Matching and Scheduling Algorithms on Large Scale Dynamic Graphs

Author:
Xander van den Eelaart

Supervisors:
Prof. dr. Rob H. Bisseling
ir. Marcel F. van den Elst
Second reader:
dr. Tobias Müller

Master's Thesis Mathematical Sciences
November 28, 2013

Universiteit Utrecht

Preface

I have done the research for this thesis at the company Progressive Planning, where I was given the opportunity to choose topics to investigate. On the one hand, I appreciated using mathematical reasoning to solve practical business problems. On the other hand, I enjoyed researching a problem in scientific computing, my field of study, while adopting a theoretic approach. These problems are *scheduling* and *matching*, respectively, and both concern one of the most important data structures in this age: graphs. Therefore, I think that the reader will appreciate this thesis for presenting two different approaches to developing fast graph algorithms.

Acknowledgments

I would like to thank my supervisor and tutor Rob Bisseling, who has guided me through my Master in Mathematics and specifically through this thesis. His help in developing and criticizing graph algorithms was extremely useful, as well as his knowledge about formulating and structuring the thesis, and the proofs in particular. I would also like to thank my boss Marcel van den Elst. He gave me the opportunity to apply my knowledge on the work floor, and his insights and enthusiasm helped greatly in producing this work.

Contents

Chapter 1

Introduction

Graphs are a powerful mathematical data model that emphasizes the relationships within the data. We do not mean the graphical representation of functions, but graph theory, where a graph consists of a set of vertices, or nodes, connected by edges. The vertices in a graph could represent the users of a social network application so that the edges are the friendship connections between the users. In another application the nodes are pages on a website and the edges are the hyperlink references from one page to another. As these graphs can be large, the computations manipulating them should be efficient.

This thesis introduces a specific type of graph in Chapter 2 that restricts the order in which tasks can be executed by their processors. For this problem, the tasks are the vertices and the edges denote the precedence relations between nodes. Such a graph is *directed*; an edge from the first task to a second task means that the first precedes and should be executed before the second, not the other way around. This is unlike the friendship relations in the social, *undirected* graph, where an edge from one person to the other means that both are friends with each other. We study a specific problem, *matching*, on undirected graphs in Chapter 3. Matching is about forming pairs of connected nodes in a graph, and we consider algorithms that quickly compute them.

Many graphs undergo changes; they are *dynamic*. In the social graph, new users sign up and friendships are made or broken. For the web graph, entire websites can be launched or taken off the internet and many links are modified each day. We embrace the dynamism of graphs and research algorithms that can incorporate changes effectively. The power of such algorithms is that they exploit the fact that changes generally occur gradually. We can reuse most of the information that we have already obtained on the graph before it faced changes. As graphs become bigger, the need for such dynamic algorithms grows larger as well, because recomputing the solution to a problem turns inefficient.

1.1 Progressive Planning

This research has been motivated by the company Progressive Planning, which has developed organizational methods for planning and managing projects. Progressive Planning is a small and young company started in Utrecht, The Nether-

lands, in 2009 with the goal of reinventing project management worldwide. Its main difference from conventional approaches to project management is that the people doing the actual work for the projects, also do the planning. In the Progressive Planning web application, all contributors to a project describe their own work, including the relationships to work of others and the estimates for the workload. The application computes a schedule for the work of all contributors. This bottom-up approach has two advantages in particular. First, it creates ownership, responsibility, and accountability in the project contributors. Second, the data is entered and updated by the people who know that data the best: the people doing the actual work. This means that the schedule and other statistics are as accurate and up-to-date as possible.

In the Progressive Planning web application, people can subscribe and create different roles in which they can structure and plan their work. Roles have a set availability which can be seen as a working speed rate; for example, one person could be a software engineer 30 hours per week and a student 10 hours a week, and plan his work and academic projects in the respective roles.

In a role, a person can create deliverables which represent the products of work, not work itself. Deliverables in turn, contain activities, on which the user can estimate the amount of time required to perform each activity. Activities within a deliverable should be done in a sequential order, specified by the user. A deliverable is finished as soon as all its activities are done.

Deliverables can be used to make a product breakdown by indicating parent-child relationships between the deliverables. In this way, a top deliverable can be divided into subdeliverables in a tree structure. This entails that a single deliverable can have several child deliverables, but only one parent deliverable. Team work is structured by creating parent-child relationships between deliverables of different people. Activities within a deliverable can only be started after all of its child deliverables are finished. In addition to the constraint of sequential order of execution on activities, they can also be dependent on other deliverables. Such a dependency entails that the activity cannot be started before the deliverable it depends on is completed.

Summarizing, there are three types of precedence constraints, which are all of the form finish-to-start: the first activity in a deliverable can start after its deliverable's child deliverables are finished; a subsequent activity within a deliverable can start only after its preceding activity is finished; and every activity has to wait for all the deliverables it is dependent on. As such, a single activity can have several precedence constraints, so it can only start after the last is relieved.

The user can also set various time constraints on roles, deliverables, and activities. Firstly, a role can have a start or end time, or both. This means that the role is only available for doing activities after the start time or before the end time. Secondly, an activity can have a fixed start time at which it will be done by its role. Such activities can be seen as calendar items; for example, the activity 'attend the conference' could be scheduled at a fixed start time. Thirdly, deliverables can have a deadline, which is a time before which they should be finished.

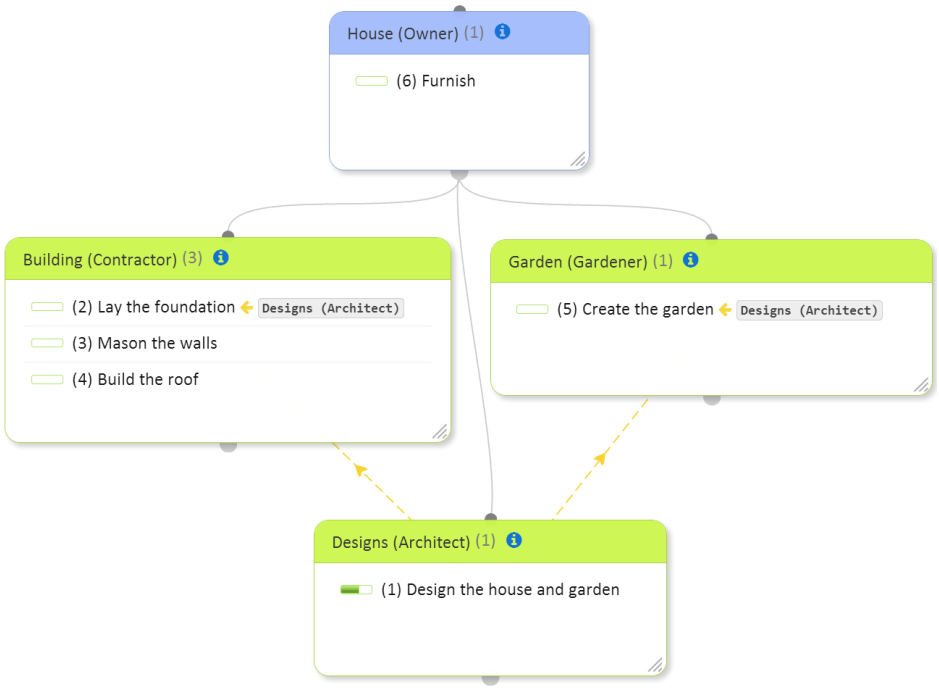
The challenge is that Progressive Planning should provide the users with time schedules of their activities and deliverables for their different roles. We refer to the start and end times of all activities and deliverables in the system as the global

schedule, or simply the schedule. It should definitely be feasible, or consistent: activities should take their estimated time, be scheduled one at a time on their role, not exceed the role's availability, and start only after their predecessors have completed. We allow preemption, which is the interrupting of a scheduled activity to continue processing where it was left off at a later time, allowing other jobs to be scheduled in between. Moreover, the schedule should minimize the completion times of all activities and deliverables, favoring those with earlier deadline or higher risk. Most importantly, the schedule should be computed rapidly. It is, however, not clearly defined how to strike a balance between these criteria.

1.1.1 Example

Figure 1.1 depicts an example of the product breakdown of building a house in Progressive Planning. There are four people in the roles of owner, contractor, gardener, and architect. They partake in the project "House", which is divided up in three subdeliverables: building, garden, and design. Each deliverable in turn contains one or more activities.

Figure 1.1: Product breakdown of a project of a house consisting of four deliverables, created in Progressive Planning. Each deliverable contains one or more activities, dependencies are depicted by the yellow arrows and the gray lines denote that "House" has three child-deliverables.



The roles need to spend a certain amount of hours to complete each activity, as depicted in Table 1.1. Given this information, the amount of time it takes between the start and completion of an activity is determined by the availability of its role. Suppose that the architect, the contractor and the owner can work

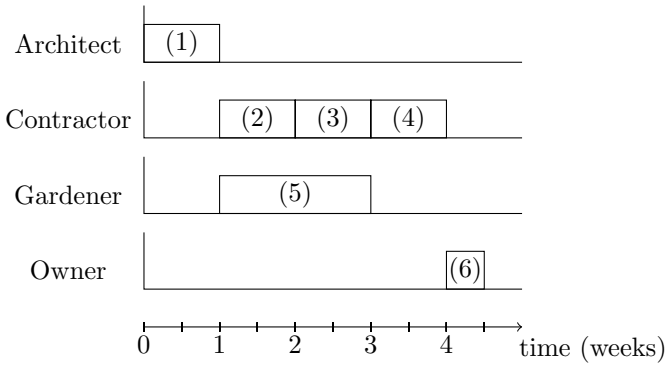
40 hours per week and the gardener can work 20 hours per week. In that case, the time between the start and end time of each activity is given by the duration column in Table 1.1.

Table 1.1: The amount of work, role and duration of each activity in the project of Figure 1.1.

Activity	Work	Role	Duration
(1) Design the house and garden	40 hours	Architect	1 week
(2) Lay the foundation	40 hours	Contractor	1 week
(3) Mason the walls	40 hours	Contractor	1 week
(4) Build the roof	40 hours	Contractor	1 week
(5) Create the garden	40 hours	Gardener	2 weeks
(6) Furnish	20 hours	Owner	1/2 week

The precedence constraints in this project are implicit in Figure 1.1. The activity "Furnish" cannot start before the building, garden and design are completed. The activities "Lay the foundation" and "Create the garden" are dependent on the design to finish first. Lastly, the activity "Mason the walls" can start only after "Lay the foundation" and is in turn succeeded by "Build the roof". Unless otherwise specified, the schedule always plans the remaining work to start right now. Using this information, a possible schedule is presented in Figure 1.2.

Figure 1.2: Gantt chart of a possible schedule for the activities, referenced by their number, in the project of Figure 1.1. Vertically, a calendar is given for each role in the project, where time is plotted horizontally.



In Figure 1.2, the activity "Design the house and garden" is scheduled for the architect to start with right away. Therefore it will be finished a week from now, meaning that the design has finished. Only at this point, it is possible to start the activities for the building and the garden. They can be done in parallel, because the gardener and the contractor are independent. Lastly, "Furnish" is planned to start right after the building has finished.

Note that the order of execution of the activities in this example is entirely determined by the precedence constraints. Therefore, it is easy to confirm that this schedule is the fastest way to finish the project. However, it is not the only possible schedule, because we could delay the furnishing until week 5, for example.

1.2 Overview

The precedence graph of the tasks in Progressive Planning which motivated this thesis, is large and dynamic. Therefore, our focus is on fast practical algorithms for creating a task schedule, finding the longest chain of tasks and maintaining certain properties and values on the graph. Some of these problems, we try to solve dynamically. This will be the topic of Chapter 2.

Matching is a technique that is widely used in graph partitioning, the splitting of a graph in smaller components. As such, it can be used to divide the people, deliverables, and activities over the data servers in such a way that most information requested by one user can be retrieved from a single server. In Chapter 3, we develop a dynamic matching algorithm that is suited for parallel computing.

Chapter 2

Scheduling

Scheduling theory is a branch of combinatorial optimization that can be used to formalize the problem of Progressive Planning. It covers an extensive area of research with many applications in business optimization and the allocation of computer resources to computation tasks. In Section 2.1, we introduce the notation commonly used in scheduling theory. This helps us in our attempts to define the problem more formally. However, as we argue in Section 2.1.2, focusing on optimization clouds the presence of the great uncertainty in the data. For this and other reasons, we develop an algorithm in Section 2.2 based on simple heuristics.

Section 2.3 discusses splitting up the precedence graph in connected components. This can greatly reduce computation time, because it reduces the problem size greatly. We present an easy approach to analyze the quality of the resulting schedule by measuring the critical path in Section 2.4. Lastly, we discuss several algorithms and approaches concerning the dynamism of the precedence graph in Section 2.5.

2.1 Introduction to scheduling theory

Our problem can be formulated as a problem in scheduling theory, an area of research which is concerned with mapping task jobs to time slots on machines. Following Brucker [9] and Pinedo [25], the appropriate nomenclature and notation is as follows. Deliverables and activities make up a set J of n *jobs*, $J = \{j_1, j_2, \ldots, j_n\}$. Here job j_i has a deterministic *processing time* p_i, which is the activity's duration and zero if it is a deliverable. The precedence relations are given by a directed acyclic graph $G = (V, E)$, where $V = J$ is the set of vertices, and E a set of edges. An edge $(j_1, j_2) \in E$ indicates that job j_1 needs to be done before j_2 can start. The roles are modeled by a set M of m parallel *machines*, or *processors*, $M = \{M_1, M_2, \ldots, M_m\}$.

In scheduling theory, it is uncommon that jobs can only be processed by a specific processor, so there is no general notation. However, we could view the processors as disjoint sets of jobs:

$$M_i \subseteq J, \quad \forall i$$

$$M_i \cap M_j = \emptyset, \quad \forall i \neq j$$

$$\bigcup_{i=1}^{m} M_i = J.$$

Here, $j \in M_i$ means that job j has to be processed by M_i. Because every job is uniquely assigned to a role, we can scale its processing time by the availability of its role. This gives the actual duration of a job's execution, just as we did in the example in Section 1.1.1.

Time constraints, on the other hand, are often modeled in scheduling theory. Jobs corresponding to a deliverable with a deadline have a due date d_i before which they should be completed. Fixed start time activities are more tricky; they could, for example, be incorporated by putting a release date r_i and a due date $d_i = r_i + p_i$ on such a job. In this way, the job only becomes available at its fixed start time r_i, and has to be scheduled immediately in order to make its deadline. These release and due dates can also be used for roles with start or end times. Jobs corresponding to a role with a start time, have a release date equal to the start time of the role, not overriding their own release date if they have one. Similarly, for a role with an end time, the jobs can be given the end time as a due date.

We do not only need to meet the above constraints, as scheduling is also an optimization problem. Consequently, we require some objective function f to minimize. Assume that the resulting schedule is feasible, meaning that it adheres to all the rules defined above. Also, let c_i be the completion time of job j_i. In that case, it is sensible to have a function on these completion times, $f : (c_1, c_2, \ldots, c_n) \mapsto \mathbb{R}$. Common measures in scheduling theory are the *makespan* ($C_{\max} = \max\{c_i\}$) and sum of weighted completion times ($\sum w_i c_i$), or a sum or maximum of the lateness ($L_i = c_i - d_i$) or tardiness ($T_i = \max\{0, L_i\}$). Given some objective function, the optimization problem is to minimize it over the set of all feasible schedules.

2.1.1 Job-shop scheduling

Job-shop scheduling is a problem that resembles our case most closely, and is covered thoroughly in the literature. In this problem, a job consists of one up to m operations, or subjobs. Each operation has its own processing time and is assigned to a specific processor, with no more than one operation per job on each machine. The operations of a single job have to be executed in a certain order. However, this order can be different for the operations of different jobs. There are no other precedence constraints than these orders, so the precedence graph consists of simple paths; each operation has at most one predecessor and at most one successor. Job-shop scheduling can be seen as a production process where each job has to go through several stages of production before it is finished.

The similarity between the job-shop problem and Progressive Planning is that the operations are already assigned to their processors. However, because of the structure of jobs and their operations in job-shop scheduling, the type of precedence constraints is limited. In our case, there can be any precedence constraints as long as there is no circular dependency. As such, we can argue that our problem is in fact a generalization of job-shop scheduling.

One reason why this is a helpful realization, is that Gonzalez and Sahni [17] have proven that the decision version of the job-shop problem is NP-complete in the strong sense. Here, the decision version is for some constant B: given an instance of the job-shop scheduling problem, does there exist a feasible schedule with $C_{\max} \leq B$? This result is still valid even if there are only two processors and preemption is allowed. Consequently, the job-shop optimization problem is NP-hard and so is the scheduling problem for Progressive Planning if we try to minimize the makespan. As a result, it is likely that there does not exist an algorithm to solve this problem of which the running time is bounded polynomially by the input size.[1] Although we do not only want to minimize the makespan, it can be safely assumed that the problem remains NP-hard if we try to minimize other common objective functions.

A common approach to the job-shop problem is to formulate it as a (mixed) integer linear program, as is described by Pinedo [25], who provides a branch-and-bound technique to solve the program optimally. Two approximation algorithms are given by Porto and Ribeiro. [26], who apply a Tabu search method to job-shop scheduling, and Bierwirth and Mattfeld [6], who present a genetic algorithm for the problem. Both claim to obtain efficient and good approximations to specific instances of versions of the job-shop scheduling problem. However, these methods all have in common that they require the generation of large sets of feasible schedules, which has as a result that with increasing problem size, either these algorithms do not scale well, or their solutions become less accurate. Not surprisingly, the size of typical problem instances solved in the literature does not exceed a hundred jobs.

2.1.2 Heuristic approach

The simplest heuristic applied in scheduling theory is *list scheduling*, which tries to schedule jobs according to their order in some predefined list, as described by Brucker [9]. In its most basic form, the list is just a random ordering of the jobs. This order could be in conflict with the ordering of jobs enforced by the precedence constraints, but can be overcome by treating the list order as a type of priority: at any moment, schedule the available job that has the highest order in the list. A job is *available* when all its predecessors have been completed and it has been released. Depending on the order in the list, list scheduling is a natural approach to scheduling in many environments: think of a cashier who serves customers according to their order in the queue. However, as far as we know, list scheduling has not been analyzed for job-shop scheduling, let alone our generalization of it, probably because of this ambiguity in conflicting orders of the list and precedence constraints.

Depending on the implementation, list scheduling has a running time of $O(n \log n)$ because of the sorting, which makes it incredibly attractive for the basis of a scalable algorithm. Also, taking the list as a random order would be naive; the trick

[1] That the decision version of Job-shop scheduling is NP-complete means that it is at least as hard as all other problems in NP (the class of polynomial verifiable decision problems). It is strongly NP-complete, because it remains NP-complete even if all the numerical input variables are bounded by a polynomial in the input size. Job-shop scheduling as optimization problem is strongly NP-hard, because a solution cannot be verified in polynomial time; it is at least as hard as all problems in NP.

of list scheduling is to find a good list. The best list is of course the one that leads to an optimal schedule.

Scheduling theory is a branch of mathematical or business optimization that is concerned with extremely controlled environments. Generally, all variables are assumed to be known exactly, and in case some are not, their distributions are known. Applications are in production environments, where the processing time for each job on each machine can be measured precisely and stochastic events such as the arrival of new jobs can easily be assumed to follow some distribution. The environment of Progressive Planning is different; absolutely everything can, and does change. The strength is that the method allows users to continuously update their estimates of the structure and duration of the work. Moreover, it is aimed for such general applications of project planning that no coherent objective function on the schedule quality can be formulated. Therefore, formulating the case of Progressive Planning as a scheduling problem as we have done in the previous sections, conceals these intrinsic problems. In the next sections, we loosen the assumptions enforced by scheduling theory, while we realize that scheduling theory does not provide for satisfiable and fast solution algorithms for our problem in any case. We will present our own scheduling algorithm that resembles list scheduling, which has been implemented in the Progressive Planning application. In Section 2.2.1, we develop the notion of decision moments, which formalizes what is meant when we say that list scheduling schedules the highest ordered job *at any moment*. In Section 2.2.2, we elaborate on the construction of the order in the lists.

2.2 Scheduling for Progressive Planning

The number of activities and deliverables in Progressive Planning that can be scheduled should be large, possibly in the order of millions. Also, note that changes are frequent and that a single change could affect the entire schedule. For example, consider the case that a user adds a new activity which has a deadline for the same day. The schedule should probably move all other jobs of that user forward in time to make place for this new activity, and consequently all other jobs that are dependent on these moved jobs need to be rescheduled, and so on. Theoretically, the entire schedule could be different. Summarizing, we are scheduling on a large dynamic graph, and a change can already cost $O(n)$ operations. Take this together; if we can schedule the entire problem in linear time, this would reduce the need for a fully dynamic algorithm (which in the general case cannot do better than $O(n)$). This has led us to build on the near-linear list scheduling and develop a method that we call *highest-priority-first scheduling*.

2.2.1 Highest-priority-first scheduling

The main question throughout this chapter is: given a set of tasks and people, when should everyone do which of their tasks? Because we do not know what other work is coming our way, we start the work as soon as possible. Assume that each job has a certain priority measure, meaning that it is preferable that we start and finish a job with a higher priority over one with a lower priority. Then

it makes sense that each person in each of their roles starts right away with the available job with the highest priority. At the time that the first job finishes, we need to make a new decision for that role on which job to schedule next. These completion times are the *decision moments* at which the scheduler in Algorithm 2.1 decides which jobs a role should perform next. For the moment, we leave out the possibility of preempting a job to complete it at a later time, which makes Algorithm 2.1 aptly named the non-preemptive, highest-priority-first scheduler.

Algorithm 2.1 Non-preemptive, highest-priority-first scheduler

Output: A schedule $\{(s_1, c_1), (s_2, c_2), \ldots, (s_n, c_n)\}$ of starting times s_j and completion times c_j of each job $j \in \{1, 2, \ldots, n\}$.

```
1: while there are decision_moments do
2:    remove the minimum value t from decision_moments
3:    for each job j completed at time t do
4:       set j's successors as available if they have no uncompleted predecessors
5:    for each idle role r with available jobs do
6:       let j be the available job with highest priority on r
7:       schedule j from s_j := t to c_j := t + p_j
8:       add c_j to decision_moments
```

The scheduler roughly described in Algorithm 2.1 loops over a set of *decision_moments*, which is initialized to only contain the point of time 0. In each iteration, it takes the earliest moment in time from *decision_moments*. In the first iteration, this will be $t = 0$ and the scheduler will continue to loop over all the roles. All roles are idle, because they have not been scheduled to perform any tasks yet, and for each role it is determined which of its jobs are available for processing. A job is available when all its predecessors have been completed, which at $t = 0$ means that it does not have any predecessors. We pick the job j with highest priority from this set of available jobs, and schedule it by setting its starting time s_j to the current time t and its completion time c_j to its processing time added to its starting time. All the completion times of the jobs scheduled at $t = 0$ are added to *decision_moments*.

The scheduler continues to the next earliest point in time t of *decision_moments*. Now, at least one job has finished, because adding its completion time to *decision_moments* caused us to arrive at t. For all jobs that have been finished since the previous decision moment, we check if their successors in the precedence graph are now available. Now, all roles that have a job with a completion time greater than t are still occupied. The other, idle roles need to consider their available jobs again to schedule the one with the highest priority.

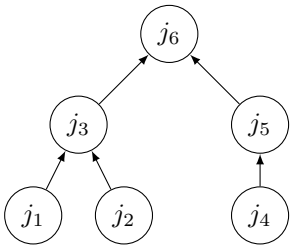
In this way, the scheduler continues moving forward in time until the set *decision_moments* is empty. When that happens, all the jobs have been scheduled, because at the last decision moment, we checked every role for available jobs and found none. The precedence graph is directed and acyclic, which means that if we remove jobs that have been completed, at any point in time there either has to be at least one job without predecessors, or the graph is empty and all jobs have been scheduled.

Example of the non-preemptive, highest priority first scheduler

The example given in Figure 2.1 shows how Algorithm 2.1 could run. It consists of two roles, each with three jobs. As *decision_moments* is initialized to contain the time point 0, the first iteration over *decision_moments* starts with $t = 0$. At this point, for role r_1 the jobs j_1 and j_2 are available. The algorithm picks j_1, because it has a higher priority $w_1 = 2$ than job j_2 with $w_2 = 1$. Consequently, j_1 is scheduled from $s_1 = t = 0$ to $c_1 = s_1 + p_1 = 2$, and c_1 is added to *decision_moments*. For role r_2, only job j_4 is available at this point, hence it is picked and scheduled from $s_4 = 0$ to $c_4 = 1$.

Figure 2.1: Example of a scheduling instance consisting of six jobs on two roles, and the output of Algorithm 2.1 for the instance. The precedence constraints are depicted in graph form in (a). The other inputs: role, processing time, and priority for each job are tabulated in (b). The output consisting of the start and completion time of each job are given in (b) as well as (c).

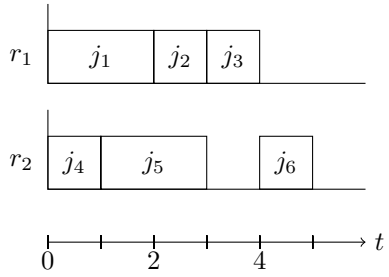
(a) Precedence graph of the scheduling instance; $a \to b$ means that a has to be completed before b can start.



(b) Table containing for each job j its role, processing time p_j, priority w_j, start time s_j, and completion time c_j.

Job	Role	p_j	w_j	s_j	c_j
j_1	r_1	2	2	0	2
j_2	r_1	1	1	2	3
j_3	r_1	1	2	3	4
j_4	r_2	1	1	0	1
j_5	r_2	2	1	1	3
j_6	r_2	1	1	4	5

(c) Gantt-chart of the schedule for roles r_1 and r_2; time is depicted horizontally.



In the next iteration, *decision_moments* contains the time points 1 and 2, so we get $t = 1$. Job j_4 has finished by this time, and its successor, j_5, does not have any other predecessors, meaning that it is now set to be available. Looping over the roles, we find that r_1 is busy, but r_2 is idle. For r_2, only job j_5 is available, giving $s_5 = t = 1$ and $c_5 = s_5 + p_5 = 3$. The moment $c_5 = 3$ is added to *decision_moments*.

Now, we have that *decision_moments* $= \{2, 3\}$, resulting in $t = 2$. At this point, job j_1 finishes, but its successor, j_3, still has an uncompleted predecessor,

j_2, so it does not become available yet. Consequently, for r_1 the algorithm picks j_2, giving $s_2 = 2$ and $c_2 = 3$, whereas r_2 is busy.

In the fourth iteration, *decision_moments* $= \{3\}$, hence we must get $t = 3$. Both job j_2 and j_5 finish at this moment, but only job j_3 becomes available. Role r_1 schedules job j_3, whereas role r_2 is idle, but does not have any available jobs. The fifth iteration gives $t = 4$, when j_3 finishes, making j_6 available for processing. Role r_1 is idle but has no available jobs, but r_2 schedules j_6.

In the last iteration, $t = 5$, but both roles are idle and have no available jobs. As a result, the set of *decision_moments* remains empty, causing Algorithm 2.1 to terminate; all jobs have been scheduled.

Preemption

Algorithm 2.1 can be extended by allowing jobs to be preempted, which means that we can interrupt a job at a certain point in time to continue processing where it had been left off at a later point in time. A job j that is preempted once has two starting and stopping times as a result. Its schedule can be denoted by:

$$(s_{j,1}, c_{j,1}), (s_{j,2}, c_{j,2}),$$

where $(s_{j,i}, c_{j,i})$ is the i^{th} interval during which it is continuously processed. There is no cost or penalty when preempting, so it should hold that $(c_{j,1} - s_{j,1}) + (c_{j,2} - s_{j,2}) = p_j$. Moreover, note that a job can be preempted any number of times, but its successors can only become available after the job has been entirely completed, i.e., the sum of intervals on which the job has been scheduled equals its processing time.

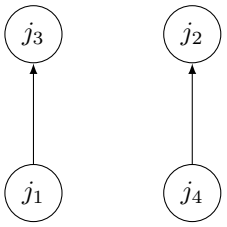
Incorporating preemption into the scheduler means that at every decision moment, we can also look at all the non-idle roles and check if the job that they have scheduled at that moment is still the available job with the highest priority. If it is, then we do not need to do anything. Otherwise, we preempt the currently scheduled job by changing its latest completion time to the current point in time and subsequently we schedule the higher priority job in its entirety from this moment until it is completed (while adding this new completion time to *decision_moments*).

Figure 2.2 gives an example of how preemption incorporated in Algorithm 2.1 works. At $t = 0$, roles r_1 and r_2 schedule j_1 and j_4 respectively, since they are the only available jobs. At $t = 2$, however, j_4 finishes, making j_2 available. Because j_2 has a higher priority $w_2 = 2$ than j_1 with $w_1 = 1$, we preempt j_1 and schedule j_2 on r_1. Only after j_2 has finished, we can continue scheduling j_1 for its remaining processing time and when j_1 has finished entirely, j_3 is made available for scheduling.

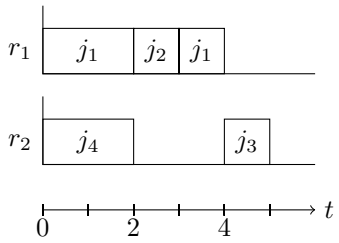
We remark that preemption provides a degree of flexibility to our approach of scheduling that results in "better" schedules. For example in the instance of Figure 2.2, preemption allows the high priority job j_2 to be scheduled as soon as possible. If we had wanted to do that without preemption, we would have had to know in advance at $t = 0$ that j_2 was going to be available at $t = 2$ and we should have waited with scheduling j_1 until $t = 3$. This also causes that j_1 is only finished at $c_1 = 6$. Preemption allows us to complete j_1 already by $c_1 = 4$ without having to delay j_2.

Figure 2.2: Example of a scheduling instance on which we applied Algorithm 2.1 *with* preemption. Job j_1 is preempted at $t = 2$ to make place for j_2, and continued processing at $t = 3$.

(a) Precedence graph of the scheduling instance.



(b) Gantt-chart of the schedule for roles r_1 and r_2.



(c) Table containing for each job its role, processing time, priority, and start and completion times for every scheduled interval.

Job	Role	p_j	w_j	$(s_{j,1}, c_{j,1})$	$(s_{j,2}, c_{j,2})$
j_1	r_1	3	1	(0,2)	(3,4)
j_2	r_1	1	2	(2,3)	-
j_3	r_2	1	1	(4,5)	-
j_4	r_2	2	1	(0,2)	-

Start and release dates

In Progressive Planning, jobs can have a release date, before which they cannot be scheduled. Also roles can have a start date, which entails that all its jobs can only start after a certain point in time. As mentioned before, these can both be modeled with a release date r_j on job j, where r_j is the maximum of a job's release date and its role's start date. To incorporate release dates into Algorithm 2.1, we add all the release dates to *decision_moments* in the initialization and set all jobs with release date greater than zero to unavailable. Then, at each decision point in time, we check if there are jobs with a release date equal to this point in time. For all such jobs, we make them available if their predecessors have been completed.

Moreover, we should take care that the jobs remain unavailable before their release dates. This conveys that in checking for a finished job's successors, we should incorporate a check on whether the current point in time is greater than the successor's release date.

Time complexity

Concerning the time complexity of Algorithm 2.1, Theorem 2.1 shows that it is $O(n \log n + |E|)$ even if preemption and release dates are incorporated. This means that for typical sparse precedence graphs, where $|E| \sim n$, the scheduler runs in $O(n \log n)$.

Theorem 2.1 (Scheduling time complexity)**.** *Algorithm 2.1 extended with preemption and release dates has a running time of* $O(n \log n + |E|)$*, where* n *is the*

number of jobs and $|E|$ the number of edges in the precedence graph.

Proof. First consider Algorithm 2.1 on its own. At each decision moment, it schedules at least one job, hence the number of outer iterations is at most n. Consequently, the set *decision_moments* has at most n insertions and n removals of the minimum value. We do this by maintaining a priority queue of the decision moments, which if implemented as a binary heap, can be done in $O(n \log n)$. A binary heap is a data structure that keeps the data sorted by size, and as such it handles insertions in $O(\log n)$, as well as removals of the maximum value in $O(\log n)$, where n is the number of elements in the heap.

Line 4 of Algorithm 2.1 runs once for every job that is completed, which is again n times as every job finishes only once. For every job it checks all its successors, which over all finished jobs amounts up to a total of $|E|$ operations, because a check is done over every edge. At every check, the finished job needs to remove itself from the set of predecessors of the specific successor and check whether this set is empty, which can be done in $O(1)$.

If we maintain a set of idle roles with available jobs, then throughout the algorithm, there are at most n additions and n removals from this set. Accordingly, in the entire algorithm, we visit n times an idle role with available jobs, each time resulting in a new scheduled job. If for each role, we maintain a priority queue of available jobs sorted by their priorities, then for role r_i there are n_i additions and n_i removals from this priority queue, where n_i is the number of jobs that r_i has. In an efficient implementation this costs $O(n_i \log n_i)$, which for m roles boils down to $O(n \log n)$. In conclusion, Algorithm 2.1 runs in $O(n \log n + |E|)$ without extensions.

It is clear that the introduction of release dates into Algorithm 2.1 does not worsen its time complexity, seeing that it is a simple $O(1)$ check per node to see if it has been released yet. Also, the number of decision moments increases by at most n, as that is the maximum number of release dates.

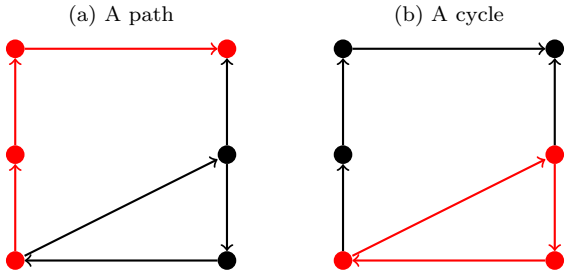
Incorporating preemption on the other hand, requires a closer analysis. What triggers a preemption? It can only be that some job that was not previously available has become available during the interval that the preempted job was scheduled. This can either occur through a release date, or a job whose predecessors are all completed. However, only n jobs become available at some point during the algorithm, so the number of preemptions is bounded by n. Except for the costs of rescheduling, which is $O(1)$ per preemption, there are additional costs incurred by the fact that in line 5 of Algorithm 2.1 we need to loop over non-idle roles as well. Actually, we need to consider roles that are idle and have available jobs, and non-idle roles that have jobs that became available in the same iteration. As we just argued, these newly available jobs are the ones that can trigger a preemption. Moreover, a set containing roles with newly available jobs can be maintained in $O(n)$ over the entire algorithm. Finding the available job with the highest priority still costs $O(\log n)$, but is done more often because of preemption. However, there are at most n preemptions, which means that the time complexity of the entire algorithm is kept at $O(n \log n + |E|)$. □

2.2.2 Propagation of the decision measure

The quality of the scheduler in Algorithm 2.1 is in the end determined by the measure of priority for each job. By picking the right measure to decide which job to schedule before which other jobs, it is possible to optimize any of the objective functions mentioned in Section 2.1. As it is equivalent to solving the scheduling problem to optimality, finding these optimal measures is NP-hard. Moreover, we wish to take all possible criteria for the objective function into consideration, which makes it hard to formulate an objective function. We dismiss this idea of finding the optimum or an approximation of some objective function and instead we focus on finding a *decision measure* that "makes sense".

Consider the precedence graph $G = (V, E)$ of the scheduling problem. As mentioned before, G is directed, which means that for an edge $(u, v) \in E$, we say that u precedes v (in our case this means that job u has to finish before v can start). It would be paradoxical if $(v, u) \in E$ as well, because then u precedes v and v precedes u at the same time; therefore, G has to be acyclic. A definition useful for formalizing this, is that of a path in a directed graph, which is a sequence of vertices where for every two subsequent vertices $u, v \in V$ there exists the edge $(u, v) \in E$. Then in a directed graph, a cycle is a path of which the first and last vertex are the same, see Figure 2.3. Unsurprisingly, a directed graph is acyclic if it has no cycles.

Figure 2.3: Examples of a path and a path that is also a cycle on the same directed graph. Red vertices and edges are part of the path.



Suppose a certain job $v \in V$ in the precedence graph has a high priority; it should be finished as soon as possible. Before v can be scheduled, however, all its direct and indirect predecessors need to be completed. These are all the vertices that have a path in the directed graph to v. If we give all these predecessors the same priority as v, then from $t = 0$ there will always be a job from this set scheduled (assuming no release dates). Consequently, v can be expected to be scheduled much sooner than without this propagation of priority.

We take three factors into account in Progressive Planning when deciding which job to schedule. These are all in a different way dependent on a job's direct and indirect successors. The measure of priority of a job j is dependent on its:

i) own deadline or a deadline of its direct successors. If one of its direct successors i has a deadline d_i, then the job's deadline becomes $d_j = \min\{d_j, d_i - p_i\}$,

which means that the job should be completed in time for its successors to be able to complete in time;

ii) number of roles that are dependent on j, because a job becomes more critical when it is at the basis of much teamwork;

iii) amount of work that is dependent on j, because when a large project depends on a job, it is more important.

For each job j, we take all these factors together and combine them in some sensible way to obtain a general decision measure $w_j \in \mathbb{R}$. This measure can subsequently be used as the priority in the scheduler in Algorithm 2.1 to decide which job to schedule at every point in time.

In order to set a job's decision measure we only consider its direct successors rather than inspecting all of its direct and indirect successors. We can still carry on the information of all successors of each job by having a job inspecting its successors only when those successors have already updated their decision measures. We developed Algorithm 2.2, which takes care of this. It is similar to the topological sorting algorithm attributed to Kahn [19], except that it does not explicitly create a topological order. It does, in fact, visit the vertices in reverse topological order, which ensures that a vertex is only visited after its successors have been visited.

Algorithm 2.2 Propagation of the decision measure

Input: Directed acyclic graph $G = (V, E)$ with vertex weights $w(v)$ for $v \in V$ and the weight combining function f.
Output: The set of vertices with weights updated according to the weights of their successors.

```
1: S := {u ∈ V : ∄ (u, v) ∈ E}
2: while S ≠ ∅ do
3:    pick a v ∈ S
4:    S := S \ {v}
5:    for each u ∈ V : (u, v) ∈ E do
6:       w(u) := f(u, v)
7:       E := E \ {(u, v)}
8:       if ∄ t ∈ V : (u, t) ∈ E then
9:          S := S ∪ {u}
```

Algorithm 2.2 defines a set of sink nodes S, which consists of those vertices that do not have outgoing edges. Iteratively, the algorithm pops a vertex v from S and visits its direct predecessors, removing v as a successor. In this way, once all the successors of a vertex have been visited, it has become a sink node and is added to S. When visiting a predecessor u of vertex v, we add the decision information of v to the weight of u in a specific way as captured by the function $f(u, v)$. The weight $w(u)$ is a multidimensional variable that catches the deadline, amount of work and dependent roles, and $w(u) = f(u, v)$ adds the part of this information from $w(v)$ needed by u.

Theorem 2.2 states that we can propagate in linear time, as long as our function f runs in time independent of the problem size. This is intuitive, as the

algorithm visits every edge once and every vertex at least once, but not more times than it has outgoing edges. The variables used to create the decision measure for each vertex change continuously, so we propagate the decision measure with Algorithm 2.2 every time before we schedule with Algorithm 2.1. Taking these together, the entire scheduling procedure still runs in $O(n \log n + |E|)$.

Theorem 2.2 (Topological sorting, Kahn [19])**.** *If we assume that $f(u, v)$ can be executed in $O(1)$, then Algorithm 2.2 has a running time of $O(n + |E|)$.*

2.2.3 Numerical tests

Implementations of Algorithms 2.1 and 2.2 are now incorporated in the Progressive Planning application. We generated data that follows the structure enforced by Progressive Planning. We measured averages for all the variables, assumed their distributions where necessary and created the data in the following steps. For each role, we assumed a geometric distribution for the total number of deliverables, where the geometric distribution is supported on $\{0, 1, 2, \dots\}$. For each deliverable we created a number of activities by also sampling a geometric distribution. We took a uniform distribution for the number of sub deliverables per deliverable, where the uniform distribution has a range from zero to two times the mean. This way, we created (sub) deliverables as long as we did not reach the total number of deliverables on the role. Only a small percentage of sub deliverables is owned by other roles than their parent deliverable. Doing this for a total number of roles, we obtain a collection of many deliverable trees filled with activities. At this stage, we randomly picked pairs of activities and deliverables and created a dependency between them if this would not create a cycle. Lastly, we sampled processing times, release dates, and deadlines from a continuous uniform distribution, without regard of the feasibility of the time constraints.

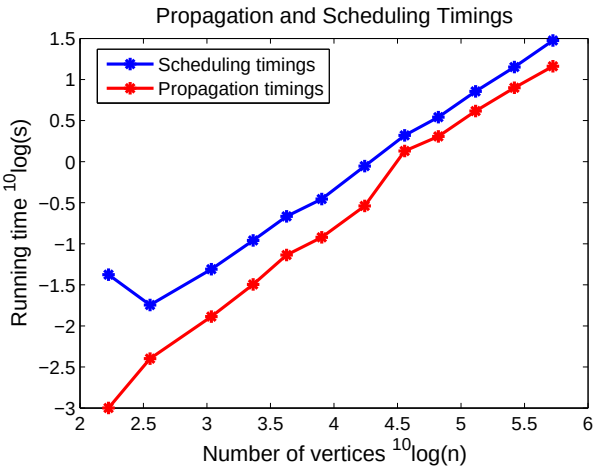
Although the choices for distributions are quite arbitrary and careless, from the running times of Algorithms 2.1 and 2.2 we can assume that what really matters are the number of vertices and edges. The number of edges in the precedence graph is determined by the number of dependencies, which we have depended on the measured average number of dependencies per activity. In this way, we obtained an average of 1.2 outgoing edges per vertex.

Figure 2.4 depicts the running times of the implementations of the scheduling and propagation algorithms. For both algorithms, we have fitted the logarithm of the timings to the logarithm of the number of vertices, which gives the exponent α when assuming that the running time is of $O(n^\alpha)$. We obtained $\alpha = 1.03$ for scheduling and $\alpha = 1.19$ for propagation. Both algorithms show a slightly larger than linear order running time, which could be expected as we picked $|E| \sim n$. This gives $O(n)$ running time for Algorithm 2.2 (propagation) and $O(n \log n)$ for Algorithm 2.1 (scheduling). Therefore, it is also remarkable that scheduling shows a lower increase in running time than propagation. However, this discrepancy is likely due to imperfect implementation or measurement errors.

2.3 Connected components

In Progressive Planning, it is necessary to frequently update a user's schedule. The user can make changes and even if he does not, the passing of time requires

Figure 2.4: Log-log plot of the time performance in seconds of implementations of Algorithm 2.1 (scheduling) extended with release dates and preemption and Algorithm 2.2 (propagation). The timings are based on randomly created scheduling instances consisting of different numbers of jobs / vertices. Linear fitting of the logarithms of the scheduling timings (excluding the first data point) gives a slope of 1.03, and of the propagation timings 1.19.



that the schedule is updated as release dates and deadlines come closer. However, the data is largely connected and the schedule of a single person depends on the schedules of many other people. If we can determine precisely what data influences or can influence which other data, we could reduce the scheduling problem when we are required to give output for only a single or a few users.

One way to achieve this is to find the connected components in the graph that contains all links of jobs influencing each other directly. The task dependencies are given in the precedence graph, but influence stretches further than the links in this graph. Two jobs that are in completely different projects and precedence graphs can influence each other's schedules because they have to be performed by the same role which cannot work on both jobs simultaneously. In project management, such links are called the resource dependencies.

Now, take the precedence graph and add to the graph the roles as vertices and also add edges from each role to each of its jobs. Jobs and roles from different connected components in this graph, see Definition 2.1, definitely do not influence each other's schedules.

Definition 2.1 (Connected component)**.** *A connected component on an undirected graph $G = (V, E)$ is a set $C \subseteq V$ such that:*

i) for each $u, v \in C$ there exists a path from u to v;

ii) for each $u \in C$ and $v \in V \setminus C$, we have that $(u, v) \notin E$.

We present Algorithm 2.3, which gives a procedure for finding the connected components in our case. It is based on a breadth-first search and hence runs in linear time.

Algorithm 2.3 Connected components

Input: Undirected graph $G = (V, E)$, set of roles $R = \{r_1, r_2, \ldots, r_m\}$, and for each job $v \in V$ its corresponding role $role(v)$

Output: Sets $C_1, C_2, \ldots$ of connected components containing jobs and their respective roles

```
1:  V := V ∪ {r_1, r_2, ..., r_m}
2:  E := E ∪ {(v, role(v)) : v ∈ V}
3:  i := 1
4:  while V ≠ ∅ do
5:      pick a v ∈ V
6:      N := {v}
7:      C_i := ∅
8:      while N ≠ ∅ do
9:          V := V \ N
10:         C_i := C_i ∪ N
11:         N := {u ∈ V : ∃ (u, w) ∈ E such that w ∈ N}
12:     i := i + 1
```

Algorithm 2.3 starts with an undirected version of the precedence graph and adds the resource dependencies. It picks a vertex from V, explores all direct neighbors of v, and recursively finds all other vertices that are indirectly connected to v. These form the first connected component C_1 and the algorithm repeats this procedure until all vertices are assigned to a connected component.

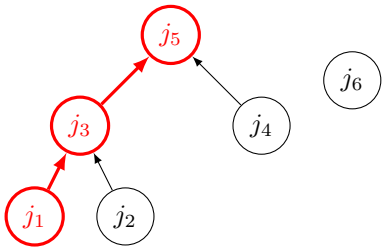
Now, whenever we need the schedule for only a specific role, we can look up in which connected component it is and schedule all the jobs in that connected component. Leaving out the cost of computing the connected components, this can give a huge speedup in scheduling time for an individual user as connected components can be quite small. Connected components are not the only way of determining which jobs influence which others. However, it is a straightforward way and we will describe in Section 2.5.2 how we maintain them efficiently when new links are made.

2.4 Critical path and critical chain analysis

The critical path is a concept in scheduling theory that defines a lower bound on the makespan (latest completion time) of a schedule. Brucker [9] defines the critical path as the succession of jobs on a path in the precedence graph such that the sum of their processing times is maximum. An example of such a critical path is given in Figure 2.5. Regardless of which roles perform these jobs, it is clear that the schedule cannot be completed before $t = p_1 + p_3 + p_5 = 5$, as the jobs on the critical path have to be processed subsequently. Therefore, the lower bound on the schedule's makespan as given by the critical path can be used to assess how far the schedule optimizes the makespan. For example, if the schedule of the instance in Figure 2.5 is completed at $t = 5$, then we know for sure that it has an optimal makespan.

This notion of a single critical path for the entire problem instance is of limited

Figure 2.5: Precedence graph of six jobs and a table of their processing times. The critical path in this example is depicted in red and consists of the jobs j_1, j_3 and j_5.



Job	p_j
j_1	2
j_2	1
j_3	2
j_4	3
j_5	1
j_6	2

use in Progressive Planning, because project managers would like to know the critical path for their own projects. Therefore, we define the critical path of a job to be the path through the precedence graph to the job such that the sum of the processing times of the jobs along the path is maximum. As such, the critical path of a job j gives a lower bound on its completion time c_j.

2.4.1 Computation of the critical path

The critical path of a vertex is the longest path in the precedence graph where the incoming edges of each vertex have a weight equal to the processing time of that vertex. The critical paths to all vertices can be calculated at once in linear time using the longest path algorithm of Evans and Minieka [14], which is a modification of Dijkstra's well-known shortest path algorithm [11].

This approach is followed in Algorithm 2.4, which visits the vertices in topological order. For each vertex v, it finds the length $\ell(v)$ of the critical path to v and the first predecessor in the critical path $pred(v)$. We initialize $pred(v)$ to be *null* and $\ell(v)$ to $r_v + p_v$, because a job is certainly not going to be finished before it is released and processed. Surely, these are the correct values for the critical paths of the vertices that do not have any predecessors. In fact, the algorithm maintains the set S of unvisited vertices without incoming edges, for which we have already found the critical path. In the outer loop of the algorithm, we visit a vertex u from S, removing it from the set. Then it looks at each successor v of u and checks if the critical path to v through u is longer than the current critical path to v. This is possible to establish, because if the first predecessor of v in its critical path is u, then the critical path to v is the same as to u, but extended by v. If this is the case, the critical path values are updated to $\ell(v) = \ell(u) + p_v$ and $pred(v) = u$. Finally, the algorithm removes the edge (u, v) and if this means that v does not have any more predecessors, then we have found its critical path and we can add it to S.

When Algorithm 2.4 terminates, the critical path to each vertex v is known in the sense that $\ell(v)$ is its length and the vertices in the path are in reverse order: $[v, pred(v), pred(pred(v)), pred(pred(pred(v))), \dots]$. As every vertex and every edge is visited exactly once and all operations can be executed in $O(1)$, the algorithm runs in $O(n + |E|)$ time.

Algorithm 2.4 Critical path to every vertex

Input: Directed acyclic graph $G = (V, E)$, processing time p_v for each $v \in V$
Output: The critical path length $\ell(v)$ and first predecessor *pred*(v) in the critical path for each $v \in V$

```
1: for all v ∈ V do
2:    ℓ(v) := r_v + p_v
3:    pred(v) := null
4: S := {v ∈ V : ∄ (u, v) ∈ E}
5: while S ≠ ∅ do
6:    pick a u ∈ S
7:    S := S \ {u}
8:    for all v ∈ V : (u, v) ∈ E do
9:       if ℓ(u) + p_v > ℓ(v) then
10:         ℓ(v) := ℓ(u) + p_v
11:         pred(v) := u
12:      E := E \ {(u, v)}
13:      if ∄ t ∈ V : (t, v) ∈ E then
14:         S := S ∪ {v}
```

2.4.2 Critical chain

The critical path only gives a lower bound on when a job can be scheduled, as can be seen in Figure 2.5. If in this example all the jobs have to be performed by the same role, then j_5 can only be completed by time $t = p_1 + p_2 + p_3 + p_4 + p_5 = 9$. This is caused by what the management "guru" Goldratt [16] calls the resource constraints. The critical path only takes the precedence constraints in account and does not include the time constraints caused by the fact that jobs of the same role cannot be processed at the same time. Goldratt coined the term critical chain, which is the set of jobs with the largest sum of processing times that have to be executed in sequence in order for the project to finish. In our example, the critical chain for j_5 is composed of the jobs j_1, j_2, j_3, j_4 and j_5 and has length 9.

Although the critical chain provides more useful information than the critical path, it is not straightforward which jobs belong to it. A useful way to think of the critical chain is that it is the set of those jobs that increase the optimal schedule's makespan by dt when the job's processing time is increased by dt. As such, they are critical in finishing the schedule on time. Since we have developed a heuristic scheduling method, we also estimate the critical chain instead of determining it exactly. We do this by adding edges between jobs that have the same role in the order in which they are scheduled. In a way, this is a literal interpretation of Goldratt arguing we should take into account the resource constraints. Now, our critical chain is just the longest path through the graph as calculated by Algorithm 2.4.

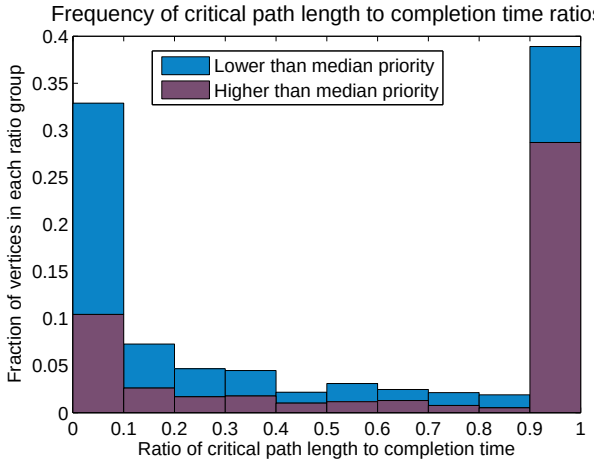
2.4.3 Critical path experiments

We have implemented Algorithm 2.4 for Progressive Planning and did measurements on real-life data, for which the results are depicted in Figure 2.6. The figure

plots the frequency of the approximation ratio of the jobs' critical path lengths to their completion times in a histogram. The plot has an interesting shape, with most of the mass concentrated at the extremes: around 0 and 1. If a job's ratio of critical path length to completion time is exactly 1, then it is sure that the job has been scheduled as early as possible. Otherwise, it could still be that the job is scheduled as early as possible, but we only know its ratio to the lower bound given by the critical path.

We make two main observations from Figure 2.6. First, most of the jobs are either scheduled close to their earliest possible start time, or they are scheduled exceptionally late compared to their critical path. The second observation is that higher priority jobs seem to be scheduled better compared to their critical path than lower priority jobs. A possible factor in explaining the high mass close to zero in Figure 2.6, is that there are many loose activities in Progressive Planning that are not part of a project and therefore have low priority. Mostly, these jobs get scheduled at the end of a person's schedule, whereas their critical path consists only of their own processing time.

Figure 2.6: Histogram of the ratio of critical path length to the completion time of each job. This ratio, $\ell(v)/c_v$ for each $v \in V$ is counted and normalized for ten buckets on the x-axis. Jobs are divided into two groups: those with a priority (decision measure) higher than the median priority, and those with a lower priority. Data is taken from about 2500 jobs from the Progressive Planning database.



2.5 Dynamic Algorithms

Because changes are so frequent in the data of Progressive Planning, dynamic algorithms can be beneficial in reducing the computing load. These are algorithms that exploit the solution of the previous problem to compute a new solution when changes have occurred, rather than recomputing the entire problem from scratch. The first example of a dynamic algorithm is given in Section 2.5.1, which exploits the fact that the precedence graph is directed and acyclic to determine whether

it is still directed and acyclic when new edges are added. In the Sections 2.5.2, 2.5.3 and 2.5.4, we discuss dynamic approaches to the computation of connected components, the decision measure, and the schedule, respectively.

2.5.1 Cycle checking

A cycle in the precedence graph would mean that a certain job a has to be finished before b can start, but that also b has to finish before a. In order to prevent this paradox, the user should not be able to make circular dependencies in the data. Algorithm 2.5 gives an efficient procedure to check whether adding an edge creates a cycle in the graph. The user enters changes one by one, so the algorithm checks for the addition of a single edge only. The removal of an edge can never cause the graph to become cyclic, so it is always permitted.

Algorithm 2.5 assumes that the graph G is acyclic before we try to add the edge (u, v). Therefore it can be argued that it is a dynamic algorithm, because it does not check for other parts of the graph unaffected by the addition of (u, v). The algorithm finds the set S of all the direct and indirect successors of v through a directed breadth-first search. If u is in that set, then the addition of (u, v) would form a cycle, because u is a (indirect) successor of v and the edge (u, v) would also make u a predecessor of v. The breadth-first search is done by iteratively adding to S the unvisited successors N of the vertices already in S. If it appeared that $u \notin S$, then adding (u, v) to the graph does not create a cycle and is allowed.

The cycle checking algorithm runs in $O(n + |E|)$, because in the worst case every edge and vertex is visited once. Another method would be to maintain a reachability matrix for the precedence graph, the so-called transitive closure. From this matrix, we can infer in $O(1)$ whether a certain vertex can be reached from another vertex, and thus whether adding an edge between the vertices would create a cycle. However, after adding the edge, the reachability matrix needs to be updated. The fastest dynamic algorithm for this by Demetrescu and Italiano [10] updates the matrix in $O(n^2)$, where recomputing the matrix from scratch is $O(n \cdot |E|)$. Comparing our algorithm with that of Demetrescu and Italiano, if the user adds an edge that creates a cycle, we handle it in $O(n+|E|)$ and Demetrescu and Italiano in $O(1)$. On the other hand, if the edge does not create a cycle and is added to the graph, we still handle it in $O(n + |E|)$ against the $O(n^2)$ using the reachability matrix.

Algorithm 2.5 Cycle checking

Input: Directed acyclic graph $G = (V, E)$ and a new edge (u, v).
Output: Boolean on whether adding (u, v) would create a cycle in G.

```
1: S := {a ∈ V : (v, a) ∈ E}
2: N := S
3: while N ≠ ∅ do
4:    if u ∈ S then
5:       return False
6:    N := {x ∈ V \ S : ∃ (x, y) ∈ E such that y ∈ N}
7:    S := S ∪ N
8: return True
```

2.5.2 Connected components

Rather than recomputing the connected components every time we schedule, we could maintain them dynamically while the connection graph used in Section 2.3 changes. It is very straightforward to do so: if an edge (u, v) is added to that graph and $u \in C_i$ and $v \in C_j$ where $i \neq j$, then we merge the two connected components into one. Otherwise, if an edge (u, v) is removed, we know that u and v are in the same connected component C_i. We do a breadth-first search from u and as soon as we find v, we know that C_i is still entirely connected. If the breadth-first search terminates without finding v, then C_i has to split into two parts. The vertices visited during the search constitute a new connected component C_j and we update C_i by removing these vertices: $C_i = C_i \setminus C_j$.

Although these breadth-first searches when an edge is deleted are quite fast, their running time is dependent on the size of the connected component. Therefore, they are not as efficient as the $O(1)$ operation for checking if two vertices are in different components when an edge is added. Indeed, we found it to be more effective to store the connected components, but only update them incrementally; we merge two connected components as soon as an edge between them is added, but we do not check or separate components when edges are removed. This is because the resulting over-sized connected components do not interfere with the schedule's consistency, but they are more efficient to maintain. Also, the chance of two connected components to come together after they have split is considerable; a person often does more projects with the same people. Then we have the "maintenance" script of Algorithm 2.3 which could recompute the connected components in linear time whenever the application server is idle.

2.5.3 Propagation

Depending on how the scheduler runs dynamically, we would also need the decision measure of each job to be maintained dynamically; otherwise, we would have to run the propagation algorithm every time we adjust the schedule. This is not hard to do, but we first analyze the impact of dynamic propagation. Every time the user updates a job's processing time, predecessors, successors or deadline, we would need to update the decision measure on that job and all its direct and indirect predecessors accordingly.

Assume that each update requires an average of k vertices to be visited in order to have their decision measure updated. In that case, the question whether maintaining the priority value dynamically is efficient, depends on how many changes occur to the data before it is needed again for scheduling. If the data is rescheduled after every update, then indeed it would be more efficient to handle the updates dynamically (as k is bounded by n). However, if there are on average more than $\frac{n}{k}$ updates before we reschedule, we might as well run the static Algorithm 2.2 right before we schedule.

2.5.4 Scheduling and robustness

The volatile environment of our scheduling problem raises the question of how stable the resulting schedules are. It is not always preferable that there are drastic changes in the scheduled order of jobs between two subsequent schedules.

The nature of Progressive Planning, however, is that the schedule always changes, because users update their spent and estimated processing times frequently. Even without those updates, a schedule becomes irrelevant as time passes and release dates and deadlines come closer, because if no time is clocked by the users, there is no progress on the completion of the jobs. On the other hand, the power of frequently renewing the schedule is that at any moment in time, there is always the best and most relevant schedule available. This gives users a unique advantage over traditional static schedules which force people to stick with plans that seemed good at some point in the past. It is important to take these considerations into account when discussing dynamic schedules.

In the literature, the term robustness is used to describe the volatility of a schedule as it incorporates the changes occurring in the problem instance. However, robustness is not measured or even defined easily, as Pinedo [25] points out. It measures the change in the objective function of the schedule as caused by, for example, a delay in the processing time of a certain job. It can also incorporate the change in completion times $c_j - c'_j$ for each job j from c_j in the original schedule to c'_j in the updated schedule. Nair et al. [22] assume known distributions for the processing time of each job and present a thorough analysis on the robustness of the total schedule length. They conclude that there is a trade-off between robustness and optimality of a schedule. Robustness is also related to the critical path, because a schedule with a makespan equal to the length of its critical path or critical chain will surely be delayed when one of the jobs on the critical path or chain takes more time than estimated, as explained in Section 2.4.

The naive solution to our dynamic scheduling problem is computing the entire schedule from scratch every time that it is needed. Indeed, because of the difficult trade-off between optimality and robustness, we have not yet implemented a more efficient solution. The complete renewal of the schedule at every event of change, means that we prioritize optimality entirely above robustness. Here, by optimality we mean a schedule that follows the order dictated by the decision measure. In this light, we discuss how robustness could in fact be the key to a more efficient dynamic scheduling procedure.

It is difficult to maintain a schedule that is similar to the output of the scheduler in Algorithm 2.1, because a change in the data of a job means that the decision measure of that job changes. This change in turn, is propagated to the jobs available at the starting time of the schedule. Such a cascade of changes does not necessarily propose a problem; if the order of jobs by their decision measure does not change, the schedule can remain the same. On the other hand, if the order does change, then the schedule could have to switch the execution of certain jobs. Consequently, we have to reschedule all the next jobs of the involved roles, while taking dependencies to jobs of other roles into account as well. By this mechanism, a small change in one job could lead to rescheduling many other jobs. Also, it is hard to determine beforehand which jobs are going to be affected by a change.

Confronted by these difficulties, an increased focus on robustness forms a good excuse to adjust the schedule dynamically in a simpler way. The first, perhaps radical in the view of traditional scheduling, way is to not modify the schedule at all if the change is small and relatively far in the future. This would result in an invalid schedule, which can be problematic depending on how it is used. In most

cases, however, it is likely that more significant changes will occur, diminishing the problem of the wrong schedule.

The second, more useful way is to reschedule only the jobs of the role on which a change a occurred. We deal with the precedence constraints on jobs on other roles as follows: if job j_1 on another role precedes job j_2 on the role that we are rescheduling, we take $r'_2 = c_1$, where c_1 is the completion time of j_1 in the current schedule and r'_2 is the release date that we use for j_2 in the new schedule to ensure that it is scheduled after j_1. Conversely, if j_2 precedes j_1, then we set the new deadline of j_2 to the start time of j_1 in the current schedule. Perhaps, it is not possible to satisfy all the new deadlines imposed in this way, but we can do our best. In this way, we ensure complete robustness for the roles that did not incur changes, but we can sacrifice the validity of the schedule as well as opportunities to schedule jobs on other roles earlier. One advantage of rescheduling only for one role is that it is a clear and logical way of drastically limiting the scope of jobs to reschedule[2].

The above two methods do not only trade optimality for robustness, but can also invalidate the schedule. However, we can maintain a measure for how invalid the schedule has become. For example, how many precedence constraints are not satisfied and by how much time. If this measure passes a certain threshold, it triggers a complete rescheduling.

Although we did not implement a dynamic scheduling algorithm, our highest-priority-first scheduler is fast and therefore well suited for a dynamic environment. Additionally, the scheduling is sped up further by splitting the precedence graph in connected components. In the next chapter, we develop an approximate matching algorithm that does exploit the dynamism of the graph.

[2]In practice, this has the second advantage that all this information is available for the user in the web application, which means that the computation can be moved from the server to the client.

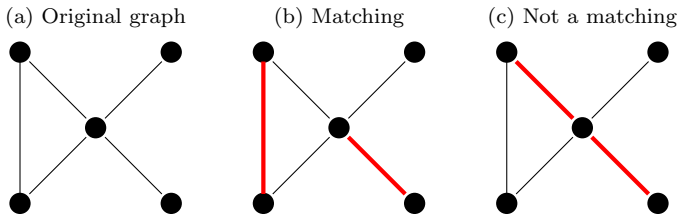
Chapter 3

Dynamic Matching

3.1 Introduction

On an undirected graph $G = (V, E)$, a subset of edges $M \subseteq E$ is called a matching if no edges in M have vertices in common. In other words, an edge in M matches a vertex in G to an adjacent vertex, such that no vertex is matched to more than one other vertex. There are two common problems; the first is to construct a matching with the highest number of matched vertices or edges, called a maximum cardinality matching, and the second, to construct a matching with the largest sum of edge weights. Note that the first is a special case of the second, namely where all edge weights are set equal.

Figure 3.1: Example of a matching. Left is the graph G, in the middle the subset M of red edges forms a matching on G. On the right, the subset of red edges is not a matching, because the middle vertex is connected to two edges in the subset.



A simple practical example of the matching problem is a group of people looking for a dating partner on an online dating service. The people are the nodes in the graph and they have an edge to each potential partner, for example, a man could have edges to all women in his same age category. Then, the dating service should suggest a partner for everyone, but not more than one partner per person; this makes the set of suggestions a matching. The dating service could make as many suggestions as possible, which is a maximum cardinality matching. Otherwise, if the weight of an edge represents the degree to which the two persons are suited as partners, the dating service could suggest the overall highest quality of partners, which is the maximum weighted matching.

A similar application of matching is finding the overall best combinations or

organ donors and recipients in organ donation [28]. In scientific computing, however, matching is applied most often for graph coarsening. For example in graph partitioning [2] [27], which in a simple form is dividing the vertices of a graph into two sets of somewhat similar size while minimizing the edge cut, i.e., the sum of weights of edges that connect vertices from the two parts. Graph partitioning is an NP-hard problem, but matching is not. Therefore, in applications, the graph is coarsened by contracting the pairs of nodes that are matched together. Such a step reduces the number of nodes by ideally a half of the original number, and is therefore repeated until the graph reduces to a certain size on which the graph partitioning algorithm can run quickly. Lastly, the solution on the coarsened graph can be projected onto the original graph. By matching edges with a high weight and contracting the adjacent nodes, it is prevented that these nodes end up in the different parts of the partition and subsequently that the high-weight edge ends up in the edge cut.

In such applications, the graphs can be very large, which requires the matching algorithms to have a small time complexity. Gabow has presented an algorithm to optimally solve the maximum weighted matching problem in $O(|V| \cdot |E| + |V|^2 \log |V|)$ [15]. However, this time complexity is generally too high for practical purposes, so recently, much attention has been devoted to linear-time approximation algorithms and distributing those for parallel computation. Still, there are few efforts to efficiently deal with dynamic matching problems, those that require maintaining matchings on graphs that change gradually over time. Specifically, no distributed procedures for such dynamic matching problems are known to us today.

This chapter will first outline basic matching theory in Section 3.2.1. Then, in Section 3.2.2, two sequential approximation algorithms for the maximum weighted matching problem are presented in order to develop useful insights. Third, we will describe the parallel algorithm that is based on similar heuristics in Section 3.2.3. It is at the basis of our approach to the dynamic problem, for which we developed an efficient algorithm when limited to dynamic graphs on which edges are only added and not removed or updated. This algorithm and its theoretical basis are the topic of Section 3.3.

3.2 Static matching

3.2.1 Optimal matching

We have a graph $G = (V, E)$ where V is the set of nodes or vertices and E the set of edges. An edge $e \in E$ represents a connection between two nodes in V and can also be written as $e = (a, b)$ with $a, b \in V$, meaning that vertices a and b are connected by e. The edges are weighted, defined by the weight function $w : E \to \mathbb{R}^+$ such that $w(e), e \in E$ or $w(a, b), (a, b) \in E$ is the weight of edge $e = (a, b)$. For a set of edges $M \subseteq E$, their aggregate weight is given by $w(M) = \sum_{e \in M} w(e)$. We start by defining the concepts already mentioned in the previous section.

Definition 3.1. *Let $M \subseteq E$ be a subset of edges of the graph $G = (V, E)$. If for each edge $(a, b) \in M$, we have that $(a, x) \in M \implies x = b$ and $(b, x) \in M \implies$*

$x = a$, then M does not contain edges with common vertices and we call M a matching on G. Moreover,

*i) if M is a matching on G with highest number of matched edges, $|M|$, then M is a **maximum cardinality** matching on G;*

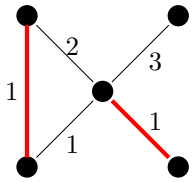
*ii) if M is a matching on G with highest aggregate weight, $w(M)$, then M is a **maximum weighted** matching on G, which we denote here with M^*;*

*iii) if M is a matching on G and adding any edge from $E \setminus M$ to M means that M is no longer a matching, then M is a **maximal** matching.*

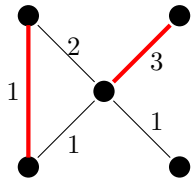
Note that maximum-cardinality and maximum-weighted matchings are always maximal matchings, but that a maximal matching does not need to be either as the example in Figure 3.2 shows.

Figure 3.2: Examples of a maximum cardinality matching (left), a maximum weighted matching (middle) and a maximal matching (right). Note that also the maximum cardinality and maximum weighted matching are maximal matchings. The red edges are in the matching.

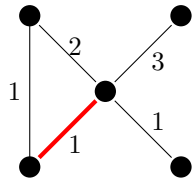
(a) Maximum cardinality matching



(b) Maximum weighted matching



(c) Maximal matching



An important concept in matching theory is that of alternating paths and cycles, given in Definition 3.2.

Definition 3.2 (Alternating path and cycle)**.** *For a matching M on the graph $G = (V, E)$, we define that:*

i) an alternating path of M is a simple path p on G, such that for any two adjacent edges in p, one is in $E \setminus M$ and the other is in M.

ii) an alternating cycle of M is a simple cycle p on G, such that for any two adjacent edges in p, one is in $E \setminus M$ and the other is in M.

The following result for maximum-cardinality matchings was first proved by Berge [5]:

Theorem 3.1 (Maximum cardinality matching, Berge [5])**.** *A matching M is a maximum cardinality matching if and only if there exists no odd length alternating path connecting a free vertex to another free vertex.*

Although we do not present a full proof here, it is easy to understand why a matching is not a maximum cardinality matching if there exists an odd length alternating path connecting two free vertices, as follows. Suppose there is such a path p, then it starts and ends with unmatched edges, which means that the

number of unmatched edges in p is one more than the number of matched edges in p. However, since $M' = (M \setminus p) \cup (p \setminus M)$ is also a matching, and $|M'| = |M|+1$, we conclude that M is not a maximum cardinality matching. For weighted matchings, a similar result holds, but we need the notion of augmenting paths and cycles:

Definition 3.3 (Augmenting path and cycle)**.** *On a graph $G = (V, E)$, let the augmented weight of a set $p \subseteq E$ with respect to a matching M be: $w(p \setminus M) - w(p \cap M)$. We distinguish paths and cycles:*

i) Let p be an alternating path p of M for which an end vertex of p is either free, or it is matched by an edge that is in p. If p has positive augmented weight, then it is an augmenting path of M.

ii) An alternating cycle of M with positive augmented weight is an augmenting cycle.

Note that the extra condition for augmenting paths that an end vertex of p is either free, or it is matched by an edge that is in the path, ensures that the path can be flipped. This means that the set $(M \setminus p) \cup (p \setminus M)$ is also matching. Augmenting cycles are always of odd length and are necessarily also flippable.

In the literature, the term augmenting path is sometimes also used for odd length alternating paths connecting free vertices. Here, however, with augmenting paths we refer to weighted matchings, where it is logical to consider those paths and cycles that, when flipped, increase the weight of the matching.

Theorem 3.3 is the generalization of Theorem 3.1 to weighted matchings. Gabow [15] uses this to give an efficient implementation of Edmond's algorithm [12] for finding maximum weighted matchings. The algorithm starts with an empty matching and continues by iteratively flipping the maximum augmenting path or cycle. Gabow proves that this procedure can be done in $O(|V| \cdot |E| + |V|^2 \log |V|)$. However, neither author explicitly states or proves Theorem 3.3, so we cannot present a reference.

In order to proof Theorem 3.3 ourselves, we need to compare a maximum weighted matching with a matching without augmenting paths. For two matchings in general, Pettie and Sanders [24] mention that their symmetric difference consists of disjoint alternating paths and cycles. We have formalized and proved this statement in Lemma 3.2, which we can in turn use to proof Theorem 3.3.

Lemma 3.2 (Symmetric difference between matchings)**.** *Let M_1 and M_2 be two matchings on the graph G. Then their symmetric difference defined by $M_1 \triangle M_2 = (M_1 \setminus M_2) \cup (M_2 \setminus M_1)$, can be decomposed in a union of vertex-disjoint and edge-disjoint alternating paths and cycles which are alternating both with respect to M_1 and to M_2.*

Proof. Consider the subgraph $H = M_1 \triangle M_2$. Every edge in H is also either in M_1, or in M_2. Clearly, two adjacent edges in H cannot be both in M_1, as it would contradict that M_1 is a matching. As a result, of two adjacent edges in H, one is in M_1 and the other is in M_2. By the same logic, there can be no more than two edges in H incident on the same vertex, so each edge is on a simple path or cycle $p \subseteq H$ that is vertex-disjoint, and thus edge-disjoint, of other paths and cycles in $H \setminus p$. Therefore, these simple paths and cycles in H form alternating paths and cycles that are alternating both with respect to M_1 and to M_2. □

Theorem 3.3 (Maximum weighted matching)**.** *A matching M is a maximum weighted matching if and only if there exist no augmenting paths and no augmenting cycles.*

Proof. It is easy to see that a maximum weighted matching M^* has no augmenting paths or cycles, because if it had an augmenting path or cycle p, then the set $M' = (M^* \setminus p) \cup (p \setminus M^*)$ is also a matching and has a higher weight than M^*.

Now, consider the symmetric difference between a maximum weighted matching M^* and a matching M that does not have any augmenting paths or cycles. From Lemma 3.2 we know that $M \triangle M^*$ consists of a set of k vertex-disjoint and edge-disjoint alternating paths and cycles p_i with respect to M. Let $p_i, i = 1, 2, \ldots, k$, be these disjoint alternating paths and cycles. Then amongst other properties, we have that:

$$M \triangle M^* = \bigcup_{i=1}^{k} p_i,$$

$$p_i \cap p_j = \emptyset, \quad \forall i \neq j.$$

We can decompose the weight of the two matchings in that of their common edges, and their edges on the alternating paths and cycles above:

$$w(M) = w(M \cap M^*) + \sum_{i=1}^{k} w(p_i \cap M),$$

$$w(M^*) = w(M \cap M^*) + \sum_{i=1}^{k} w(p_i \cap M^*).$$

Now, because these alternating paths and cycles are flippable and not augmenting to either M or M^*, we get that their weights must be same:

$$w(p_i \cap M) = w(p_i \cap M^*), \quad \forall i \in \{1, 2, \ldots, k\},$$

showing that $w(M) = w(M^*)$, which means M is also a maximum weighted matching and which proves the theorem. □

3.2.2 Approximate matching

The most basic algorithm for approximate matching is the Greedy-algorithm in Algorithm 3.1 described by Avis in [3]. It is a $\frac{1}{2}$-approximation algorithm, meaning that on any graph, the Greedy-algorithm will return a matching that is at least half of the optimal value. It requires sorting the edges by weight, which results in a time complexity of $O(|E| \log |E|)$. The Greedy-algorithm loops over the remaining edges, and adds the edge of highest weight to the matching in each iteration, while removing that edge and all adjacent edges from the graph in order to prevent doubly matched vertices.

The Greedy-algorithm is not well suited for parallelization, because it requires a global sort of all the edges. Preis [27] developed a more local algorithm to obtain a similar matching, although not with the intention of parallel computation, but for the resulting $O(|E|)$ time complexity. It is based on the concept of a locally

Algorithm 3.1 Greedy-Algorithm

Input: Undirected graph (G, E), with edge weights w
Output: Matching M

1: $M := \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: $\quad (a, b) := \underset{(a,b)\in E}{\operatorname{argmax}}\, w(a, b)$
4: $\quad M := M \cup \{(a, b)\}$
5: $\quad E := E \setminus \big\{(x, y) \in E : x \in \{a, b\}\big\}$

heaviest edge, i.e., an edge (a, b) such that $w(a, b) \geq w(x, y)$ whenever we have $x \in \{a, b\}$. The algorithm of Preis is commonly referred to as the LocalMax-algorithm, and is outlined in Algorithm 3.2. Again, the algorithm loops over all edges, this time picking a locally heaviest edge to add to the matching and removing all adjacent edges. Preis finds consecutive locally heaviest edges without visiting an edge twice, resulting in linear running time. The LocalMax-algorithm also has a $\frac{1}{2}$-approximation quality, which is given in Theorem 3.4 below. Although Preis [27] proves Theorem 3.4, a more simple proof would be identical to our proof of the approximation quality of the Greedy algorithm, which we present in Theorem 3.6 in Section 3.3.2.

Algorithm 3.2 LocalMax-Algorithm, by Preis [27]

Input: Undirected graph (G, E), with edge weights w
Output: $\frac{1}{2}$-Approximation matching M

1: $M := \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: $\quad (a, b) :=$ **some locally heaviest edge from** E
4: $\quad M := M \cup \{(a, b)\}$
5: $\quad E := E \setminus \big\{(x, y) : x \in \{a, b\}\big\}$

Theorem 3.4 (Preis [27])**.** *The LocalMax-Algorithm 3.2 returns a matching M_{LM} that has at least $\frac{1}{2}$ the summed edge weight of a maximum weighted matching M^**

3.2.3 Parallel approximate matching

The advantage of Preis's LocalMax-algorithm is that it is much more local than the Greedy-algorithm. Manne and Bisseling [20] built on the work of Preis [27] and Hoepman [18] and developed a parallel distributed-memory algorithm that determines the locally heaviest edges from the perspective of vertices. These vertices need to know only information from their neighboring vertices to figure out which are the locally heaviest edges; this limits the inter-processor communication to neighboring vertices on different processors that are sharing information.

The distributed LocalMax-algorithm follows the Bulk Synchronous Parallel (BSP) model described in detail by Bisseling [8]. The BSP model assumes a

distributed memory and computation system connected by a communication network. Computation supersteps are alternated with communication supersteps, separated by a global barrier synchronization. In a computation superstep, each processor can perform local computations with locally available data and modify them. After every processor has finished its own work, or after a certain amount of time, there is a global synchronization and messages are passed between all processors. The global communication superstep is particularly helpful for irregular communication patterns that come with most graph algorithms, as communication needs to be called only one-sided.

Algorithm 3.3 outlines the distributed LocalMax-Algorithm, which is due to Manne and Bisseling [20], but is described more precisely by Bisseling [7], whom we follow here. We start with a distribution of the vertices over the processors: each processor s of the p processors has a set V_s of local vertices such that:

$$\bigcup_{s=1}^{p} V_s = V,$$

$$V_s \cap V_t = \emptyset, \quad \forall s \neq t.$$

Also, the edges and edge weights are stored with the vertices on the processors. The algorithm proceeds to define the set $Adj(v)$, which are the vertices adjacent to vertex v that are still available for matching. As input, the algorithm also requires the set of halo vertices of each processor H_s defined by

$$H_s = \bigcup_{v \in V_s} Adj(v) \setminus V_s,$$

which are those vertices that reside on other processors, but have a direct edge to a local vertex. The set E_s, on the other hand, is the set of all the edges which are directly connected to the vertices V_s on processor s:

$$E_s = \{(u, v) \in E : u \in V_s\}.$$

The last input is the vertex distribution ϕ, where $\phi(v) = s$, for $v \in V$ means that vertex v resides on processor s, i.e., $v \in V_s$.

The distributed matching algorithm initializes by setting the preferred match $pref(v)$ for each of the local vertices $v \in V_s$; this is simply the edge with the highest weight. All the edges for which both vertices prefer each other are locally heaviest edges and are added directly to the local matching M_s. However, this can only be known if both vertices are local, so if $pref(v)$ resides on a different processor, a proposal is put for matching that edge in the remote processor. This message is then buffered until the next global synchronization, when it is bundled with the other messages to the same processor and sent and received in bulk.

Each vertex that gets matched, is added to the set D_s. We need to update the unmatched vertices that had preferred to match with an already matched vertex in D_s. In the computation superstep, this is done by updating the set $Adj(x)$ of neighbors $x \in Adj(v)$ of the vertices $v \in D_s$; we remove v from $Adj(x)$, and update $pref(x)$ to the vertex on the remaining adjacent edge with the highest weight. Again, we check if new pairs have preferred each other and can be matched. However, if a matched vertex $v \in D_s$ has neighbors on one or more

Algorithm 3.3 Distributed LocalMax-Algorithm, by Manne and Bisseling [7]

Input: V_s, E_s, H_s, vertex distribution ϕ
Output: Distributed matching M_s

1: **for all** $v \in V_s$ **do**
2: $\quad pref(v) = null$
3: $D_s := \emptyset$
4: $M_s := \emptyset$
5: **for all** $v \in V_s$ **do**
6: $\quad Adj(v) := \{w \in V_s \cup H_s | (v, w) \in E_s\}$
7: $\quad pref(v) := \text{argmax}\{w(u, v) : u \in Adj(v)\}$
8: $\quad$ **if** $pref(v) \in V_s$ **then**
9: $\quad\quad$ **if** $pref(pref(v)) = v$ **then**
10: $\quad\quad\quad D_s := D_s \cup \{v, pref(v)\}$
11: $\quad\quad\quad M_s := M_s \cup \{(v, pref(v))\}$
12: $\quad$ **else**
13: $\quad\quad$ put $proposal(v, pref(v))$ in $P(\phi(pref(v)))$
14: **repeat**
15: $\quad$ Computation superstep, Algorithm 3.4
16: $\quad$ **sync**
17: $\quad$ Communication superstep, Algorithm 3.5
18: **until** there are no more messages

other processors, a message is sent to those processors that v is unavailable for matching.

During synchronization, each processor receives a number of messages, which are handled in the communication superstep. If the message is a proposal from a remote vertex to match with a local vertex, this is done if the preference is mutual, after which the remote processor is notified that the vertices have been matched. When receiving such an acceptation, the other processor also adds this edge to its local matching, and adds the local vertex to its set of matched vertices in order to remove adjacent edges. Lastly, if the message is that a remote vertex is unavailable for matching, we need to remove edges from local vertices to this remote vertex. This updates the $Adj(v)$ and $pref(v)$ of local neighbors $v \in V_s$ of the remote vertex.

Algorithm 3.3 terminates after at most $\frac{1}{2}|E|$ supersteps of computation and communication when all matched edges connect vertices on different processors and form a chain of decreasing edge weight. However, the assumption is that the vertices are distributed in such a manner that in a computation superstep, each processor can find many matches locally. The algorithm finds the same matching as Algorithm 3.2, but then distributed over the processors. That is:

$$M_{\text{LM}} = \bigcup_{s=1}^{p} M_s,$$

where M_{LM} is the LocalMax matching by Preis from Algorithm 3.2. However, this is only true if the LocalMax matching is unique, for which we need unique edge weights. Unique edge weights also prevent deadlock in the distributed algorithm. For example, consider the case when three vertices a, b

Algorithm 3.4 Distributed LocalMax-Algorithm, computation superstep

```
1: while D_s ≠ ∅ do
2:   pick v ∈ D_s
3:   D := D \ {v}
4:   for all x ∈ Adj(v) \ {pref(v)} : (x, pref(x)) ∉ M_s do
5:     if x ∈ V_s then
6:       Adj(x) := Adj(x) \ {v}
7:       pref(x) := argmax{w(x, y) : y ∈ Adj(x)}
8:       if pref(x) ∈ V_s then
9:         if pref(pref(x)) = x then
10:          D_s := D_s ∪ {x, pref(x)}
11:          M_s := M_s ∪ {(x, pref(x))}
12:      else
13:        put proposal(x, pref(x)) in P(φ(pref(x)))
14:    else
15:      put unavailable(v, x) in P(φ(x))
```

Algorithm 3.5 Distributed LocalMax-Algorithm, communication superstep

```
1: for all messages m received do
2:   if m = proposal(x, y) then
3:     if pref(y) = x then
4:       D_s := D_s ∪ {y}
5:       M_s := M_s ∪ {(x, y)}
6:       put accepted(x, y) in P(φ(x))
7:   else if m = accepted(x, y) then
8:     D_s := D_s ∪ {y}
9:     M_s := M_s ∪ {(x, y)}
10:  else if m = unavailable(x, y) then
11:    Adj(y) := Adj(y) \ {x}
12:    pref(y) := argmax{w(y, z) : z ∈ Adj(y)}
13:    if pref(y) ∈ V_s then
14:      if pref(pref(y)) = y then
15:        D_s := D_s ∪ {y, pref(y)}
16:        M_s := M_s ∪ {(y, pref(y))}
17:    else
18:      put proposal(y, pref(y)) in P(φ(pref(y)))
```

and c with three edges connecting each other and $w(a,b) = w(b,c) = w(a,c)$. Then it could happen that $pref(a) = b, pref(b) = c$ and $pref(c) = a$, and Algorithm 3.3 consequently terminates without adding any of these edges. We can ensure total ordering of the weights by using the vertex identification ID_v of vertex v, as done by Manne and Mjelde [21]. Assuming each vertex has a unique and comparable ID, then consider the effective weight triplet defined by $w(a,b) = \big(w(a,b), \max\{ID_a, ID_b\}, \min\{ID_a, ID_b\}\big)$. Sorting this effective weight triplet, first by the first element, then the second and then the third, gives a total ordering of the edge weights. Also, each node can compute the effective weight

of its adjacent edges locally.

3.3 Dynamic approximate maximum weighted matching

In many applications, graphs are subject to discrete changes over time, such as additions and deletions of edges or vertices. A dynamic graph algorithm should update the solution to a given problem efficiently when such changes are applied to the graph, as Eppstein et al. [13] define it. In the case of matching, only changes to edges are relevant; adding a vertex without edges does not affect the matching. If in a problem edges are only added and never removed, we speak of an incremental graph problem or algorithm; otherwise, if edges are only removed, it is a decremental problem. An update can be seen as a deletion and addition of the same edge with different properties, so an algorithm that handles additions and deletions can also be applied to updates and is called a fully dynamic algorithm.

Changes can be applied one by one, or as a set of changes at a time. We consider sets of updates, as they are more general and more likely accompany the large scale graphs considered in this chapter. Examples of such graphs are the internet graph or a social network graph, which undergo a large number of changes in each time step. The advantage of such bulk changes, is that they can be distributed over a parallel computer system.

In the following section, we describe approaches to dynamic weighted matching by Manne and Mjelde [21] and Anand et al. [1]. In Section 3.3.2, we present our development of the Incremental LocalMax-algorithm that handles single edge additions. We extend this algorithm in Section 3.3.3 to handle bulk additions in parallel.

3.3.1 Other efforts

Recently, Neiman and Solomon [23] and Baswana et al. [4] have developed fully dynamic algorithms for maintaining maximal matchings. These are the first efforts that go beyond simply repeating a run of a static algorithm after changes in the graph. These algorithms open the doors for further research on dynamic matching, but are essentially trivial themselves; maintaining a maximal matching is simply guaranteeing that every edge is either matched or adjacent to a matched edge.

Manne and Mjelde [21] have developed a version of the LocalMax algorithm that can be interpreted as a fully dynamic weighted matching algorithm. It was developed as a self-stabilizing algorithm, which means that the algorithm will produce the same result irrespective of the state it was initialized in. As such, self-stabilization has as goal to provide fault tolerance in parallel computing. The algorithm is given in Algorithm 3.6 and requires the edge weights to be unique, for example by considering the effective weight triplet as presented in Section 3.2.3.

For a node $v \in V$, the algorithm maintains the variable m_v which is the preferred vertex to match for v. The variable $h_v = w(v, m_v)$ is the weight of the edge between the vertex v and its preferred vertex. In an inner loop of the algorithm, the neighborhood $N(v)$ of vertex v is determined to be those vertices

$u \in V$ neighboring v, for which the edge to v has a higher weight than the weight h_u of the preferred match of the neighbor. From $N(v)$, we pick the vertex with the highest weight on the corresponding edge and set it as the preferred match. In Algorithm 3.6, we naively visit all nodes in one iteration of the while loop, but Manne and Mjelde describe more efficient ways to decide which nodes to visit.

Algorithm 3.6 Self-stabilizing LocalMax-algorithm by Manne and Mjelde [21]

Input: Graph $G = (V, E)$, arbitrary $m_v \in V$ and $h_v \in \mathbb{R}^+$ for each $v \in V$
Output: Matching M

```
1: while there are changes do
2:   for all v ∈ V do
3:     N(v) = {u ∈ V : (u, v) ∈ E ∧ w(u, v) ≥ h_u} ∪ null
4:     Bestmatch(v) = argmax_{u∈N(v)} w(u, v)
5:     if m_v ≠ Bestmatch(v) or h_v ≠ w(v, m_v) then
6:       m_v = Bestmatch(v)
7:       h_v = w(v, m_v)
8: M = {(u, v) ∈ E : m_u = m_v}
```

Figure 3.3 shows how the algorithm would execute; it is an example taken and modified from Manne and Mjelde. We start in state (a), where vertices c and b prefer each other. In the first iteration of the **while** loop and assuming the **for** loop executes in the order $a \to b \to c \to d$, we start with vertex a, which selects c as it is the neighbor with the highest weight and $w(a, c) \geq w(c, m_c)$. It continues with b, which keeps preferring c, because $w(a, b) < w(a, m_a)$. Vertex c changes its preference to a, and vertex d has no edge which is higher than its neighbors preference. Now, the algorithm is in state (b), where the edge (b, c) is removed from the matching, but (a, c) has been added. In the next iteration of the **while** loop, a does not change, but b changes its preference to d, because c is now preferring a higher weight edge. Subsequently, c does not change it status, but d now gets to prefer b. In the next iteration of the **while** loop, nothing changes, so the algorithm terminates, leaving $M = \{(a, c), (b, d)\}$.

Although the algorithm could visit every edge many times, its concept is easy to parallelize. Moreover, it has some inherent dynamism; whenever a new graph G' that is similar to G needs to be matched, the algorithm can take the previous state, all values m_v and h_v for $v \in G$, as starting point. As a result, the algorithm would converge to a result much faster than running it without initialized values of m_v and h_v. However, as Algorithm 3.6 maintains a LocalMax matching, a single change can affect the entire matching as the example in Figure 3.4 shows.

Dynamic $\frac{1}{8}$-approximation matching algorithm

To our best knowledge, Anand et al. [1] are the only authors who have presented an efficient dynamic algorithm for maintaining a weighted matching. Their approach is derived from a dynamic maximal matching algorithm and based on the observation that a maximal matching is a $\frac{1}{2}$-approximation for a maximum cardinality matching. As such, it is also a $\frac{1}{2}$-approximation for a maximum weighted matching where all weights are equal. They generalize this notion, by observing

Figure 3.3: An example of the execution of Algorithm 3.6. In (a) the initial state is displayed, where the arrows pointing out of the nodes indicate their preferred match. When two nodes prefer each other, the edge connecting them is in the current matching (red). In (b) and (c), the state after respectively one and two iterations of the **while** loop are depicted.

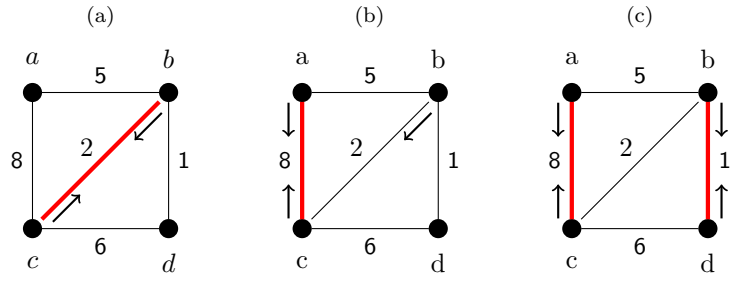
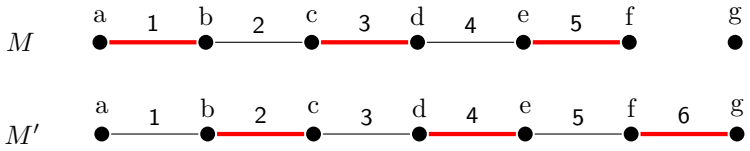


Figure 3.4: Cascading changes in a LocalMax matching. M is the LocalMax matching as obtained on this graph by Algorithms 3.1, 3.2, 3.3 and 3.6. M' is the matching on the same graph with the edge (f, g) added. The addition of (f, g) has as consequence that every edge changes its status from matched to unmatched or vice versa.



that if α is the ratio between the maximum and minimum edge weight, a maximal matching is a $\frac{1}{2\alpha}$-approximation.

Anand et al. partition the edges in a number of levels, where an edge is in the set E_i of edges at level i if its weight is in the range $[\alpha^i, \alpha^{i+1})$. First, on each level a maximal matching M_i is computed on the graph $G_i = (V, E_i)$. However, the union of these level-matchings M_i is not a matching on the graph G, as vertices that have edges on multiple levels can be matched multiple times. The authors greedily select edges from the level-matchings by starting with adding all the matched edges at the highest level to the general matching M. Then, they add those edges of the next highest level which are not adjacent to edges already in M. Let *level*(e) denote the level of edge e, then the algorithm maintains the following property:

Observation 1. *In the algorithm of Anand et al. it holds that* $\forall e \in \bigcup_i M_i$, *either* $e \in M$ *or* e *is adjacent to an edge* $e' \in M$ *such that* *level*$(e') >$ *level*(e).

The authors prove that with $\alpha = 2$ and as long as Observation 1 holds, the resulting matching M is a $\frac{1}{8}$-approximation of the maximum weighted matching. They maintain the property in Observation 1 as follows. The addition or deletion of an edge can affect the maximal matching M_i at one level, which is handled dynamically by the algorithm of Baswana et al. [4]. This results in a

number of edges being removed from or added to M_i, or both. If an edge (u, v) is removed from M_i and it was not in the general matching M, nothing happens. Otherwise, if it was in M, then the vertices u and v are now free, i.e, they do not have matched edges incident on them. These free vertices are handled with the procedure **HandleFree**(u, lev) outlined in Algorithm 3.7, where u is the free vertex and *lev* is its respective level. Also, the parameter $L^{\min}$ denotes the lowest level.

Algorithm 3.7 Recursive procedure for processing a vertex that has become free at a certain level

procedure **HandleFree**(u, lev) :

```
1: for i from lev down to L^min do
2:   if there exists a v ∈ V such that (u, v) ∈ M_i then
3:     if ∄ v' ∈ V : (v, v') ∈ M then
4:       M = M ∪ {(u, v)}
5:     else
6:       find v' ∈ V such that (v, v') ∈ M
7:       if level(v, v') < i then
8:         M = (M \ {(v, v')}) ∪ {(u, v)}
9:         HandleFree(v', level(v, v'))
```

The procedure **HandleFree**(u, lev) tries to find a match for the vertex u in the set $\cup_i M_i$. In fact, it only needs to look at levels lower than *lev*, because the fact that u was matched at *lev* means that there is no available edge at a higher level from Observation 1. Moreover, the vertex u can have at most one incident edge matched at each M_i, because else two edges at that level would be incident on the same vertex and M_i would not be a matching. At a level i, let (u, v) be this incident edge if it exists. Now, if v is not matched in M, we simply add the edge (u, v) to the matching. Otherwise, if there exists some v' such that $(v, v') \in M$, and $level(v, v') < i$, then note that as $i = level(u, v)$, Observation 1 is violated. Consequently, we remove (v, v') from the matching and add (u, v) instead. This results in v' becoming free, which is in turn dealt with by calling **HandleFree**$(v', level(v, v'))$.

The procedure starts with checking for a neighbor at level *lev* and checks for subsequent lower levels until it returns. Consequently, the available edge from the highest possible level is added which gives the same result as the static greedy procedure. The same procedure can be applied when an edge (u, v) is added to the level-matching M_i. If u or v were already matched on a higher level, then we do not add (u, v) to the matching as the property in Observation 1 is satisfied. Otherwise, either vertex can be free or is matched with an edge on a lower level. Let (u, u') and (v, v') be these adjacent edges, and if one or both exist, we remove the edges (u, u') and (v, v') from M and call **HandleFree**$(u', level(u, u'))$ and **HandleFree**$(v', level(v, v'))$.

The authors prove that an edge addition or deletion to E leads to $O(\log |V|)$ (recursive) calls of **HandleFree**. Also, the maintaining of the maximal matching M_i costs $O(\log |V|)$. Let

$$C = \frac{\max_{e \in E} w(e)}{\min_{e \in E} w(e)},$$

be the ratio between the maximum and minimum edge weight on G, then $\log_\alpha C$ is the number of levels. As at most all the levels are visited in each call, an edge addition or deletion can be handled in $O(\log |V| \log_\alpha C)$.

Although the algorithm is good first attempt at dynamic matching, its performance bound of $\frac{1}{8}$ times the optimal matching is quite low. One reason is that using a maximal matching as an approximation for a maximum weighted matching is rather naive. Moreover, an edge addition at the highest level could lead to a cascade of changes throughout every level, which would frustrate parallelization of the algorithm because local changes can have a global effect.

3.3.2 Incremental LocalMax-algorithm

Our approach is motivated by keeping changes as local as possible and is inspired by the LocalMax algorithm. However, as explained in Figure 3.4, maintaining a LocalMax matching could lead to a cascade of changes as the result of a single edge addition. Here, we consider an incremental graph, so we only deal with edge additions.

Define $match(a) = null$ if a is a free vertex in the matching M and $match(a) = b$ if $(a, b) \in M$, and let $w(x, null) = 0$ for all x. We start off with a LocalMax matching M on a graph $G = (V, E)$ and handle the addition of an edge (a, b) to E with Algorithm 3.8. When a and b are free, we can simply add (a, b) to the matching. Otherwise, we only add (a, b) if its weight exceeds the weight of the edge or pair of edges that have to be removed from M. In the case that one or two edges are removed, we check for each if it is adjacent to edges that are now free and add the one with highest weight to the matching. Consequently, the addition of an edge to E leads to at most three edge additions and two edge removals from M. We claim that Algorithm 3.8 maintains a $\frac{1}{2}$-approximation over an arbitrary set of subsequent edge additions. In the rest of this section, we develop the theory to prove this approximation ratio.

Algorithm 3.8 Incremental LocalMax-Algorithm

Input: (Incremental) LocalMax matching M, weighted graph $G = (V, E)$, added weighted edge (a, b)
Output: $\frac{1}{2}$-Approximation matching M

```
1:  if w(a,b) ≥ w(a, match(a)) + w(b, match(b)) then
2:    if a is matched and match(a) has free neighbors then
3:      Let e ∈ E \ M be the maximum-weight free edge incident on match(a)
4:      M := M ∪ {e}
5:    M := M \ {(a, match(a))}
6:    if b is matched and match(b) has free neighbors then
7:      Let e ∈ E \ M be the maximum-weight free edge incident on match(b)
8:      M := M ∪ {e}
9:    M := M \ {(b, match(b))}
10:   M := M ∪ {(a, b)}
```

Pettie and Sanders [24] base their analysis of the approximation quality on the minimum length of an augmenting path or cycle of the approximation matching.

We, on the other hand, try to bound the weight of an augmenting path or cycle. If the weight in M of any augmenting path or cycle p is at least half the weight of p, then M is in fact a $\frac{1}{2}$-approximation, see Theorem 3.5.

Theorem 3.5 ($\frac{1}{2}$-approximation)**.** *Let M be a matching and M^* a maximum weighted matching on a graph G and assume that for every alternating path and cycle p of M, it holds that $w(p \cap M) \geq \frac{1}{2} w(p \setminus M)$. Then $w(M) \geq \frac{1}{2} w(M^*)$.*

Proof. Following the proof of Theorem 3.3, consider the symmetric difference between M and M^*: $M \triangle M^*$. It consists of a set of k vertex-disjoint and edge-disjoint alternating paths and cycles p_i, both with respect to M and to M^*, such that $p_i \setminus M = p_i \cap M^*$. Following Pettie and Sanders [24], we can decompose the weight of the two matchings in that of their common edges, and their edges on the alternating paths and cycles:

$$w(M) = w(M \cap M^*) + \sum_{i=1}^{k} w(p_i \cap M),$$

$$w(M^*) = w(M \cap M^*) + \sum_{i=1}^{k} w(p_i \cap M^*).$$

Subsequently, our contribution is that using the assumption on the bound of the weight of any alternating path and cycles, we get that:

$$\begin{aligned} w(M) &= w(M \cap M^*) + \sum_{i=1}^{k} w(p_i \cap M), \\ &\geq w(M \cap M^*) + \sum_{i=1}^{k} \frac{1}{2} w(p_i \cap M^*), \\ &\geq \frac{1}{2}\Big(w(M \cap M^*) + \sum_{i=1}^{k} w(p_i \cap M^*)\Big) = \frac{1}{2} w(M^*), \end{aligned}$$

proving, in fact, that M is a $\frac{1}{2}$-approximation of M^*. □

Theorem 3.5 provides a helpful framework for proving the approximation quality of a matching. As an example, in Theorem 3.6 we provide a proof for the Greedy matching in Algorithm 3.1, by using Theorem 3.5.

Theorem 3.6 (Greedy)**.** *Let M be the Greedy matching created by Algorithm 3.1 and M^* a maximum weighted matching. Then $w(M) \geq \frac{1}{2} w(M^*)$.*

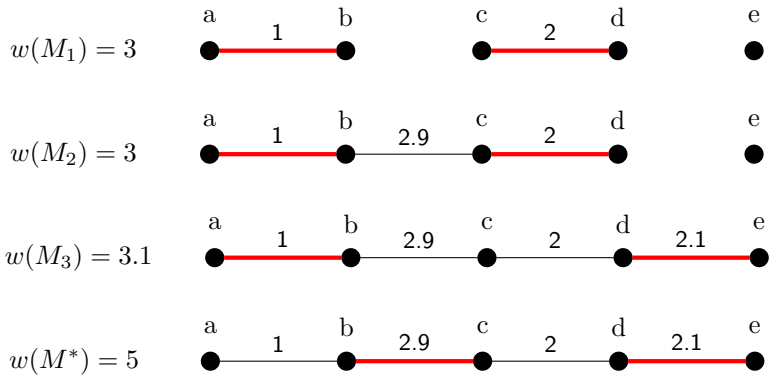
Proof. Let p be an alternating path or cycle with respect to M. Now, take an unmatched edge $a \in p \setminus M$; it must have either one or two matched neighbors in p. For at least one such neighbor $b \in p \cap M$ of a, we have that $w(b) > w(a)$, because else edge a would have been picked and matched by the Greedy algorithm before b. So every unmatched edge in p is dominated by a neighboring matched edge in p. However, it can happen that two unmatched edges are dominated by the same matched edge. That is how we get to the factor $\frac{1}{2}$:

$$w(p \cap M) \geq \frac{1}{2} w(p \setminus M).$$

Subsequently, it follows from Theorem 3.5 that M is indeed a $\frac{1}{2}$-approximation. □

Let M be the incremental LocalMax matching maintained by Algorithm 3.8. It is not true that $w(p \cap M) \geq \frac{1}{2} w(p \setminus M)$ for every alternating path or cycle p with respect to M, see the counterexample in 3.5. Note that in this example, still, the overall matching is a half-approximation. Therefore, we need to divide the graph G into paths and cycles which are not necessarily alternating, but for which we still have the property that M is a $\frac{1}{2}$-approximation on these paths and cycles and that these paths and cycles cover G in total.

Figure 3.5: Counterexample to the alternating path assumption. The incremental LocalMax matching M_3 is built in two steps from M_1 and M_2 by adding the edges (b, c) and (d, e). Comparing our matching with the maximum weighted matching M^*, we see that $p = M_3 \triangle M^* = \{(a, b), (b, c)\}$ with $w(M_3 \cap p) = 1$ and $w(M^* \cap p) = 2.9$, such that $w(M_3 \cap p) < \frac{1}{2} w(M^* \cap p)$.



For this reason, we prove in Theorem 3.7 that the incremental LocalMax matching is a $\frac{1}{2}$-approximation on linear graphs, on which it is trivial that the disconnected paths and cycles cover the entire graph.

Theorem 3.7 (Incremental LocalMax on linear graphs)**.** *Let $G = (V, E)$ be a linear graph, which means that each vertex has at most two incident edges. Let edges be added to E such that G remains linear. Also, let M be the incremental LocalMax matching maintained by Algorithm 3.8, which starts of with M being a LocalMax matching on G and updates M under the addition of edges. Lastly, let M^* be a maximum weighted matching on G. Then $w(M) \geq \frac{1}{2} w(M^*)$.*

Proof. Note that the graph G is composed of disconnected simple paths and cycles. We start with a graph $G_0 = (V, E_0)$ with a LocalMax matching M_0 and add edges until we arrive at $G = (V, E)$. Let $G_i = (V, E_i)$ be the graph after i edges have been added and let M_i be the matching obtained by applying Algorithm 3.8 to each edge addition. Assuming that for M_{i-1} it holds that $w(M_{i-1}) \geq \frac{1}{2} w(M^* \cap E_{i-1})$, we will show that after the addition of $a = E_i \setminus E_{i-1}$ to the graph, it holds that $w(M_i) \geq \frac{1}{2} w(M^* \cap E_i)$. As we start with a matching M_0 satisfying $w(M_0) \geq \frac{1}{2} w(M^* \cap E_0)$, this proves the theorem.

At step i, take the addition of edge a such that $E_i = E_{i-1} \cup \{a\}$. We distinguish four possibilities on whether a is in M_i or M^*, or both, and show for each that the inequality holds.

i) Not matched, $a \notin M_i$ and $a \notin M^*$. In this case, we have that $w(M_i) = w(M_{i-1}) \geq \frac{1}{2} w(M^* \cap E_{i-1}) = \frac{1}{2} w(M^* \cap E_i)$.

ii) Not matched, $a \notin M_i$, but $a \in M^*$. The edge a must have one matched neighbor $b \in M_{i-1}$ with $w(b) > w(a)$, or two matched neighbors $b, c \in M_{i-1}$ with $w(b) + w(c) > w(a)$. The edges b and c are not in M^*, but each may be adjacent to another edge matched in M^*. Similar to the proof of Theorem 3.6 of the Greedy algorithm, this results in a tight inequality on M_i, i.e., $w(M_i) \geq \frac{1}{2} w(M^* \cap E_i)$.

iii) Matched, $a \in M_i$, but $a \notin M^*$. This is a trivial case: $w(M_i) \geq w(M_{i-1}) \geq \frac{1}{2} w(M^* \cap E_{i-1}) = \frac{1}{2} w(M^* \cap E_i)$.

iv) Matched, $a \in M_i$ and $a \in M^*$. If $M_i = M_{i-1} + \{a\}$, no edges have been removed by Algorithm 3.8, so $w(M_i) = w(M_{i-1}) + w(a) \geq \frac{1}{2}(w(M^* \cap E_{i-1}) + w(a)) = \frac{1}{2} w(M^* \cap E_i)$. Otherwise, one or two edges have been removed from M_i by Algorithm 3.8. This is the difficult case, which we treat in detail here.

If a has two previously matched neighbors $b_1, b_2 \in M_{i-1}$, then we can split a in a_1 and a_2 such that $w(a_1) \geq w(b_1)$ and $w(a_2) \geq w(b_2)$. Otherwise, if a has one previously matched neighbor $b_1 \in M_{i-1}$, we take $a_1 = a$ such that $w(a_1) \geq w(b_1)$. We define three cases on whether and how b_1 is connected to other edges, see Figure 3.6.

In case (a), b_1 has no other neighbors. In E_{i-1}, the edge b_1 had no neighbors, so it can be removed from M_{i-1} such that M_{i-1} remains a $\frac{1}{2}$-approximation with respect to $M^* \cap E_{i-1}$.

In case (b), b_1 has neighbor c which is not adjacent to another matched edge. Again, removing b_1 from M_{i-1} while adding c will maintain that $w(M_{i-1}) \geq \frac{1}{2} w(M^* \cap E_{i-1})$. This is logical if we consider a scenario where b_1 has never been added to the graph. Note that also, $c \in M_i$.

In case (c), b_1 has a neighbor c that in turn has a matched neighbor d. The important observation is that since c is not matched in M_{i-1}, its weight satisfies $w(c) < w(b_1) + w(d)$. If also $w(c) \leq w(d)$, then c is already dominated by d, and we can remove b_1 from M_{i-1} while maintaining $w(M_{i-1}) \geq \frac{1}{2} w(M^* \cap E_{i-1})$, again by considering a scenario where b_1 does not exist. Otherwise, if $w(d) < w(c) < w(b_1) + w(d)$, then $c \in M^*$ by Theorem 3.3 on the maximum weighted matching. Also, $a_1 \in M^*$, which on these four edges gives:

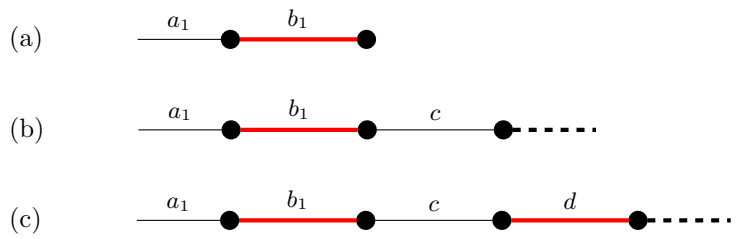
$$w(M_i \cap \{a_1, b_1, c, d\}) = w(a_1) + w(d),$$

$$w(M^* \cap \{a_1, b_1, c, d\}) < w(a_1) + w(b_1) + w(d) \leq 2w(a_1) + w(d).$$

The edge d may be adjacent to another edge matched in M^*, resulting in a tight inequality on M_i.

One of the three cases in Figure 3.6 is also applicable to a_2 if there exists an edge b_2. It is not excluded that a is on a cycle, where it is possible that c is adjacent to a in case (b), or that $d = b_2$ in case (c).

Figure 3.6: Three cases as illustration to point iv) of the proof of Theorem 3.7. The edge a_1 is the part of the added edge a such that $w(a_1) \geq w(b_1)$. Red edges are in the matching M_{i-1}. A dashed line signifies that there may or may not be another adjacent, unmatched edge.



□

Conjecture 3.8 is the formulation of the $\frac{1}{2}$-approximation quality of the incremental LocalMax algorithm. An approach to the proof could be made in a similar fashion as Theorem 3.5. Consider a subset $S \subseteq E$ of edges, which includes the symmetric difference:

$$M \triangle M^* \subseteq S,$$

and possibly some more edges. In the proof of Theorem 3.5, we had that $S = M \triangle M^*$ and S consisted exclusively of vertex-disjoint alternating paths and cycles. With S containing the symmetric difference, we get that:

$$w(M) = w\big((M \cap M^*) \setminus S\big) + w(S \cap M),$$

$$w(M^*) = w\big((M \cap M^*) \setminus S\big) + w(S \cap M^*),$$

meaning that we prove the theory if we can find such a set S satisfying $w(S \cap M) \geq \frac{1}{2} w(S \cap M^*)$, which in Theorem 3.5 we proved by showing that S consisted of alternating paths and cycles which satisfied this bound. As the counter example in Figure 3.5 and the proof of Theorem 3.7 show, we need to extend S in such a way that it also contains the paths and cycles on which the alternating part of the path or cycle is not bounded and show that indeed $w(S \cap M) \geq \frac{1}{2} w(S \cap M^*)$. We have not succeeded in finding such an S, but we also did not find an example that disproves Conjecture 3.8.

Conjecture 3.8 (Incremental LocalMax)**.** *Let M be the incremental LocalMax matching maintained by Algorithm 3.8 on the general graph $G = (V, E)$ and let M^* be a maximum weighted matching. Then $w(M) \geq \frac{1}{2} w(M^*)$.*

Analyzing the running time of the incremental LocalMax algorithm, we note that checking whether the added edge (a, b) satisfies $w(a, b) \geq w(a, \mathit{match}(a)) + w(b, \mathit{match}(b))$ can be done in $O(1)$. So if $w(a, b)$ is smaller, then the algorithm terminates in $O(1)$ as well. Also, if neither a or b were matched, the algorithm takes $O(1)$. Otherwise, checking whether $\mathit{match}(a)$ has matched neighbors and finding the respective maximum-weight edge, requires us to visit $\mathit{deg}(v)$ vertices and edges, where $\mathit{deg}(v)$ is the degree of $v = \mathit{match}(a)$, i.e., the number of edges incident on v. In general graphs $\mathit{deg}(v)$ is bounded by the number of vertices,

but in many practical applications, the maximum degree in a graph is relatively small. This gives a running time of $O(\max_{v \in V} deg(v))$ for processing the addition of a single edge with Algorithm 3.8. In the next section, we will show that the incremental LocalMax algorithm is well suited for parallelization.

3.3.3 Parallel incremental LocalMax

Building on the work of Manne and Bisseling [20], we developed a generalization of the distributed LocalMax algorithm described in Section 3.2.3. Given a distributed matching M_s on processor s on a graph with vertices V_s and edges E_s, Algorithms 3.9, 3.10, 3.11, 3.12 and 3.13 compute a new matching on the graph when a set of edges A_s is added to E_s. It is a generalization of Algorithm 3.3 in the sense that if the initial matching is empty, $M_s = \emptyset$, and we give all edges of the graph at once as input, $A_s = E_s$, then Algorithm 3.9 runs in the same way, resulting in the same matching. In this section, we will provide a detailed description of the algorithm.

Algorithm 3.9 runs on p processors, each of which maintains its own piece of the data. Similarly as in Algorithm 3.3, a processor s has a unique vertex set V_s as input:

$$\bigcup_{s=1}^{p} V_s = V,$$

$$V_s \cap V_t = \emptyset, \quad \forall s \neq t.$$

Furthermore, E_s is the set of edges on s, which contain all edges incident on at least one vertex in V_s. New are the inputs M_s, a subset of E_s of edges already in the matching, and the set of added edges A_s. The definition of the set of halo vertices H_s on processor s is updated to also include vertices connected by edges in A_s:

$$H_s = \bigcup_{t \neq s} \{v \in V_t \ : \ \exists \ (u, v) \in E_s \cup A_s\}.$$

They are the vertices residing on other processors, that have a, possibly new, edge to a vertex in V_s.

Similarly to the static parallel matching algorithm, Algorithm 3.9 is composed of an initialization step, followed by alternating supersteps of computation and communication. In the initialization, the edge set is updated, followed by setting the preferred new match *pref*(v) of each vertex v to the default *null*. Subsequently, for the new edges that connect to a vertex on a different processor, we communicate the weight of the local adjacent matched edge (or 0 if it does not exist) to the remote processor, for which we will see the purpose. In the BSP model, messages are buffered until a global barrier synchronization, which allows all processors to communicate simultaneously. Because each processor needs the weights of the matches of its halo vertices directly, we follow with such a **sync** here.

The algorithm maintains the set *Adj*(v) for each vertex v, which contains the new neighbors of v. In the procedure **update_pref**(x) presented in Algorithm 3.12, we pick a vertex y from this set *Adj*(x), such that the new edge $(x, y) \in A_s$ would give the highest gain in the weight of the matching. This means that it should not only meet the incremental LocalMax criterion in Algorithm 3.8 that

the new edge should have a higher weight than the potential matched adjacent edges, but it should also have the highest corresponding difference in weights from all newly added edges adjacent to x. The gain for the addition of edge (x, y) is $w(x, y) \geq w(x, \mathit{match}(x)) + w(y, \mathit{match}(y))$. As the vertex y could reside on another processor, it is here that we require the weight $\mathit{match}(y)$ of the potential match of y.

If the preferred new match of x resides on the same processor, and if the preference is mutual, the procedure **update_pref**(x) can create a new match. In order to do so, it will have to break up the matches that may be incident on the new edge. This is handled by the procedure **break_up**(u, v) given in Algorithm 3.13, which takes care of the freed neighbors of the broken up edge. The two vertices are subsequently added to the set D_s of recently matched vertices and the matching is updated. Otherwise, if the preferred new match resides on another processor, we put a proposal for matching in the remote processor.

The procedure **break_up**(x, y) removes the edge (x, y) from the matching and subsequently checks for free neighbors of y. Note that vertex x has just been matched to some other vertex in favor of being matched to y. These free neighbors are added to $\mathit{Adj}(y)$, and for each of them we add y to their own set of potential new matches. If a free neighbor z of y resides on the same processor, this is done easily. Additionally, we update the preference of z directly to include y, but we do not call **update_pref**(z), because y has not yet updated its own preference so we can save on some function calls this way. If z resides on another processor, we put the message $\mathit{added}(y, z)$ in the remote processor, which will do the same once it is processed. Finally, we update the preference of y, which will eventually decide if and how y should be matched. If y itself resides on another processor, the message $\mathit{freed}(x, y)$ will cause **break_up**(x, y) to be executed on the remote processor once it is received.

After initialization, the algorithm continues with a computation superstep depicted in Algorithm 3.10. Here, it pops elements from the set D_s, which are the vertices that have just been matched. For each vertex $v \in D_s$, it visits the neighboring vertices that have a newly added edge to v in order to process the fact that v is no longer available for matching. For such a vertex x, we remove v from its adjacency list of newly added edges and updates its preference, unless x belongs to another processor, in which case the message $\mathit{unavailable}(v, x)$ will cause it to be done on the remote processor.

After a global synchronization, the communication superstep handles all the incoming messages as were explained before. Lastly, the $\mathit{proposal}(x, y)$ message means that the remote vertex x wants to match with y. If this preference is mutual, we break up the possible local match to y, and add (x, y) to the matching. A new message $\mathit{accepted}(x, y)$ is sent to do the same on the remote processor for x.

Correctness of the algorithm

It is difficult to prove that a parallel algorithm produces correct results, especially when much different communication is involved. One method is to reduce it in some way to a sequential algorithm and show that the sequential algorithm is correct. In our case, we could try to reduce the parallel incremental LocalMax algorithm to the sequential incremental LocalMax algorithm. However, the cor-

Algorithm 3.9 Parallel Incremental LocalMax-Algorithm

Input: Distributed matching M_s, new edges A_s, vertices V_s, halo vertices H_s, old edges E_s, vertex distribution ϕ
Output: Updated distributed matching M_s

1: $E_s = E_s \cup A_s$
2: **for all** $v \in V_s$ **do**
3: $\quad pref(v) = null$
4: **for all** $(u, v) \in A_s : u \in V_s, v \in H_s$ **do**
5: $\quad$ **put** $w(u, match(u))$ **in** $P(\phi(v))$
6: **sync**
7: $D_s = \emptyset$
8: **for all** $v \in V_s$ **do**
9: $\quad Adj(v) = \{u \in V_s \cup H_s : (u, v) \in A_s\}$
10: $\quad$ **update_pref**(v)
11: **while** there are messages **do**
12: $\quad$ Computation superstep, Algorithm 3.10
13: $\quad$ **sync**
14: $\quad$ Communication superstep, Algorithm 3.11

Algorithm 3.10 Computation superstep

1: **while** $D_s \neq \emptyset$ **do**
2: $\quad$ pick $v \in D_s$
3: $\quad D = D \setminus \{v\}$
4: $\quad$ **for all** $x \in Adj(v) \setminus \{pref(v)\} \ : \ (x, pref(x)) \notin M_s$ **do**
5: $\quad\quad$ **if** $x \in V_s$ **then**
6: $\quad\quad\quad Adj(x) = Adj(x) \setminus \{v\}$
7: $\quad\quad\quad$ **update_pref**(v)
8: $\quad\quad$ **else**
9: $\quad\quad\quad$ **put** $unavailable(v, x)$ **in** $P(\phi(x))$

rectness of the incremental LocalMax is already under the scrutiny of Conjecture 3.8, so such a proof is not useful at this point.

Algorithm 3.11 Communication superstep

```
1: for all messages m received do
2:   if m = proposal(x, y) then
3:     if pref(y) = x then
4:       if match(y) ≠ null then
5:         break_up(y, match(y))
6:       D_s = D_s ∪ {y}
7:       M_s = M_s ∪ {(x, y)}
8:       put accepted(x, y) in P(φ(x))
9:   else if m = accepted(x, y) then
10:    if match(x) ≠ null then
11:      break_up(x, match(x))
12:    D_s = D_s ∪ {x}
13:    M_s = M_s ∪ {(x, y)}
14:  else if m = unavailable(x, y) then
15:    Adj(y) = Adj(y) \ {x}
16:    update_pref(y)
17:  else if m = added(x, y) then
18:    match(x) = null
19:    Adj(y) = Adj(y) ∪ {x}
20:    update_pref(y)
21:  else if m = freed(x, y) then
22:    break_up(x, y)
```

Algorithm 3.12 update_pref(x)

```
1: pref(x) = argmax{w(x, y) − w(x, match(x)) − w(y, match(y)) : y ∈ Adj(x)}
2: if w(x, pref(x)) − w(x, match(x)) − w(pref(x), match(pref(x)) < 0 then
3:   pref(x) = null
4: if pref(x) ∈ V_s then
5:   if pref(pref(x)) = x then
6:     if match(x) ≠ null then
7:       break_up(x, match(x))
8:     if match(pref(x)) ≠ null then
9:       break_up(pref(x), match(pref(x)))
10:    D_s = D_s ∪ {x, pref(x)}
11:    M_s = M_s ∪ {(x, pref(x))}
12: else
13:   put proposal(x, pref(x)) in P(φ(pref(x)))
```

Algorithm 3.13 break_up(x, y)

1: $M_s = M_s \setminus \{(x, y)\}$
2: **if** $y \in V_s$ **then**
3: $\quad N = \left\{z \in V_s \cup H_s : (y, z) \in E_s \wedge \mathit{match}(z) = \mathit{null}\right\}$
4: $\quad \mathit{Adj}(y) = \mathit{Adj}(y) \cup N$
5: $\quad$ **for all** $z \in N$ **do**
6: $\quad\quad$ **if** $z \in V_s$ **then**
7: $\quad\quad\quad \mathit{Adj}(z) = \mathit{Adj}(z) \cup \{y\}$
8: $\quad\quad\quad \mathit{pref}(z) = \text{argmax}\{w(u, z) - w(u, \mathit{match}(u)) : u \in \mathit{Adj}(z)\}$
9: $\quad\quad$ **else**
10: $\quad\quad\quad$ **put** $\mathit{added}(y, z)$ **in** $P(\phi(z))$
11: $\quad$ **update_pref**(y)
12: **else**
13: $\quad$ **put** $\mathit{freed}(x, y)$ **in** $P(\phi(y))$

Chapter 4

Conclusion

Graphs are a fundamental component of many applications in scientific computing. They can be very large and undergo changes frequently, so it is important to develop fast algorithms that deal with change effectively. In this thesis, we considered two dynamic problems in particular: scheduling on a precedence graph and approximate matching.

4.1 Scheduling algorithms

The problem defined by Progressive Planning is NP-hard if we want to minimize the makespan of the schedule. It is, however, unclear which objective function we wish to minimize. For these reasons, we have developed a heuristic algorithm to schedule the jobs that is based on list scheduling. This changes the question of which function to minimize to which jobs to prioritize over which other jobs. This priority is composed on the basis of three simple criteria: jobs with an earlier deadline, jobs which are succeeded by a large amount of work in other jobs, and jobs which are succeeded by work shared by many different people, are all more important. The last two criteria require that we propagate the processing times and people on the jobs through the precedence graph to each job's predecessors, which we do in linear running time. We do this for the deadline as well; if a job's successor has a deadline, then the job itself has a deadline equal to its successor's deadline minus its successor's processing time. Consequently, we have all the information gathered in a single priority measure for each job. We schedule by picking the available job with the highest priority and process it whenever a role is idle.

The quality of the resulting schedule is difficult to analyze, because we have not defined a clear measure for the optimality of the schedule. Moreover, if we had one, our problem size is too large to compute an optimal solution that we can compare our schedule with. Lastly, our problem is composed of estimates rather than deterministic data, which means that the optimal schedule could only be known in retrospect when no more changes can occur. On the other hand, the resulting schedule is based on simple heuristics that are easy to understand and make sense of. In linear time, we have shown how to compute the critical path to each job, which forms a lower bound on the completion time of a job.

The scheduling algorithm is great in terms of computation speed and is shown to scale well with larger problem sizes. We speed up the scheduling further by limiting the problem size to the connected components in the undirected graph composed of the precedence relations between jobs and their dependencies on their roles. The schedules for the connected components are completely independent, so scheduling only one is much easier. This speed is an advantage for the dynamic problem that we face. The data changes, and as it does, we can quickly compute a new schedule for the connected component where the change happened. Without giving in on a schedule's feasibility, we cannot do better than rescheduling the entire connected component in the worst case.

On the other hand, we have implemented a dynamic algorithm for answering queries on whether the graph is acyclic when a certain edge is added. Also, the connected components are maintained dynamically in a lazy fashion. We do not check whether a connected component has to be split after a certain change, because this is unlikely to happen and we can save much effort if we do not check this.

In conclusion, our scheduling algorithm is extremely fast and has the benefit of easily incorporating all the constraints given by Progressive Planning. In the light of scheduling theory, it is remarkable that there has not been any research to this problem with jobs pre-assigned to their processors and general precedence constraints. This case definitely deserves more attention, because of its applicability in project management. It would be interesting to see which measures can be (approximately) minimized in which ways. We could also compare our scheduler on these measures. On the other hand, we do not consider it advantageous to try to model the uncertainty in our problem mathematically, because we update the schedule as soon as a change occurs and the environment for use of Progressive Planning is uncertain in any case.

4.2 Matching algorithms

We have developed a simple and fast algorithm for maintaining a $\frac{1}{2}$-approximation of a maximum weighted matching on general weighted graphs. It is based on the Greedy and LocalMax algorithms of Avis [3] and Preis [27] for computing a $\frac{1}{2}$-approximation matching statically. Also, our algorithm was motivated to keep the exchange of information as local possible. On incremental graphs, Algorithm 3.8 adds a new edge to the matching only if removing adjacent matched edges does not cost more in weight than the weight of the new edge. If it is added, we check if the edges adjacent to the two edges that might have been removed, have become free, and we add the ones with maximum weight. Although this requires us to visit up to $2 \max_{v \in V} deg(v)$ edges, it is likely much faster than maintaining a LocalMax matching, where the addition of a single edge can change the entire matching.

We have developed a theoretical framework based around Theorem 3.5, in which we showed that if for a matching we can bound the weight of the matched edges in an alternating path or cycle by the half of the weight of the edges not matched, then the matching is a $\frac{1}{2}$-approximation. This theorem provides a practical approach to proving the approximation quality of matchings; we provided an alternative proof of the $\frac{1}{2}$-approximation quality of a Greedy matching.

Additionally, we proved that our incremental LocalMax algorithm maintains a $\frac{1}{2}$-approximation on linear graphs. We easily extended our algorithm to also handle edge removals from a graph.

Although our theoretical analysis is thorough, we could not prove that the incremental LocalMax algorithm maintains a $\frac{1}{2}$-approximation on general incremental graphs. We do, however, expect that it maintains a $\frac{1}{2}$-approximation. Proving this approximation ratio should definitely be the topic of further research. This would also pave the way for an efficient fully dynamic matching algorithm. As interesting as the theoretical performance of incremental LocalMax, we would like to see how useful the resulting matchings are in practical applications.

Our approach to incremental matching only uses information of the direct neighbors of the updated edge and of their neighbors. This keeps the accessing of information local, which is useful for parallel computation of the matching. We generalized the parallel LocalMax algorithm by Manne and Bisseling [20] to be able to handle a bunch of edge insertions at the same time dynamically. To our knowledge, Algorithm 3.9 is the first parallel algorithm to maintain a matching dynamically. Therefore, it is interesting to see where it can be employed in practice and how it performs in these applications.

Bibliography

[1] Abhash Anand et al. "Maintaining approximate maximum weighted matching in fully dynamic graphs". In: *arXiv preprint arXiv:1207.3976* (2012).

[2] Bas O Fagginger Auer and Rob H Bisseling. "A GPU algorithm for greedy graph matching". In: *Facing the Multicore-Challenge II*. Springer, 2012, pp. 108–119.

[3] David Avis. "A survey of heuristics for the weighted matching problem". In: *Networks* 13.4 (1983), pp. 475–493.

[4] Surender Baswana, Manoj Gupta, and Sandeep Sen. "Fully dynamic maximal matching in O (log n) update time". In: *Proceedings-Annual Symposium on Foundations of Computer Science*. IEEE. 2011, pp. 383–392.

[5] Claude Berge. "Two theorems in graph theory". In: *Proceedings of the National Academy of Sciences of the United States of America* 43.9 (1957), p. 842.

[6] Christian Bierwirth and Dirk C Mattfeld. "Production scheduling and rescheduling with genetic algorithms". In: *Evolutionary Computation* 7.1 (1999), pp. 1–17.

[7] Rob H Bisseling. *Parallel algorithms, lecture slides*. Utrecht University, `http://www.staff.science.uu.nl/~bisse101/Education/PA/matching.pdf`.

[8] Rob H Bisseling. *Parallel scientific computation*. Oxford University Press Oxford, 2004.

[9] Peter Brucker. *Scheduling algorithms*. Springer, 2007.

[10] Camil Demetrescu and Giuseppe F Italiano. "Fully dynamic transitive closure: breaking through the $O(n^2)$ barrier". In: *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE. 2000, pp. 381–389.

[11] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (1959), pp. 269–271.

[12] Jack Edmonds. "Paths, trees, and flowers". In: *Canadian Journal of Mathematics* 17.3 (1965), pp. 449–467.

[13] David Eppstein, Zvi Galil, and Giuseppe F Italiano. *Dynamic graph algorithms*. Springer, 1998.

[14] James Robert Evans and Edward Minieka. *Optimization algorithms for networks and graphs*. Vol. 1. CRC Press, 1992.

[15] Harold N Gabow. "Data structures for weighted matching and nearest common ancestors with linking". In: *Proceedings of the first annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. 1990, pp. 434–443.

[16] Eliyahu M Goldratt. *Critical chain*. The North River Press, 1997.

[17] Teofilo Gonzalez and Sartaj Sahni. "Flowshop and jobshop schedules: complexity and approximation". In: *Operations research* 26.1 (1978), pp. 36–52.

[18] Jaap-Henk Hoepman. "Simple distributed weighted matchings". In: *arXiv preprint cs/0410047* (2004).

[19] Arthur B Kahn. "Topological sorting of large networks". In: *Communications of the ACM* 5.11 (1962), pp. 558–562.

[20] Fredrik Manne and Rob H Bisseling. "A parallel approximation algorithm for the weighted maximum matching problem". In: *Parallel Processing and Applied Mathematics*. Springer, 2008, pp. 708–717.

[21] Fredrik Manne and Morten Mjelde. "A self-stabilizing weighted matching algorithm". In: *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2007, pp. 383–393.

[22] Jayakrishnan Nair, Adam Wierman, and Bert Zwart. "Scheduling for the tail: Robustness versus Optimality". In: *48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE. 2010, pp. 969–976.

[23] Ofer Neiman and Shay Solomon. "Simple deterministic algorithms for fully dynamic maximal matching." In: *STOC*. 2013, pp. 745–754.

[24] Seth Pettie and Peter Sanders. "A simpler linear time 2/3-ε approximation for maximum weight matching". In: *Information Processing Letters* 91.6 (2004), pp. 271–276.

[25] Michael Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2012.

[26] Stella CS Porto and Celso C Ribeiro. "A tabu search approach to task scheduling on heterogeneous processors under precedence constraints". In: *International Journal of high speed computing* 7.1 (1995), pp. 45–71.

[27] Robert Preis. "Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs". In: *STACS 99*. Springer. 1999, pp. 259–269.

[28] Dorry L Segev et al. "Kidney paired donation and optimizing the use of live donor organs". In: *JAMA: the Journal of the American Medical Association* 293.15 (2005), pp. 1883–1890.