex of G°. F has an edge for each edge in G°, as well as an edge for each edge in $\text{supp}(\mathbf{f}^\circ)$. A sketch of this graph is found in Figure 4.4. In this figure, any $\tilde{v}$ corresponds to the vertex $v \in V(G^\circ)$.

We define the sets $\tilde{S} := \{\tilde{s} : s \in S\}$ and $\tilde{T} := \{\tilde{t} : t \in T\}$. The edges of F are (v, u) for $(v, u) \in E(G^\circ)$, and $(\tilde{u}, \tilde{v})$ for $(v, u) \in E(\text{supp}(\mathbf{f}^\circ))$. We also define a weight vector $\mathbf{w}$ by setting

$$\begin{aligned} w_{t,\tilde{t}} &:= 1 \quad \forall t \in T, \\ w_{\tilde{s},s} &:= 1 \quad \forall s \in S, \\ w_{v,u} &:= 0 \quad \forall (v, u) \in E(G^\circ), \\ w_{\tilde{u},\tilde{v}} &:= 0 \quad \forall (v, u) \in E(\text{supp}(\mathbf{f}^\circ)), \end{aligned}$$

i.e., the weight is one if and only if the edge is between $\tilde{S}$ and S, or between T and $\tilde{T}$.

When we traverse a path from $\tilde{S}$ to T in F, starting with an edge with weight one from $\tilde{S}$ to S, it corresponds to following a forward shortcut edge in H (i.e., a path in G°). Likewise, when we traverse a path from T to $\tilde{S}$, starting with an edge of weight one from T to $\tilde{T}$, it corresponds to following a backward shortcut edge in H (i.e., a backwards path in $\text{supp}(\mathbf{f}^\circ)$). In other words, for any path in F the number of weight-one edges, or equivalently, the sum of the weights of the edges, is equal to the length of the corresponding path in H. We can call this sum the *distance* of the path in F.

If we know the minimum distance of some $s \in S$ from the set of unsaturated sources (i.e., $\{s \in S : \delta^+(s) < a_s\}$), then we know which S^i this source belongs

to. Likewise, if we know the minimum distance of some $t \in T$ from the set of unsaturated sources, we know which T^i it belongs to. Algorithm 12 finds this minimum weight and stores it in the variable `distance`. Then we simply set

$$S^i = \{s \in S : \texttt{distance}[s] = 2i - 2\}$$

and

$$T^i = \{t \in T : \texttt{distance}[t] = 2i - 1\}.$$

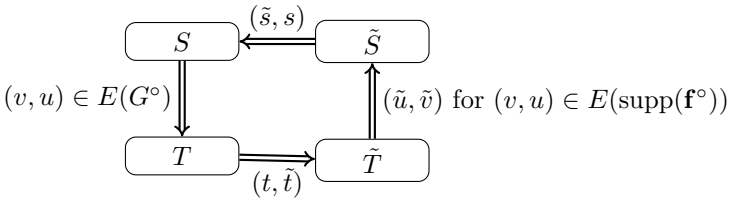


Figure 4.4: The graph F, used for finding S^i and T^i.

4.3.3 Finding a Subflow Sent by Sources or Received by Sinks

We also need to find a subflow, defined in Definition 4.11, which is sent by some subset of sources S', or alternatively a subflow that is received by some subset of sinks T'.

Definition 4.11 ([28, Definition D.7])**.** Let $G' = (V', E')$ be a directed graph and let $S \subseteq V'$ and $T \subseteq V'$ be sets of sources and sinks, respectively. Let $\mathbf{f}'$ be a valid flow. Let $S' \subseteq S$. Then, a *subflow of* $\mathbf{f}'$ *sent by* S' is a flow $\mathbf{f}''$ satisfying

$$\begin{aligned} f''_e &\leq f'_e \quad \forall e \in E, \\ f''_e &= f'_e \quad \forall e \in \delta^+_{G'}(S'), \\ f''_e &= 0 \quad \forall e \in \delta^+_{G'}(S \setminus S'). \end{aligned}$$

Likewise, a *subflow of* $\mathbf{f}$ *received by* T' is a flow $\mathbf{f}'$ satisfying

$$\begin{aligned} f''_e &\leq f'_e \quad \forall e \in E, \\ f''_e &= f'_e \quad \forall e \in \delta^-_{G'}(T'), \\ f''_e &= 0 \quad \forall e \in \delta^-_{G'}(T \setminus T'). \end{aligned}$$

Our procedure for finding such a subflow sent by S' is given in Algorithm 13. The procedure for finding a subflow received by T', which we call `find_flow_to_T`, is implemented very similarly. The major difference is that `supp_out` is replaced by `supp_inc`. Note that the procedure makes use of a vector $\mathbf{c} \in \mathbb{R}^V_{\geq 0}$, which is initialised to all zeros. The reason that this is a parameter and not a local variable, is simply that it is too large to construct within the run time bound of the subroutine `find_flow_from_S`, which is

$$O(|S'| + |V_{\text{supp}(\mathbf{f}'')}| \cdot \log |V'| + |E_{\text{supp}(\mathbf{f}'')}|).$$

For some vertex $v \in V'$, c_v represents the amount of flow in $\mathbf{f}''$ that reaches this node. This will only be updated for vertices in $V(\text{supp}(\mathbf{f}''))$, and remain zero for all other vertices. At the end of the procedure, it is reset to zero for these vertices.

4.3.4 Finding a Blocking Flow

A fast algorithm for finding a blocking flow was presented by Sleator and Tarjan in 1983 [34, p. 388]. This description is repeated in Algorithm 18 in Appendix A.2. Our implementation of this algorithm is found in Algorithm 19, also in Appendix A.2. Here we give a brief high-level overview of the algorithm, starting with the definition of a blocking flow.

Definition 4.12. Let $R = (V_R, E_R)$ be a directed graph. Let $\mathbf{c} \in [0, \infty]^{E_R}$ be a vector of capacities. Assume we have a source $s \in V_R$ and a sink $t \in V_R$, such that $\delta^-(s) = \delta^+(t) = \varnothing$. A *valid flow* is a flow $\mathbf{g}$ such that

$$\begin{aligned} g(\delta^+(v)) = g(\delta^-(v)) \quad & \forall v \in V_R \setminus \{s, t\}, \\ g_e \leq c_e \quad & \forall e \in E_R, \end{aligned}$$

i.e., the flow is conserved and the capacities of all edges are satisfied. A *blocking flow* is a valid flow $\mathbf{g}$ with the additional condition that every path from s to t in R contains an edge e for which $g_e = c_e$.

A blocking flow is thus a flow where it is not possible to increase the flow in any path from the source to the sink.

The algorithm makes use of a data structure called a *link-cut tree* or *dynamic tree*. In essence, the algorithm in question performs a depth-first search (DFS) from s to t, linking the link-cut tree nodes with all edges we walk along. If this DFS succeeds we have an augmenting path. Then we put the largest flow we can on this path, and decrease the capacities by this flow. If the DFS instead reaches a node where all the outgoing edges have capacity zero, then we remove

from the graph all edges leading to that node, to make sure we do not end up there again. Then we continue the search, cascading the removal of edges as necessary.

We repeat this until there are no more augmenting paths from s to t, i.e. all outgoing edges from s have been removed from the graph.

4.3.5 Implementation of Algorithm 15

The subroutine is implemented as seen in Algorithm 15. This implementation is based on the description of the algorithm in pseudocode from [28, Algorithm 3, p. 32], which is repeated in Algorithm 14 for convenience.

Lines 1–13 of Algorithm 15 correspond to the lines 1–7 from Algorithm 14. Here we construct the sets S^i and T^i for $i \in [1, \ell + 1]$, the sub-graphs $G^{i,+}$ for $i \in [1, \ell + 1]$ and the copies of sub-graphs $G^{i,-}$ for $i \in [1, \ell]$. The sets S^i and T^i are stored simply as vectors, but the sub-graphs are not stored as `Graph` objects. Instead, they are stored as pairs of vertices and their adjacency lists $(v, \delta^+(v))$. The handled graph G' is also constructed, by adding new vertices for all vertices in $V(G^{i,-})$ for $i \in [1, \ell]$, and defining the projection mappings π' and π'_{edge} appropriately. The mapping $\pi'_{\text{edge}} \colon E(G') \to E(G)$ is

$$\pi'_{\text{edge}} \colon (v, u) \mapsto (\pi'(v), \pi'(u)),$$

i.e., it is a mapping from pairs of vertices to pairs of vertices. However, in adjacency lists of graphs in the implementation, we represent edges by ids (integers), instead of as pairs of vertices. Thus we need to keep track of π'_{edge}, which we store as a mapping from edge ids to edge ids, in addition to π'.

Lines 14–41 correspond to the lines 8–15 from Algorithm 14. In each iteration here, we find a subflow $\mathbf{f}^{\circ(i,+)}$ of $\mathbf{f}^\circ$ sent by S^i, subtract it from $\mathbf{f}^\circ$, and set $\mathbf{f}'^{(i,+)}$ to the projection of the subflow to $G^{(i,+)}$. We also find a subflow $\mathbf{f}^{\circ(i,-)}$ of $\mathbf{f}^\circ$ received by T^i, subtract it from $\mathbf{f}^\circ$, and set $\mathbf{f}^{\circ(i,-)}$ to the projection of the subflow to $G^{(i,-)}$. Because the subflow $\mathbf{f}^{(i,+)}$ is equal to $\mathbf{f}^\circ$ on each source in S^i, this in essence "partitions" $\mathbf{f}^\circ$ into $(\mathbf{f}^{\circ(i,+)})_{i \in [1,\ell+1]}$ and $(\mathbf{f}^{\circ(i,-)})_{i \in [1,\ell]}$. Then we define the flow $\mathbf{f}'$ as the sum of all these subflows $\mathbf{f}'^{(i,+)}$ and $\mathbf{f}'^{(i,-)}$, treating missing edges as having flow zero.

Finally, lines 42–61 correspond to the lines 16–20 from Algorithm 14. Here we simply define the graph R by adding edges (v, u) for each $(v, u) \in E(G^{i,+})$ for $i \in [1, \ell + 1]$ and edges (u, v) for each $(v, u) \in E(G^{i,-})$ for $i \in [1, \ell]$. Then we find a blocking flow $\mathbf{g}$ in R and add it to, or subtract it from (depending on the type of edge), the flow $\mathbf{f}'$.

Algorithm 10 Solving the LP (4.11) by iterating over all vertices.

```
function brute_force_lp(A::Matrix, b::Vector, c::Vector)
    n = length(c)
    @assert length(b) == 2 "can only handle two constraints"
    @assert size(A) == (2,n) "invalid size for A"
    A_2 = [A  I]
    c_2 = [c ; 0 ; 0]
    obj_star = convert(eltype(c), -Inf)
    kl_star = nothing
    x_star = nothing
    for k in 1:n+2, l in k+1:n+2
        A_kl = A_2[:,[k,l]]
        if det(A_kl) == 0
            # If the determinant is zero then x[1],x[2]
            # cannot be a basic solution, as explained
            # section 2.4.2 of the report.
            # Therefore we can skip this case w.l.o.g.
            continue
        end
        x = A_kl \ b
        if x[1] < 0 || x[2] < 0
            # do not allow negative solutions
            continue
        end
        obj = c_2[k]*x[1] + c_2[l]*x[2]
        if obj > obj_star
            obj_star = obj
            kl_star = [k,l]
            x_star = x
        end
    end
    if x_star === nothing
        return nothing
    end
    x = zeros(eltype(c), n)
    if kl_star[1] <= n
        x[kl_star[1]] = x_star[1]
    end
    if kl_star[2] <= n
        x[kl_star[2]] = x_star[2]
    end
    @assert A * x <= b .+ FLOAT_TOLERANCE "not feasible!"
    @assert abs(c' * x - obj_star) <= FLOAT_TOLERANCE "wrong objective!"
    (x,obj_star)
end
```

Algorithm 11 Constructing S' and $\mathbf{f}'$ as in Theorem 4.9.

```
function find_S′_and_f(G::Graph, a::Vector, b::Vector, S, T, γ, ϵ, ϕ)
    Gl = G
    πl = collect(1:G.n) # Identity mapping of vertices
    π_edgel = collect(1:G.m) # Identity mapping of edges
    fl = zeros(eltype(a), Gl.m)
    L = floor(Int, log(1+ϵ, 3*norm(a, 1)/ϕ))

    topo_order = topological_ordering(G)
    ρ = ordering_to_rank(topo_order)

    for ℓ = 0:L
        (Gl,πl,π_edgel,fl) = inc_len(ℓ, G, Gl, πl, π_edgel, a, b, S, T, γ, fl, ρ)
    end
    f′_list = [(π_edgel[e], warn_if_negative(flow)) for (e,flow) in enumerate(fl)]
    S′ = find_S′(Gl, fl, a, b, S, T, γ, ϵ, ϕ)

    # convert flow list to flow vector
    f = zeros(eltype(a), G.m)
    for (e,flow) in f′_list
        f[e] += flow
    end
    (S′,f)
end
```

Algorithm 12 Finding S^i and T^i, that are used in `inc_len`.

```
function find_Si_and_Ti(L, G′::Graph, a::Vector, S::Vector, T::Vector, f′::Vector)
    distance = find_distance_from_S(L, G′, a, S, T, f′)

    Si = [Int[] for i=1:L+2]
    Ti = [Int[] for i=1:L+2]

    for s in S
        d = distance[G′.n+s]
        if d <= 2*L
            @assert d >= 0
            @assert rem(d, 2) == 0
            i = div(d, 2)
        else
            i = L + 1
        end
        push!(Si[i+1], s)
    end

    for t in T
        d = distance[t]
        if d <= 2*L+1
            @assert d >= 1
            @assert rem(d, 2) == 1
            i = div(d - 1, 2)
        else
            i = L + 1
        end
        push!(Ti[i+1], t)
    end

    (Si,Ti)
end
```

Algorithm 13 Finding a subflow sent by $S' \subseteq S$.

```
function find_flow_from_S(G′::Graph, π′::Vector, f′::Vector, supp_out, S′::Vector, ρ::
    Vector, c::Vector)
    # we assume that x == 0 for x in c

    # store flow as tuples (edge,flow)
    f″_list = Tuple{Int,eltype(f′)}[]
    # store as tuples (priority,value)
    H = BinaryHeap{Tuple{Int,Int}}(Base.By(first))
    E″ = Int[]

    for s in S′
        for e in supp_out[s] # δ_out(supp(f′), s)
            c[s] += f′[e]
        end
        if c[s] > 0
            push!(H, (ρ[π′[s]],s))
        end
    end

    while !isempty(H)
        (_,v) = pop!(H)
        push!(E″, v)
        a = c[v]
        for e in supp_out[v] # δ_out(supp(f′), v)
            a′ = min(a, f′[e])
            push!(f″_list, (e, a′))
            u = dst_vertex(G′, e)
            if c[u] <= 0 # more robust than c[u] == 0
                push!(H, (ρ[π′[u]],u))
            end
            c[u] += a′
            a -= a′
            if a <= 0
                break
            end
        end
    end

    for v in E″
        c[v] = zero(eltype(c))
    end
    (f″_list,E″)
end
```

Algorithm 14 The procedure `inc_len`.

Input: A handled graph G° and a valid flow $\mathbf{f}^\circ$, defining the shortcut graph H°. G° and $\mathbf{f}^\circ$ should be such that the shortest augmenting path in H° has length at least $2\ell + 1$.

Output: A handled graph G' and a valid flow $\mathbf{f}'$ such that the shortest augmenting path in the shortcut graph H' (defined by G' and $\mathbf{f}'$) has length at least $2\ell + 3$.

$S^i \leftarrow$ the sources to which the shortest alternating shortcut path in H° has length $2i - 2$, $\forall i \in [1, \ell]$

$S^{\ell+1} \leftarrow$ the sources for which there is no alternating shortcut path of length at most $2\ell - 2$

$T^i \leftarrow$ the sinks to which the shortest alternating shortcut path in H° has length $2i - 1$, $\forall i \in [1, \ell]$

$T^{\ell+1} \leftarrow$ the sinks for which there is no alternating shortcut path of length at most $2\ell - 1$

$G^{i,+} \leftarrow$ the sub-graph of G between S^i and T^i, $\forall i \in [1, \ell + 1]$

$G^{i,-} \leftarrow$ a copy of the sub-graph of G between S^{i+1} and T^i, $\forall i \in [1, \ell]$

$G' = (V', E') \leftarrow$ the handled graph with handles $\{G^{i,-} : i \in [1, \ell]\}$

for i = 1,...,ℓ+1 **do**

  find a subflow $\mathbf{f}^{\circ(i,+)}$ of $\mathbf{f}^\circ$ sent by S^i, and set $\mathbf{f}^\circ \leftarrow \mathbf{f}^\circ - \mathbf{f}^{\circ(i,+)}$

  $\mathbf{f}'^{i,+} \leftarrow$ the projection of $\mathbf{f}^{\circ(i,+)}$ to $G^{i,+}$

  if $i = \ell + 1$ **then** break

  find a subflow $\mathbf{f}^{\circ(i,-)}$ of $\mathbf{f}^\circ$ received by T^i, and set $\mathbf{f}^\circ \leftarrow \mathbf{f}^\circ - \mathbf{f}^{\circ(i,-)}$

  $\mathbf{f}'^{(i,-)} \leftarrow$ the projection of $\mathbf{f}^{\circ(i,-)}$ to $G^{i,-}$

end for

Define $\mathbf{f}' := \sum_{i=1}^{\ell+1} \mathbf{f}'^{(i,+)} + \sum_{i=1}^{\ell} \mathbf{f}'^{(i,-)}$

Define $R := (V' \cup \{s^*, t^*\}, E_R)$ and $\mathbf{c} \in [0, \infty]^{E_R}$ where E_R and $\mathbf{c}$ are constructed by:

- for every $i \in [1, \ell + 1]$ and edge e in $G^{i,+}$ add e to E_R and $c_e \leftarrow \infty$
- for every $i \in [1, \ell]$ and edge (v, u) in $G^{i,-}$ add (u, v) to E_R and $c_{u,v} \leftarrow f'_{v,u}$
- for every $s \in S^1$ add (s^*, s) to E_R and $c_{s^*,s} \leftarrow a_s - f'(\delta^+(s))$
- for every $t \in T^{\ell+1}$ add (t^*, t) to E_R and $c_{t^*,t} \leftarrow \gamma b_t - f'(\delta^-(t))$

find a blocking flow $\mathbf{g}$ for $(R, \mathbf{c})$ using Algorithm 19

for every edge e in $G^{i,+}$, for some i, **do** $f'_e \leftarrow f'_e + g_e$

for every edge (v, u) in $G^{i,-}$, for some i, **do** $f'_{u,v} \leftarrow f'_{u,v} - g_{v,u}$

return $(G', \mathbf{f}')$

Algorithm 15 Our implementation of `inc_len`.

```
function inc_len(l, G::Graph, G°::Graph, π°, π°_edge, a::Vector, b::Vector, S, T, γ, f°
    ::Vector, ρ)
    (Si,Ti) = find_Si_and_Ti(l-1, G°, a, S, T, f°)
    (adj_Gi_plus,adj_Gi_minus) = find_Gi(G, Si, Ti, l)
    # adjacency lists for supp(f°)
    supp_out = [Int[] for v in 1:G°.n]
    supp_inc = [Int[] for v in 1:G°.n]
    for e in 1:G°.m
        if f°[e] > 0
            push!(supp_out[src_vertex(G°, e)], e)
            push!(supp_inc[dst_vertex(G°, e)], e)
        end
    end
    (G′,π′,π′_edge,π′_edge_inv) = construct_G′(l, G, a, b, adj_Gi_minus)
    f′_plus_list = Vector{Vector{Tuple{Int,R}}}(undef, l+1)
    f′_minus_list = Vector{Vector{Tuple{Int,R}}}(undef, l)
    scratch_space = zeros(R, G°.n)
    for i in 1:l+1
        # f_list_plus is f°(i,+) in the paper
        (f_list_plus,visited_V_plus) = find_flow_from_S(G°, π°, f°, supp_out, Si[i], ρ,
      scratch_space)
        for (e,flow) in f_list_plus
            f°[e] = warn_if_negative(f°[e] - flow)
        end
        update_support!(f°, visited_V_plus, supp_out, supp_inc)
        f′_plus_list[i] = [(π°_edge[e°], warn_if_negative(flow)) for (e°,flow) in
   f_list_plus]
        if i >= l+1
            break # skip second part on the last iteration
        end
        (f_list_minus,visited_V_minus) = find_flow_to_T(G°, π°, f°, supp_inc, Ti[i], ρ,
    scratch_space)
        for (e,flow) in f_list_minus
            f°[e] = warn_if_negative(f°[e] - flow)
        end
        update_support!(f°, visited_V_minus, supp_out, supp_inc)
        f′_minus_with_dups = [(π°_edge[e°], warn_if_negative(flow)) for (e°,flow) in
   f_list_minus]
        f′_minus_list[i] = [(e, flow) for (e, flow) in f′_minus_with_dups
            if !(abs(flow) < EXPRESSION_TOL && count(x -> x[1] == i, π′_edge_inv[e]) !=
    1)]
    end
    f′ = sum_of_subflows(l, G′.m, f′_plus_list, f′_minus_list, π′_edge_inv)
    (R_graph,C,edge_in_R_graph,s_star,t_star) = construct_R_graph(
        l, G, G′, f′, a, b, γ, adj_Gi_plus, adj_Gi_minus, Si, Ti, π′_edge_inv)
    g = find_blocking_flow(R_graph, C, s_star, t_star)

    for i in 1:l+1
        for (_,out) in adj_Gi_plus[i], e in out
            f′[e] = f′[e] + g[edge_in_R_graph[e]]
        end
    end
    for i in 1:l
        for (_,out) in adj_Gi_minus[i], e in out
            e′ = get_edge_inv(π′_edge_inv, e, i)
            f′[e′] = warn_if_negative(f′[e′] -  g[edge_in_R_graph[e′]])
        end
    end
    # clean up near-zero floats (including small positive values)
    for e in 1:G′.m
        if f′[e] < VARIABLE_TOL
            f′[e] = zero(R)
        end
    end
    (G′,π′,π′_edge,f′)
end
```

Chapter 5

Results

The implementation of the algorithm was evaluated on a set of randomly generated instances. In this chapter we describe how the accuracy and performance of the algorithm were measured, and present the resulting data which we gathered. We also describe how the instances used for the evaluation were generated.

5.1 Instances

The instances used in our evaluation are quite simple and were generated by the simple random method described in Algorithm 16. We have three different categories of instances, characterised by the number of precedence constraints relative to the number of jobs. We name these categories L for "large" instances with $\kappa = \lfloor n^{1.5} \rfloor$, M for "medium" instances with $\kappa = n$, and S for "small" instances with $\kappa = \lfloor n^{0.5} \rfloor$.

The number of jobs used for our instances were $n = 2^k$ for $k = 3, \ldots, 10$, i.e., $n = 8, 16, 32, 64, 128, 256, 512, 1024$.

The job weights $(w_j)_{j \in J}$ were sampled from the uniform distribution with values between 0 and n, and the processing times $(p_j)_{j \in J}$ were sampled from the uniform distribution with values between 1 and n. In Section 6.1.2 we discuss evaluating the implementation on instances where w_j or p_j are sampled from nonuniform distributions.

The names of the instances consist of four fields, the first being the category, L, M or S. The second is n, and the third is κ. The fourth field is a number to identify each instance among those with identical n and κ. For example, the four instances in category L with $n = 8$ and $\kappa = 22$ are named L_00008_22_01 to L_00008_22_04.

Algorithm 16 Generating instances for the evaluation.

```
function generate_instance(n, κ, w_domain, p_domain, m, name)
    job_order = Random.randperm(n)

    max_κ = div(n*(n-1), 2)
    has_edge = Random.shuffle([trues(κ) ; falses(max_κ-κ)])
    @assert length(has_edge) == max_κ
    @assert sum(has_edge) == κ

    edges = NTuple{2,Int}[]
    k = 1
    for i in 1:n, j in i+1:n
        if has_edge[k]
            push!(edges, (job_order[i],job_order[j]))
        end
        k += 1
    end
    @assert length(edges) == κ

    w = Random.rand(w_domain, n)
    p = Random.rand(p_domain, n)

    return Instance(p, w, edges, m, name)
end
```

5.2 Main Results

We focus on evaluating the run time of the algorithm, as well as the objective value. There are a few different objective values to consider. For a given instance, we obtain from our implementation both the fractional weighted completion time

$$\mathrm{WCT}_{\mathrm{frac}} := \sum_{j \in J} w_j C_j,$$

where C_j are fractional completion times, i.e., the input to Algorithm 5 as described in Section 3.3, and the (integer) weighted completion time

$$\mathrm{WCT}_{\mathrm{round}} := \sum_{j \in J} w_j \tilde{C}_j,$$

where $\tilde{C}_j$ are integer completion times, i.e., the output of Algorithm 5.

By using an external LP solver to solve the problem (3.19)–(3.24) as described in Section 3.2, we obtain the objective value $\mathrm{WCT}_{\mathrm{LP}}$, which we can compare to our $\mathrm{WCT}_{\mathrm{frac}}$. By instead using an external IP solver to solve (3.1)–(3.6), described in Section 3.1, we obtain the objective value $\mathrm{WCT}_{\mathrm{IP}}$, which we can compare to our $\mathrm{WCT}_{\mathrm{round}}$.

In Tables 5.1–5.3 we present the main results. The evaluation was carried out on a computer with two Intel Xeon Gold 6130 Processors (2×16 cores, 2.10 GHz)

and 96 GB memory. We used Julia version 1.7.3 and the flags `--optimize=3` and `--check-bounds=no`. The command run was

```
julia --project --optimize=3 --check-bounds=no src/main.jl [instance] 0.6
```

where `[instance]` represents the path to the instance as a JSON file, and 0.6 is the value we use for ε.

The packages which our implementation depends on (and their versions), apart from those included in Julia's standard library, are: DataStructures 0.18.13, JSON 0.21.3, LinkCutTrees 0.2.1, and DoubleFloats 1.2.2. For the external IP and LP solver, we used Gurobi 9.1.0, with the flag `MIPGap=0` when solving IP instances.

The implementation was run with $\varepsilon = 0.6$, giving us the generated approximation ratio $\alpha(\varepsilon) = 34.76$. See Section 3.4 for further discussion of this.

Table 5.1: Objective values and run time for instances in category L. The values are rounded to integers.

Instance	Implementation			IP solver	LP solver
Name	$\mathrm{WCT}_{\mathrm{round}}$	$\mathrm{WCT}_{\mathrm{frac}}$	Run time (s)	$\mathrm{WCT}_{\mathrm{IP}}$	$\mathrm{WCT}_{\mathrm{LP}}$
L_00008_22_01	732	711	10	897	820
L_00008_22_02	443	533	8	584	540
L_00008_22_03	895	988	8	988	988
L_00008_22_04	1020	1251	8	1354	1251
L_00016_64_01	6349	4626	17	8158	7106
L_00016_64_02	8639	6976	18	10 157	9319
L_00016_64_03	12 532	11 392	12	15 359	13 371
L_00016_64_04	11 937	9948	15	14 722	13 300
L_00032_181_01	143 891	95 293	38	182 198	159 134
L_00032_181_02	165 572	141 680	31	206 860	189 463
L_00032_181_03	134 624	89 897	52	178 721	150 064
L_00032_181_04	138 073	95 635	44	161 316	155 764
L_00064_512_01	2 382 783	1 429 030	163	2 986 310	2 751 540
L_00064_512_02	1 599 676	1 017 180	179	1 969 660	1 828 240
L_00064_512_03	1 817 165	1 123 910	197	2 212 160	2 080 260
L_00064_512_04	2 151 365	1 304 960	162	2 679 150	2 474 070
L_00128_1448_01	35 576 544	22 381 600	756	44 792 300	44 104 800
L_00128_1448_02	29 274 349	18 568 900	949	36 624 500	35 314 000
L_00128_1448_03	32 916 917	20 410 900	832	41 117 800	39 597 800
L_00128_1448_04	35 028 410	21 589 600	719	43 840 800	42 563 700
L_00256_4096_01	547 334 689	331 015 000	2989	678 937 000	664 352 000
L_00256_4096_02	553 736 923	333 420 000	3610	684 001 000	666 590 000

L_00256_4096_03	477 893 270	296 775 000	2765	585 863 000	574 650 000
L_00256_4096_04	510 726 979	312 627 000	3381	627 204 000	613 226 000
L_00512_11585_01	8 444 619 440	5 129 530 000	18 993	10 155 200 000	10 083 850 370
L_00512_11585_02	8 004 423 451	4 934 460 000	15 260	9 700 910 000	9 520 250 967
L_00512_11585_03	8 754 628 136	5 413 090 000	19 881	10 753 300 000	10 623 579 370
L_00512_11585_04	8 611 934 544	5 228 290 000	16 099	10 426 700 000	10 261 252 930
L_01024_32768_01	139 308 315 747	82 682 100 000	77 662	168 570 000 000	164 321 728 900
L_01024_32768_02	136 932 074 485	83 436 800 000	78 213	166 807 000 000	164 846 410 200
L_01024_32768_03	135 984 768 515	83 166 700 000	78 128	166 746 577 602	163 320 689 800
L_01024_32768_04	138 245 035 142	84 717 500 000	72 990	168 712 000 000	166 860 441 600

Table 5.2: Objective values and run time for instances in category M. The values are rounded to integers.

Instance	Implementation			IP solver	LP solver
Name	WCT_{round}	WCT_{frac}	Run time (s)	WCT_{IP}	WCT_{LP}
M_00008_8_01	674	446	11	935	714
M_00008_8_02	600	512	10	803	638
M_00008_8_03	281	225	9	373	288
M_00008_8_04	492	342	10	546	484
M_00016_16_01	6054	3855	22	7847	6727
M_00016_16_02	8819	5398	24	10 484	9743
M_00016_16_03	5211	3335	21	6060	5670
M_00016_16_04	5935	3791	23	7356	6726
M_00032_32_01	121 822	77 554	82	147 851	142 337
M_00032_32_02	102 532	67 632	81	125 339	122 273
M_00032_32_03	116 259	72 851	77	143 923	139 952
M_00032_32_04	70 062	46 372	60	86 498	82 849
M_00064_64_01	1 617 043	1 075 580	300	2 001 420	1 977 030
M_00064_64_02	1 422 757	961 961	269	1 775 470	1 747 340
M_00064_64_03	1 548 156	1 028 980	296	1 914 720	1 896 860
M_00064_64_04	1 419 688	945 930	299	1 757 890	1 733 560
M_00128_128_01	25 720 939	17 271 300	983	31 884 300	31 732 300
M_00128_128_02	22 260 587	15 195 100	961	27 735 700	27 615 700
M_00128_128_03	17 301 589	11 895 500	988	21 489 200	21 405 100
M_00128_128_04	18 311 343	12 476 500	967	22 801 300	22 698 000
M_00256_256_01	303 183 703	210 261 000	3436	381 160 000	380 508 000
M_00256_256_02	320 819 294	219 799 000	2674	403 099 000	402 503 000
M_00256_256_03	320 875 908	221 356 000	3323	403 079 000	402 425 000
M_00256_256_04	297 596 069	206 050 000	3407	374 017 000	373 443 000

M_00512_512_01	5 283 999 821	3 652 330 000	12 476	6 661 250 000	6 658 491 576
M_00512_512_02	5 894 228 366	4 057 080 000	12 220	7 411 490 000	7 408 432 317
M_00512_512_03	5 674 022 489	3 908 900 000	12 347	7 134 550 000	7 131 869 225
M_00512_512_04	6 243 908 566	4 261 800 000	12 733	7 814 330 000	7 811 479 723
M_01024_1024_01	83 857 953 774	57 892 200 000	43 865	106 368 000 000	106 356 693 800
M_01024_1024_02	83 999 399 156	58 031 300 000	43 656	106 111 000 000	106 101 424 600
M_01024_1024_03	88 384 642 117	60 819 200 000	43 132	111 600 000 000	111 584 093 600
M_01024_1024_04	85 523 791 573	59 020 300 000	51 068	108 209 000 000	108 198 605 200

Table 5.3: Objective values and run time for instances in category S. The values are rounded to integers.

Instance	Implementation			IP solver	LP solver
Name	WCT_{round}	WCT_{frac}	Run time (s)	WCT_{IP}	WCT_{LP}
S_00008_2_01	355	215	11	428	366
S_00008_2_02	235	148	11	305	235
S_00008_2_03	332	205	12	415	351
S_00008_2_04	434	277	12	566	460
S_00016_4_01	6565	4240	26	8164	7557
S_00016_4_02	5091	3276	25	6162	5664
S_00016_4_03	4876	3141	27	6082	5622
S_00016_4_04	7272	4687	24	9135	8540
S_00032_5_01	98 217	66 557	77	122 645	119 097
S_00032_5_02	83 999	56 207	77	105 860	101 640
S_00032_5_03	98 416	65 579	77	121 902	118 311
S_00032_5_04	102 431	68 845	80	127 625	124 013
S_00064_8_01	1 072 206	738 604	272	1 339 410	1 318 060
S_00064_8_02	1 299 270	886 673	285	1 605 770	1 593 060
S_00064_8_03	1 499 340	1 012 090	320	1 850 720	1 839 160
S_00064_8_04	1 185 994	814 946	285	1 481 800	1 462 690
S_00128_11_01	16 728 909	11 640 100	996	21 037 000	20 971 700
S_00128_11_02	19 337 169	13 417 300	891	24 381 800	24 311 200
S_00128_11_03	16 892 041	11 753 200	954	21 197 700	21 138 900
S_00128_11_04	16 177 125	11 270 400	928	20 245 800	20 190 200
S_00256_16_01	299 863 744	209 318 000	3423	378 177 000	377 899 000
S_00256_16_02	226 965 929	159 295 000	2898	287 475 000	287 209 000
S_00256_16_03	280 702 419	196 608 000	3290	354 566 000	354 231 120
S_00256_16_04	264 662 840	186 057 000	3380	334 526 000	334 170 000
S_00512_22_01	4 470 028 858	3 139 270 000	12 303	5 658 820 000	5 657 391 961
S_00512_22_02	4 523 605 078	3 173 220 000	12 169	5 737 140 000	5 735 548 265

S_00512_22_03	4 639 265 869	3 249 050 000	12 245	5 868 530 000	5 866 812 078
S_00512_22_04	4 545 401 163	3 186 290 000	12 245	5 753 760 000	5 752 503 683
S_01024_32_01	73 308 880 692	51 406 900 000	50 011	93 120 400 000	93 112 835 570
S_01024_32_02	67 726 651 324	47 644 000 000	42 133	86 110 600 000	86 105 203 350
S_01024_32_03	72 497 386 583	50 846 800 000	50 147	91 970 900 000	91 961 586 240
S_01024_32_04	69 611 534 639	48 912 700 000	41 568	88 482 000 000	88 472 501 120

5.2.1 Run Times

We plot the run time as a function of $n+\kappa$. Such a plot is found in Figure 5.1a. We should expect the run time to grow with

$$O\Big((n+\kappa)\cdot \text{polylog}(n+\kappa)\cdot \log^3 p_{\max}\cdot \frac{1}{\varepsilon^2}\Big) = O\Big((n+\kappa)\cdot \text{polylog}(n+\kappa)\cdot \log^3 n\Big),$$

given that $p_{\max} = O(n)$ and ε is constant w.r.t. n and κ.

We can also consider the run time as a function of just n, as we have done in Figure 5.1b. Here we see that the three categories L, M and S have quite similar run times. This indicates that the "density" of precedence constraints does not have a big effect on the actual performance, but instead that the main factor is the value of n, the number of jobs.

5.2.2 Objective Values

We can compare on one hand $\text{WCT}_{\text{round}}$ to WCT_{IP}, and on the other WCT_{frac} to WCT_{LP}. We present the ratios
$\text{WCT}_{\text{round}}/\text{WCT}_{\text{IP}}$ and $\text{WCT}_{\text{frac}}/\text{WCT}_{\text{LP}}$ in Tables 5.4–5.6. We see that these ratios are all quite close to 1.

The straightforward relations that one might initially expect to hold for these values are not necessarily true. One might assume that $\text{WCT}_{\text{round}}$ should be larger than WCT_{IP}, since the former is found by an approximation algorithm and the latter by an exact solver. Likewise, one might expect that WCT_{frac} should be larger than WCT_{LP}. Thus, we would expect all of the ratios in Tables 5.4–5.6 to be greater than 1.

However, first we can note that $\text{WCT}_{\text{round}}$ is the objective value for a solution to the original problem, i.e., Definition 1.2, while WCT_{IP} is the objective value for an IP formulation of the problem, which can increase each completion time C_j by up to a factor $(1+\varepsilon)$. The actual guarantees we have are in fact $\text{WCT}_{\text{round}} \leq \alpha\text{WCT}_{\text{opt}}$ and $\text{WCT}_{\text{IP}} \leq (1+\varepsilon)\text{WCT}_{\text{opt}}$, where WCT_{opt} is the optimal objective value for the original problem Definition 1.2. Thus it is possible that $\text{WCT}_{\text{round}}$ can be lower than WCT_{IP}.

Secondly, $\mathrm{WCT}_{\mathrm{LP}}$ is the objective value for a strict solution of the LP, while $\mathrm{WCT}_{\mathrm{frac}}$ is the objective value of a solution to a relaxation of the problem (see Theorem 3.7 in Section 3.4). In particular, the constraint $\mathbf{Px} \leq \mathbf{1}$ is relaxed to $\mathbf{Px} \leq (1 + O(\varepsilon))\mathbf{1}$, which results in the possibility for $\mathrm{WCT}_{\mathrm{frac}}$ to be lower than $\mathrm{WCT}_{\mathrm{LP}}$.

Table 5.4: A comparison of the objective values from our implementation with objective values from IP and LP solvers, for instances in category L.

Instance name	$\mathrm{WCT}_{\mathrm{round}}/\mathrm{WCT}_{\mathrm{IP}}$	$\mathrm{WCT}_{\mathrm{frac}}/\mathrm{WCT}_{\mathrm{LP}}$
L_00008_22_01	0.816	0.867
L_00008_22_02	0.759	0.989
L_00008_22_03	0.905	1.000
L_00008_22_04	0.753	0.999
L_00016_64_01	0.778	0.651
L_00016_64_02	0.851	0.749
L_00016_64_03	0.816	0.852
L_00016_64_04	0.811	0.748
L_00032_181_01	0.790	0.599
L_00032_181_02	0.800	0.748
L_00032_181_03	0.753	0.599
L_00032_181_04	0.856	0.614
L_00064_512_01	0.798	0.519
L_00064_512_02	0.812	0.556
L_00064_512_03	0.821	0.540
L_00064_512_04	0.803	0.527
L_00128_1448_01	0.794	0.507
L_00128_1448_02	0.799	0.526
L_00128_1448_03	0.801	0.515
L_00128_1448_04	0.799	0.507
L_00256_4096_01	0.806	0.498
L_00256_4096_02	0.810	0.500
L_00256_4096_03	0.816	0.516
L_00256_4096_04	0.814	0.510
L_00512_11585_01	0.832	0.509
L_00512_11585_02	0.825	0.518
L_00512_11585_03	0.814	0.510
L_00512_11585_04	0.826	0.510
L_01024_32768_01	0.826	0.503
L_01024_32768_02	0.821	0.506
L_01024_32768_03	0.816	0.509
L_01024_32768_04	0.819	0.508

Table 5.5: A comparison of the objective values from our implementation with objective values from IP and LP solvers, for instances in category M.

Instance name	WCT_{round}/WCT_{IP}	WCT_{frac}/WCT_{LP}
M_00008_8_01	0.721	0.625
M_00008_8_02	0.747	0.803
M_00008_8_03	0.754	0.783
M_00008_8_04	0.901	0.707
M_00016_16_01	0.772	0.573
M_00016_16_02	0.841	0.554
M_00016_16_03	0.860	0.588
M_00016_16_04	0.807	0.564
M_00032_32_01	0.824	0.545
M_00032_32_02	0.818	0.553
M_00032_32_03	0.808	0.521
M_00032_32_04	0.810	0.560
M_00064_64_01	0.808	0.544
M_00064_64_02	0.801	0.551
M_00064_64_03	0.809	0.542
M_00064_64_04	0.808	0.546
M_00128_128_01	0.807	0.544
M_00128_128_02	0.803	0.550
M_00128_128_03	0.805	0.556
M_00128_128_04	0.803	0.550
M_00256_256_01	0.795	0.553
M_00256_256_02	0.796	0.546
M_00256_256_03	0.796	0.550
M_00256_256_04	0.796	0.552
M_00512_512_01	0.793	0.549
M_00512_512_02	0.795	0.548
M_00512_512_03	0.795	0.548
M_00512_512_04	0.799	0.546
M_01024_1024_01	0.788	0.544
M_01024_1024_02	0.792	0.547
M_01024_1024_03	0.792	0.545
M_01024_1024_04	0.790	0.545

Table 5.6: A comparison of the objective values from our implementation with objective values from IP and LP solvers, for instances in category S.

Instance name	WCT_{round}/WCT_{IP}	WCT_{frac}/WCT_{LP}

S_00008_2_01	0.829	0.586
S_00008_2_02	0.771	0.629
S_00008_2_03	0.799	0.585
S_00008_2_04	0.767	0.603
S_00016_4_01	0.804	0.561
S_00016_4_02	0.826	0.578
S_00016_4_03	0.802	0.559
S_00016_4_04	0.796	0.549
S_00032_5_01	0.801	0.559
S_00032_5_02	0.793	0.553
S_00032_5_03	0.807	0.554
S_00032_5_04	0.803	0.555
S_00064_8_01	0.801	0.560
S_00064_8_02	0.809	0.557
S_00064_8_03	0.810	0.550
S_00064_8_04	0.800	0.557
S_00128_11_01	0.795	0.555
S_00128_11_02	0.793	0.552
S_00128_11_03	0.797	0.556
S_00128_11_04	0.799	0.558
S_00256_16_01	0.793	0.554
S_00256_16_02	0.790	0.555
S_00256_16_03	0.792	0.555
S_00256_16_04	0.791	0.557
S_00512_22_01	0.790	0.555
S_00512_22_02	0.788	0.553
S_00512_22_03	0.791	0.554
S_00512_22_04	0.790	0.554
S_01024_32_01	0.787	0.552
S_01024_32_02	0.787	0.553
S_01024_32_03	0.788	0.553
S_01024_32_04	0.787	0.553

5.3 Behaviour of the Algorithm

This section contains a detailed analysis of the steps of the implementation on a single instance, `M_00010_10_01`. We are interested to see how certain guarantees given in [28] hold, for example, if there is some slack in a certain inequality, or if it is a tight bound.

In Table 5.7, we present the objective value obtained from the oracle $\mathcal{O}$ at each t in the MWU loop (for a certain instance). The objective value $\mathbf{a}^T\mathbf{x}^{(t)}$ increases as $t \to 1$, and we also see that $\mathbf{P}\mathbf{x}^{(t)}$ increases, but its components

never exceed $(1+\varepsilon)^2+\varepsilon$, which in this case is $(1+0.5)^2+0.5 = 2.75$, as we use $\varepsilon = 0.5$. This is guaranteed by Theorem 3.8.

From (3.33) in Section 3.4.2, we have $\phi = 0.5w(J)$ giving us the guarantee

$$\mathbf{a}\mathbf{x}^{(1)} \geq \mathbf{a}\mathbf{x}^* - 0.5w(J),$$

where $\mathbf{x}^*$ is an optimal solution to (3.27).

In Table 5.8, we present the value of the flow, and the sum of capacities over the cut-set S' for each γ, when iterating over Γ (for a certain instance and iteration of the MWU loop). Here we compare the value of

$$a(S \setminus S') + \frac{\gamma}{1+\varepsilon} b(T(S'))$$

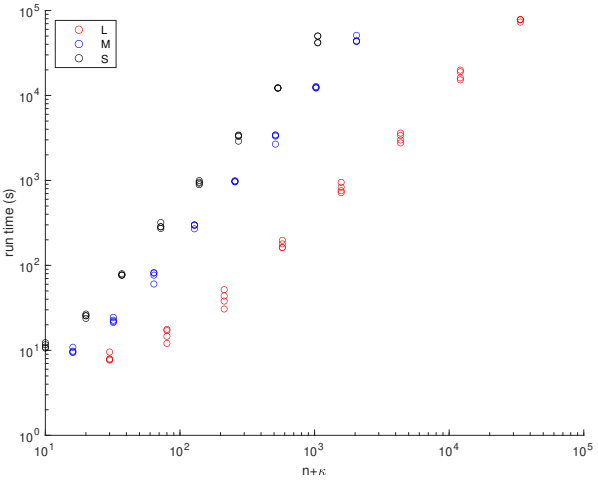
and the value of the flow $\mathbf{f}'$, for each iteration when calling `find_S'_and_f'`. Note that the guarantee in Theorem 4.6, i.e.,

$$a(S \setminus S') + \frac{\gamma}{1+\varepsilon} b(T(S')) \leq \text{val}(\mathbf{f}') + \frac{\phi}{3},$$

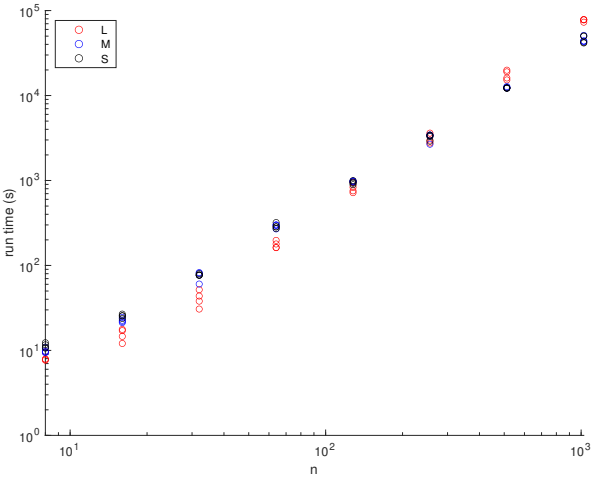
where $\phi/3 = 26.5/3 \approx 8.83$, holds empirically for each S'_γ and $\mathbf{f}'_\gamma$. In this case it even holds for $\phi = 0$.

Table 5.7: Details of each iteration of the MWU loop, for the instance `M_00010_10_01`. We used $\varepsilon = 0.5$ and $\phi = 26.5$.

t	$\mathbf{a}\mathbf{x}^{(t)}$	$\lvert\mathbf{P}\mathbf{x}^{(t)}\rvert_\infty$
0.000	0.000	0.000
0.032	120.093	0.101
0.064	240.187	0.201
0.095	360.280	0.302
0.128	484.610	0.402
0.171	644.298	0.503
0.219	821.601	0.594
0.267	998.022	0.672
0.314	1173.719	0.741
0.362	1348.810	0.803
0.410	1523.290	0.864
0.458	1699.597	0.964
0.516	1907.431	1.058
0.573	2114.912	1.152
0.630	2322.067	1.236
0.688	2529.021	1.319
0.745	2735.790	1.401
0.802	2942.379	1.483
0.860	3148.785	1.564
0.917	3354.999	1.644
0.974	3561.125	1.711
1.000	3654.212	1.756



(a) Plot of the measured run time as a function of $n+\kappa$ for our instances.



(b) Plot of the measured run time as a function of n for our instances.

Figure 5.1: Plots of the run time as a function of $n + \kappa$ and n. Note that both axes of the plots use a logarithmic scale.

Table 5.8: Details of the results of each iteration of `find_S'_and_f'`. These results were gathered at a MWU iteration with $t = 0.314$, for the instance `M_00010_10_01`, with $\varepsilon = 0.5$.

γ	$a(S \setminus S'_\gamma) + \frac{\gamma}{1+\varepsilon} b(T(S'_\gamma))$	$\text{val}(\mathbf{f}'_\gamma)$
8.833	15.407	18.869
10.819	18.869	23.110
13.250	23.110	28.304
16.228	28.304	34.665
19.875	34.665	42.455
24.342	42.455	51.997
29.812	51.997	63.683
36.513	63.683	77.996
44.719	77.996	95.525
54.769	95.525	116.994
67.078	116.994	143.287
82.154	143.287	175.490
100.617	175.490	214.931
123.230	219.453	262.373
150.926	262.373	314.941
184.846	318.875	375.181
226.389	375.181	444.141
277.268	446.154	516.985
339.583	541.975	599.497
415.903	605.794	668.059
509.375	668.059	744.318
623.854	749.493	830.613
764.062	830.613	929.966
935.781	947.235	1022.878
1146.093	1022.878	1115.521
1403.671	1115.521	1228.984
1719.139	1235.088	1361.024
2105.507	1363.126	1509.199
2578.708	1511.946	1671.305
3158.260	1714.771	1831.875
3868.063	1845.893	1963.665

Chapter 6

Conclusions and Discussion

We have implemented Li's approximation algorithm for the scheduling problem $1|\text{prec}|\sum_j w_j C_j$ in the programming language Julia. The implementation is quite simple, consisting of only about 2000 lines of code. We also have expanded on the description of the algorithm, in order to make it more accessible for nonexperts. Finally, we have evaluated our implementation on a set of randomly generated instances. In this chapter, we state our conclusions from the results and consider what further work could be done in this area. We also reflect on the development process and consider what worked well and what could be done differently.

6.1 Future Work

We start by describing possible extensions to the algorithm and other interesting future work.

6.1.1 Extensions to the Implementation

There are a few obvious improvements that could be done, regarding the implementation. Firstly, our implementation currently only handles instances for $1|\text{prec}|\sum_j w_j C_j$. However, Li's algorithm for $P|\text{prec}|\sum_j w_j C_j$ is based on the same LP relaxation as the algorithm for the single machine problem. Thus, if the list scheduling algorithm [28, Algorithm 2, p. 10] were to be implemented, we could also handle instances for $P|\text{prec}|\sum_j w_j C_j$. This is an important extension, as it would permit us to test and evaluate the performance of the algorithm on a much wider variety of instances.

Secondly, to actually handle all possible input instances, the LP relaxation must be modified slightly to handle the case where p_j (for all jobs $j \in J$) is not bounded above by poly(n). This is described in [28, Appendix E.1, pp. 35–36].

Some parts of the algorithm could possibly be performed in parallel. There are also a few other parts of the algorithm where some optimisations can be done. It is worth investigating whether these optimisations result in improved performance.

Given our implementation in the high-level language Julia, it should not take too much work to write an implementation in a more low-level programming language such as C++ or C. This could perhaps improve performance. Another possible path is to simply optimise the Julia implementation.

6.1.2 Further Testing of the Implementation

We have evaluated the implementation on a set of randomly generated instances. There are a number of interesting areas where further testing and evaluation of the implementation can be done.

If more testing were to be done, we could test even larger instances, both in terms of the number of jobs and precedence constraints. This could possibly enable us to find, or at least estimate, the size of instances for which the approximation algorithm is better than a general purpose LP solver. Such a point must exist, because the run time of a general LP solver grows faster than the run time of our algorithm, which has a run time which is approximately $(n + \kappa) \cdot \text{polylog}(n + \kappa)$, if we disregard a few technicalities. See Figure 6.1 for an illustration of how different functions grow in comparison to each other. However, note that it may be the case that the intersection point between the run time of our implementation and a general LP solver is at such a large n, that it is not relevant in practice.

One interesting extension to the evaluation could be to consider instances where the distribution that w_j or p_j are taken from is not uniform. For example, it could be interesting to see what happens if only a few jobs have nonzero weights, or if a few jobs have much larger processing times than the other jobs. It would not be particularly difficult to extend the instance generation function Algorithm 16 to take a parameter `w_distribution` instead of `w_domain`.

If multiple-machine rounding was to be implemented, this would enable us to run tests on instances gathered from many real world applications of scheduling. See for example the applications described in Chapter 1. There are some sets of publicly available instances for various scheduling problems. It could be interesting to evaluate our implementation on some such sets of instances.

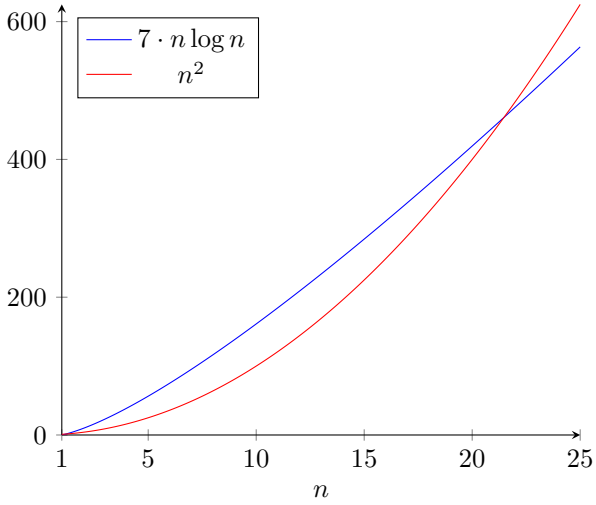


Figure 6.1: Example of the growth of two different functions. Here we see that $n^2 > 7 \cdot n \log n$ for $n > 21$.

6.2 Concluding Remarks

In this final section, we state some concluding remarks about the development methodology and some potential practical issues with our implementation that might be readily corrected.

6.2.1 Development Methodology

During the implementation of the algorithm, there were several points at which we could run the program, using an external solver instead of the "inner" parts of the algorithm (which we had not implemented at the time). At these points, we performed some tests to ensure that the algorithm still produced correct results. The outputs of these partial implementations were compared against each other to validate that they were consistent. This did hold true during the development, however, the variety of instances and the depth of the testing were not extensive enough, in retrospect. There were several mistakes and "bugs", which had to be corrected in the final implementation.

The milestones which we used were defined as: solving the IP model (3.12)–(3.18) using an external solver; solving the LP relaxation (3.19)–(3.24) using an external solver; solving the preprocessed LP (3.27) using an external solver; running the MWU loop and using an external solver as the oracle for

(3.32); finding the cut (S', T') described in Section 4.1.2 using an external solver; and finally finding the cut (S', T') using the algorithm described in Chapter 4, i.e., the complete algorithm.

6.2.2 Issues with Floating Point Numbers

The implementation as it is at the time of writing is not particularly robust to rounding errors. One example of where such a problem arises is when computing the support of a flow. The flow might have some components which are very near zero, but positive, due to floating point rounding errors. We must then take care not to include edges e where, for example, $f_e < 10^{-10}$, in $\operatorname{supp} \mathbf{f}$.

Another example is when computing the capacities of edges, before calling the blocking flow algorithm. The blocking flow algorithm requires that the capacities are nonnegative, however, they are defined by the expressions

$$c_{s^*,s} = a_s - \sum_{e \in \delta^+(s)} f'_e$$

and

$$c_{t,t^*} = \gamma b_t - \sum_{e \in \delta^-(t)} f'_e,$$

which both include a subtraction and a sum of the flow over a few edges. It is thus quite easy for a situation where the capacity should be zero, but it actually is slightly negative, to arise. This must also be handled, by setting c_e to zero whenever, for example $-10^{-10} < c_e < 0$.

It would be beneficial to make a systematic analysis of the code with regard to this problem. This should be done in order to identify and correct all areas where there might be problems.

To get more resilience to such numerical issues, the implementation currently uses extended precision floating point numbers (`Double64` from the `DoubleFloats` package). Although faster than the built-in `BigFloat`, these numbers are slower than standard floating point numbers (e.g., `Float64`). Thus, after a thorough revision of the code to better handle rounding issues, it might be possible to use `Float64`, and obtain a reduction of the run time.

6.2.3 The Practicality of the Algorithm

As we have noted above, the current implementation of the algorithm is not very useful in practice. Using an external LP solver on the LP relaxation, in combination with the rounding algorithm we have implemented, is significantly faster. However, if the changes in Section 6.2.2, together with a few other

optimisations of the algorithm, were to be implemented, we could expect the performance to improve quite a bit. It is not currently possible to say whether this would result in the implementation being competitive with an external LP solver, but perhaps for very large instances, it could be a good option. We believe this because the size of the LP relaxation of the problem grows with n. The number of variables is $O(n \log n/\varepsilon)$ and the run time of the best LP solvers grows faster than the run time of our implementation, with the number of variables. Thus, at least for large n, we should prefer our implementation. What remains is only the matter of speeding up our implementation so that the bound for what we consider "large" is low enough to be practical.

Bibliography

[1] D. Angluin and L.G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18(2):155–193, 1979. 10.1016/0022-0000(79)90045-X.

[2] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 5:121–164, 2012. 10.4086/toc.2012.v008a006.

[3] N. Bansal and S. Khot. Optimal long code test with one free bit. In *Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '09, pages 453–462, USA, 2009. IEEE Computer Society. 10.1109/FOCS.2009.23.

[4] C. Chekuri, T.S. Jayram, and J. Vondrak. On multiplicative weight updates for concave and submodular function maximization. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science*, ITCS '15, pages 201–210, New York, NY, USA, 2015. Association for Computing Machinery. 10.1145/2688073.2688086.

[5] C. Chekuri and R. Motwani. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Applied Mathematics*, 98(1):29–38, 1999. 10.1016/S0166-218X(98)00143-7.

[6] C. Chekuri and K. Quanrud. *Near-Linear Time Approximation Schemes for some Implicit Fractional Packing Problems*, pages 801–820. Society for Industrial and Applied Mathematics, 2017. 10.1137/1.9781611974782.51.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd ed.* MIT Press and McGraw-Hill, 2001.

[8] Y. Dinitz. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1 1970.

[9] M.E. Dyer and L.A. Wolsey. Formulating the single machine sequencing problem with release dates as a mixed integer program. *Discrete Applied Mathematics*, 26(2):255–270, 1990. 10.1016/0166-218X(90)90104-K.

[10] G. Even, J.S. Naor, S. Rao, and B. Schieber. Divide-and-conquer approximation algorithms via spreading metrics. *J. ACM*, 47(4):585–616, jul 2000. 10.1145/347476.347478.

[11] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[12] R.L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, 1966. 10.1002/j.1538-7305.1966.tb01709.x.

[13] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979. 10.1016/S0167-5060(08)70356-X.

[14] I. Griva, S.G. Nash, and A. Sofer. *Linear and nonlinear optimization*. Society for Industrial and Applied Mathematics, 2nd ed. edition, 2009.

[15] Y. Gurevich and S. Shelah. Nearly linear time. In A.R. Meyer and M.A. Taitslin, editors, *Logic at Botik '89*, pages 108–118, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. 10.1007/3-540-51237-3_10.

[16] L.A. Hall, A.S. Schulz, D.B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22(3):513–544, 8 1997. 10.1287/moor.22.3.513.

[17] T.D Hansen and U. Zwick. An improved version of the random-facet pivoting rule for the simplex algorithm. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 209–218. Association for Computing Machinery, 2015. 10.1145/2746539.2746557.

[18] Emil Karlsson, Elina Rönnberg, Andreas Stenberg, and Hannes Uppman. A matheuristic approach to large-scale avionic scheduling. *Annals of Operations Research*, 302:1–35, 07 2021. 10.1007/s10479-020-03608-6.

[19] L. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[20] S. Khot. On the power of unique 2-prover 1-round games. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 767–775, New York, NY, USA, 2002. Association for Computing Machinery. 10.1145/509907.510017.

[21] V. Klee and G. Minty. How good is the simplex algorithm? in inequalities. In *Proceedings of the Third Symposium on Inequalities*. Academic Press, 1972.

[22] E.L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. In B. Alspach, P. Hell, and D.J. Miller, editors, *Algorithmic Aspects of Combinatorics*, volume 2 of *Annals of Discrete Mathematics*, pages 75–90. Elsevier, 1978. 10.1016/S0167-5060(08)70323-6.

[23] Y.T. Lee and A. Sidford. Efficient inverse maintenance and faster algorithms for linear programming. 2015. 10.48550/ARXIV.1503.01752.

[24] Y.T. Lee, Z. Song, and Q. Zhang. Solving empirical risk minimization in the current matrix multiplication time. 2019. 10.48550/ARXIV.1905.04447.

[25] J.K. Lenstra and A.H.G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978. 10.1287/opre.26.1.22.

[26] J. Y-T Leung. *Handbook of Scheduling*. Computer and Information Science Series. Chapman & Hall/CRC, 2004.

[27] S. Li. Scheduling to minimize total weighted completion time via time-indexed linear programming relaxations. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 283–294, 2017. 10.1109/FOCS.2017.34.

[28] S. Li. Nearly-linear time approximate scheduling algorithms. *CoRR*, abs/2111.04897v1, 2021. https://arxiv.org/abs/2111.04897v1.

[29] A. Munier, M. Queyranne, and A.S. Schulz. Approximation bounds for a general class of precedence constrained parallel machine scheduling problems. In R.E. Bixby, E.A. Boyd, and R.Z. Ríos-Mercado, editors, *Integer Programming and Combinatorial Optimization*, pages 367–382. Springer Berlin Heidelberg, 1998. 10.1007/3-540-69346-7_28.

[30] M. Queyranne. Structure of a simple scheduling polyhedron. *Mathematical Programming*, 58:263–285, 1993. 10.1007/BF01581271.

[31] M. Queyranne and A.S. Schulz. Approximation bounds for a general class of precedence constrained parallel machine scheduling problems. *SIAM Journal on Computing*, 35(5):1241–1253, 2006. 10.1137/S0097539799358094.

[32] R. Ravi, A. Agrawal, and P. Klein. Ordering problems approximated: single-processor scheduling and interval graph completion. In *Automata, Languages and Programming*, pages 751–762, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. 10.1007/3-540-54233-7_180.

[33] N.Z. Shor. Convergence rate of the gradient descent method with dilatation of the space. *Cybernetics*, 6:102–108, 1970.

[34] D.D. Sleator and R. Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. 10.1016/0022-0000(83)90006-5.

[35] O. Svensson. Conditional hardness of precedence constrained scheduling on identical machines. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, STOC '10, pages 745–754, New York, NY, USA, 2010. Association for Computing Machinery. 10.1145/1806689.1806791.

[36] A.M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2–42(1):230–265, 1937. 10.1112/plms/s2-42.1.230.

[37] D.P. Williamson and D.B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011. 10.1017/CBO9780511921735.

[38] D.B. Yudin and A.S. Nemirovski. Informational complexity of strict convex programming. *Ekonomika i Matematicheskie Metody*, 12:550–559, 1976.

Appendix A

Additional Proofs and Algorithms

In this appendix, some algorithms which are commonly known and used are explained.

A.1 Topological Ordering

Algorithm 17 finds a topological ordering for a given graph.

Definition A.1. Let $G = (V, E)$ be a DAG. A topological ordering is an ordering of the vertices V such that if $(v, v') \in E$, then v comes before v' in the order.

We can represent a topological ordering by a function $\rho\colon V \to \mathbb{Z}_{\geq 0}$ that gives the index of each vertex in the ordering. This function should then be such that

$$(v, v') \in E \quad \Longrightarrow \quad \rho(v) < \rho(v'),$$

for it to represent a topological ordering.

Algorithm 17 works by recursively visiting each node, and then visiting its (unmarked) successors until it reaches one with no (unmarked) direct successors. Each node is marked after visiting it. This strategy is known as depth-first search (DFS) [7, Chapter 22.4, pp. 549–551]. The found node is then assigned the index i, and i is decreased. Then, we work backwards, giving each node v an index $\rho(v)$ such that

$$v \in \Delta^+(u) \quad \Longrightarrow \quad \rho(u) < \rho(v).$$

That the run time is $O(|V| + |E|)$ can be seen by noting that VISIT(v) is called

Algorithm 17 An algorithm that finds a topological ordering S for the DAG $G = (V, E)$.

Input: A graph $G = (V, E)$.
Output: A topological ordering S.

```
M ← all-false vector over V                    ▷ marked (i.e., visited) nodes
T ← all-false vector over V                    ▷ temporarily marked nodes
S ← empty vector
procedure VISIT(v)
    if M_v is true then
        return
    else if T_v is true then
        quit with error                        ▷ graph is not a DAG
    end if
    T_v ← true
    for u ∈ Δ⁺(v) do
        VISIT(u)
    end for
    T_v ← false
    M_v ← true
    push v to the beginning of S
end procedure
for v ∈ V do                                   ▷ order of iteration does not matter
    VISIT(v)
end for
```

once for each $v \in V$, i.e. $|V|$ times. The function VISIT(u) is also called once for each $u \in \Delta^+(v)$ where $v \in V$, resulting in a total of

$$\sum_{v \in V} |\Delta^+(v)| = \sum_{v \in V} |\delta^+(v)| = |E|$$

calls.

A.2 Finding a Blocking Flow

An algorithm for finding a blocking flow, i.e., a flow in which there is no s-t-path consisting entirely of unsaturated edges, was presented in 1983 by Sleator and Tarjan [34, pp. 387–389]. In Algorithm 18, their description of the algorithm is given. In Algorithm 19, our implementation of the algorithm is found.

Before we present the algorithm, we give a list of operations on link-cut tree nodes, which are used in the algorithm. For any two link-cut tree nodes v and w, the following operations (among others) are defined:

root(v) returns the root of the tree v belongs to,

parent(v) returns the parent node of v,

cost(v) returns the cost of the edge from v to its parent,

link(v, w, c) creates an edge from v to w with cost $c \in \mathbb{R}_{\geq 0}$,

cut(v) removes the edge from v to its parent,

mincost(v) find a node v' in the path from v to the root, such that the cost of the edge $(v', \text{parent}(v'))$ is minimised,

update($v, \Delta c$) update all costs of edges in the path from v to the root by adding $\Delta c \in \mathbb{R}$.

A.3 Auxiliary functions for Algorithm 15

Here we present three auxiliary functions. The first two, Algorithm 20 and Algorithm 21, are used by Algorithm 15 directly, to construct the handled graph G' and the graph R, respectively. The third, Algorithm 22, is used by Algorithm 12 to find the distance from unsatisfied sinks to all vertices in G.

Algorithm 18 Sleator and Tarjan's algorithm for finding a blocking flow.

Step 0 Create a forest of link-cut trees, with one tree for each vertex of the graph. Go to **Step 1**.

Step 1 Let $v = \text{root}(s)$. If $v = t$ go to **Step 4**; otherwise, go to **Step 2**.

Step 2 ($v \neq t$; extend path. If no edges leave vertex v, go to **Step 3**. Otherwise, select an edge (v, w) leaving v and perform $\text{link}(v, w, \text{capacity}((v, w)))$. Go to **Step 1**.

Step 3 (all paths from v to t are blocked) If $v = s$, compute the unused capacity of every tree edge using cost, and stop. Otherwise, delete from the graph every edge entering v. For each such edge (u, v) that is a tree edge, perform $\text{cut}(u)$, recording the unused capacity. Go to **Step 1**.

Step 4 ($v = t$; the tree path from s to t is augmenting) Let $v = \text{mincost}(s)$ (edge $(v, \text{parent}(v))$ is a minimum-capacity edge on the augmenting path). Let $c = \text{cost}(v)$. Perform $\text{update}(s, -c)$. Go to **Step 5**.

Step 5 (delete edges with no remaining capacity) Let $v = \text{mincost(s)}$. If $\text{cost}(v) = 0$, delete $(v, \text{parent}(v))$ from the graph, perform $\text{cut}(v)$, recording an unused capacity of zero, and repeat **Step 5**. Otherwise (when $\text{cost}(v) > 0$), go to **Step 1**.

Algorithm 19 Our implementation of Algorithm 18.

```
function find_blocking_flow(G::Graph, capacity::Vector, s_star, t_star)
    trees = [make_tree(Int, Int, R, v) for v in 1:G.n]
    m = length(capacity)
    unused_cap = zeros(R, m) # unused (residual) capacity
    s = trees[s_star]
    t = trees[t_star]
    while true
        while true
            # step 1
            v = find_root(s)
            if v === t break end # go to step 4
            # step 2
            if !isempty(δ_out(G, label(v)))
                # arbitrarily select an outgoing edge
                e = first(δ_out(G, label(v)))
                link!(v, trees[dst_vertex(G, e)], e, capacity[e])
                continue # go to step 1
            end
            # step 3
            if v === s
                for w_idx in 1:G.n # compute unused capacity
                    w = trees[w_idx]
                    u = parent(w)
                    for e in δ_out(G, w_idx)
                        unused_cap[e] += trees[dst_vertex(G, e)] === u ? cost(w) :
 capacity[e]
                    end
                end
                return [capacity[e] - unused_cap[e] for e in 1:m] # and stop
            else
                for e in collect(δ_inc(G, label(v)))
                    # record the unused capacity and cut incoming edges
                    u = trees[src_vertex(G, e)]
                    if parent(u) === v
                        unused_cap[e] += cost(u)
                        cut!(u)
                    else
                        unused_cap[e] += capacity[e]
                    end
                    delete_edge!(G, e) # and delete e from the graph
                end
                continue # goto step 1
            end
        end
        # step 4
        (v,c) = find_mincost(s)
        add_cost!(s, -c)
        while true
            # step 5
            (v,c) = find_mincost(s)
            if parent(v) === nothing
                break
            end
            if c > 0
                break # go to step 1
            end
            delete_edge!(G, edge_label(v))
            cut!(v)
            # record unused capacity of 0 and go to step 5
        end
    end
end
```

Algorithm 20 Constructing the graph G'.

```
function construct_G′(
    l::Int,
    G::Graph,
    a::Vector{R},
    b::Vector{R},
    adj_Gi_minus::Vector{Vector{Tuple{Int,Vector{Int}}}}) where R<:Real
    # construct G′
    # this assumes that none of the edges of G have been removed
    G′_edges = [(src_vertex(G, e),dst_vertex(G, e)) for e in 1:G.m]
    n′ = G.n
    π′ = collect(1:G.n)
    # again assume that no edges have been removed
    π′_edge = collect(1:G.m)
    π′_edge_inv = [Tuple{Int,Int}[] for _ in 1:G.m]
    for i in 1:l
        π′_inv = zeros(Int, G.n)
        for (v,_) in adj_Gi_minus[i]
            if a[v] > 0 || b[v] > 0
                # vertices in S or T are not copied
                π′_inv[v] = v
            else
                n′ += 1 # add new vertex
                push!(π′, v) # record what it's a copy of
                π′_inv[v] = n′ # = the new vertex
            end
        end
        for (v,out) in adj_Gi_minus[i]
            for e in out
                u = dst_vertex(G, e)
                push!(π′_edge, e)

                push!(G′_edges, (π′_inv[v],π′_inv[u]))
                e′ = length(G′_edges)
                push!(π′_edge_inv[e], (i,e′))
            end
        end
    end
    G′ = Graph(n′, G′_edges)
    (G′,π′,π′_edge,π′_edge_inv)
end
```

Algorithm 21 Constructing the graph R.

```
function construct_R_graph(
    l, G::Graph, G′::Graph, f′::Vector, a::Vector, b::Vector, γ,
    adj_Gi_plus, adj_Gi_minus, Si, Ti, π′_edge_inv)
    R_edges = Tuple{Int,Int}[] # R = (V′,E_R), C : E_R → [0,∞]
    C = R[]
    inf_cap = 2*sum(a) # larger than any necessary capacity
    # note that -1 is never a valid edge index
    edge_in_R_graph = fill(-1, G′.m)

    for i in 1:l+1
        for (v,out) in adj_Gi_plus[i]
            for e in out
                u = dst_vertex(G, e)
                push!(R_edges, (v,u))
                edge_in_R_graph[e] = length(R_edges)
                push!(C, inf_cap)
            end
        end
    end
    for i in 1:l
        for (_,out) in adj_Gi_minus[i]
            for e in out
                e′ = get_edge_inv(π′_edge_inv, e, i)

                u = dst_vertex(G′, e′)
                v = src_vertex(G′, e′)
                push!(R_edges, (u,v)) # note reversed direction
                edge_in_R_graph[e′] = length(R_edges)
                push!(C, f′[e′])
            end
        end
    end

    s_star = G′.n+1
    for s in Si[1]
        push!(R_edges, (s_star,s))
        # edge_in_R_graph[e] = 0
        c = a[s] - sum(f′[e] for e in δ_out(G′, s); init=zero(R))
        c = warn_if_negative(c)
        push!(C, c)
    end
    t_star = G′.n+2
    for t in Ti[l+1]
        push!(R_edges, (t,t_star))
        # edge_in_R_graph[e] = 0
        c = γ*b[t] - sum(f′[e] for e in δ_inc(G′, t); init=zero(R))
        c = warn_if_negative(c)
        push!(C, c)
    end

    # clean up near-zero negative floats
    for k in 1:length(C)
        if C[k] < 0 && abs(C[k]) < VARIABLE_TOL
            C[k] = zero(R)
        end
    end

    R_graph = Graph(G′.n+2, R_edges)
    (R_graph,C,edge_in_R_graph,s_star,t_star)
end
```

Algorithm 22 Finding the distance from unsatisfied sinks.

```
function find_distance_from_S_and_T(L, G′::Graph, a::Vector, S::Vector, T::Vector, f′::
    Vector)
    N = G′.n
    F_edges = [(src_vertex(G′, e),dst_vertex(G′, e)) for e in 1:G′.m]
    weight = zeros(Int, G′.m)
    for e in 1:G′.m
        if f′[e] > 0
            # the edge (v,u) is in the support of f′, so add the edge (u′,v′) to F
            v = src_vertex(G′, e)
            u = dst_vertex(G′, e)
            push!(F_edges, (N+u,N+v))
            push!(weight, 0)
        end
    end
    for s in S
        # add edge from s′ to s with weight 1
        push!(F_edges, (N+s,s))
        push!(weight, 1)
    end
    for t in T
        # add edge from t to t′ with weight 1
        push!(F_edges, (t,N+t))
        push!(weight, 1)
    end
    F = Graph(2*N, F_edges)

    # construct S[1], ..., S[L+2] and T[1], ..., T[L+2]
    q0 = Queue{Int}()
    q1 = Queue{Int}()
    distance = fill(10*(L+1), 2*N) # 10*(L+1) is basically infinity
    visited = falses(2*N)

    for s in S
        out_flow = sum(f′[e] for e in δ_out(G′, s); init=zero(R))
        if EXPRESSION_TOL < a[s] - out_flow # s not saturated by flow f′
            enqueue!(q0, N+s)
        end
    end

    d = 0
    while true
        if !isempty(q0)
            v = dequeue!(q0)
            if !visited[v]
                distance[v] = d
                visited[v] = true
                for e in δ_out(F, v)
                    if weight[e] == 0
                        enqueue!(q0, dst_vertex(F, e))
                    else
                        enqueue!(q1, dst_vertex(F, e))
                    end
                end
            end
        elseif !isempty(q1)
            (q0,q1) = (q1,q0) # swap q0 and q1
            d += 1
        else
            break
        end
    end
    distance
end
```