N.J. van der Kooy

Project planning with temporal and resource constraints

Bachelorthesis

Supervisor: Dr. F.M. Spieksma

June 5, 2015

Mathematical Institute, University of Leiden

Contents

Abstract

Within the world of operational research, project scheduling plays a large and important part. Being able to plan a project in such a way that is deemed optimal, by minimizing a given objective, is a challenging mathematical problem. Depending on the constraints placed on the project, there might not even exist any straightforward algorithm to obtain an optimum.

For time constrained problems, polynomial time algorithms exists to calculate the most cost effective solution for any provided set of jobs. However, if jobs are additionally required to compete for resources, such a general solution does not exist.

However, this thesis studies a method in which the Lagrangian relaxation of a resource constrained project can be efficiently solved by transforming it into an equivalent time constrained problem. This time constrained problem is subsequently solved by computing the minimum cut in a derived directed graph.

1 Introduction

Project scheduling problems are some of the most fundamental optimisation-related mathematical problems. Due to their generality a wide variety of problems can be formulated as project scheduling problems. Hence, the question of how to (efficiently) solve these problems is an important one.

In [6], the NP-hard project scheduling problem is studied where jobs are not only related by time-constraints, but additionally by resource constraints. An optimal solution for this problem is approximated by performing a so-called Lagrangian relaxation on the resource constraints. This results in a subproblem that is shown to be equivalent to a project scheduling problem with only start-time dependent costs, which in turn is shown to be efficiently solvable by transforming it into a minimum cut problem.

In this thesis, we first examine the project scheduling problem without resource constraints. It is shown how this problem is transformed to a minimum cut problem, and how this can then be efficiently solved. This is covered in §2.
In this same chapter, we also examine aspects of the time-constrained problem not covered by [6]. Namely, we study the claimed feasibility test of a given problem (§2.1). The proof of this claim was self-developed, and is given here. In addition, we explicitly define algorithms that were determined to be usable to calculate values (the *Earliest feasible start times* and *Latest feasible start times*) necessary to solve a problem (§2.3).

Following this, in §3, we look at project scheduling problems expanded by adding resource constraints. This additions makes the problem NP-hard, and we examine how these problems are relaxed into a time-dependent project scheduling problem using Lagrangian relaxations. For this section, an implementation of the algorithm described in [6] was developed.

However, as we will see, the choice of so called Lagrangian multipliers is not a simple task, but one that requires a significant amount of numerical analysis. In §3.2 we discuss the process of Lagrangian relaxations, while in §3.4 we study the actual steps performed in the calculation of these multipliers.

The goal of this thesis is not to expand on the methods provided in [6]. Rather, it is to fill in the gaps by proving assumptions made in the article, as well as providing explicit methods where this is not done in the article itself.

2 Project schedules without resource constraints

In this chapter, we consider one of the most basic project scheduling problems: those without resource constraints, but *with* temporal constraints. This means that we look at a project consisting of multiple jobs, which are potentially interconnected through the requirement that the start times of these jobs need to satisfy some criteria. Consider for example the building of a house, where the job of "building the walls" cannot be performed until after the job of "setting the foundation" has been completed.

Consider a set of jobs $J = \{0, ..., n\}$.

Definition 2.1. A *schedule* is a vector $S = (S_0, ..., S_n)$, $S_j \in \mathbb{N}$, indicating the starting time of job $j \in J$.

Definition 2.2. For every job $j \in J$, its *processing time* $p_j \in \mathbb{N}$ is the time required for the job to be completed.

In order to be valid, a schedule S needs to satisfy given so-called *time-lags.* A time-lag identifies jobs that are temporally dependent on each other. This is a common occurrence in project scheduling problems, since often jobs cannot be carried out until another task has been completed.

Let $L \subseteq J \times J$ be a set of time lags (i, j) between jobs $i, j \in J$.

Definition 2.3. For all $(i, j) \in L$, $d_{ij} \in \mathbb{Z}$ is the *length of the time lag* between jobs i and j.

What definition 2.3 says is that for an $(i, j) \in L$, a positive d_{ij} indicates that job i must have been started for at least d_{ij} time units before job j can start. For a negative d_{ij}, it means that i can start no more than $-d_{ij}$ time units after job j.

In other words, a valid schedule S has to satisfy $S_i + d_{ij} \leq S_j \leq S_i - d_{ji}$. We consider jobs 0 and n as artificial jobs that indicate the project start and the project end. Consequently, $p_0 = p_n = 0$, and $S_0 = 0$. In addition, we assume that $(0, j) \in L$ for all $j \in J$ with $d_{0j} = 0$ and $(j, n) \in L$ for all $j \in J$ with $d_{jn} = p_j$. This first set of edges is needed to indicate that every job must start after the start of the project as a whole, while the second set ensures that the start time of job n indicates the completion time of the last 'real' job.

Once a feasible schedule S is determined (if it exists), its cost w_S can be determined through the cost w_{jt} incurred when job j is started at time t. Here, $t = 0, 1, ..., T$ and T is a predetermined upper bound on the project makespan. In other words, after time T, each job has to be completed. Hence, $S_j \leq T - p_j$ for all $j \in J$.

When a valid schedule has been found, the total cost of the schedule can be found by adding the cost of every individual job:

$$w_S = \sum_{j \in J} w_{j,S_j}. \tag{1}$$

If we want to complete all jobs with minimum costs, our objective becomes to find the smallest possible w_S. Let $\mathcal{S} = \{\text{Feasible schedules } S\}$.

$$w(J) = \min_{\mathcal{S}} w_S = \min_{\mathcal{S}} \left(\sum_{j \in J} w_{j,S_j} \right), \tag{2}$$

obtained with schedule

$$\arg\min_{\mathcal{S}} w_S = \arg\min_{\mathcal{S}} \left(\sum_{j \in J} w_{j,S_j} \right). \tag{3}$$

2.1 Feasibility

A first question that comes to mind, when trying to minimize the costs in a project scheduling problem, is whether a feasible solution exists in the first place. In [6], it is claimed that this can be verified through the use of Bellman's algorithm.[1] What follows is a verification of this claim. It is based on looking at a graph with jobs $i \in J$ as nodes, and time lags $(i, j) \in L$ as edges of length d_{ij} between nodes i and j. We assume that infeasibility of a problem never stems from an insufficient time horizon (T), but rather from conflicts within the time lags.

Definition 2.4. The *time-lag graph* of a set of jobs is a weighted and directed graph where every job $j \in J$ is represented by a node, with every time-lag in L defining an edge. The weight of edge $(i, j) \in L$ is defined as d_{ij}.

Proposition 2.5. *A time-constrained project schedule has a feasible solution* $\Leftrightarrow$ *every cycle in its time lags has non-positive length.*

Proof. Throughout this proof, when working in a cycle of positive length that contains both i and j we will use the notation D_{ij} for the shortest length from job i to job j while going through this cycle. This length is then either d_{ij}, when jobs i and j are directly connected, or the sum of the edges between these two jobs. Note that since there is only one way to get from i to j in a cycle, D_{ij} is unique. Additionally, by defining D_{ij} as *the shortest* length from job i to j, we ensure we do not select the path that walks the cycle multiple times. Since we only use D_{ij} in the context of positive length cycles, we know the shortest length is well defined.

$\Rightarrow$
Take a feasible time-constrained project schedule S, and assume that the graph of its time lags contains a cycle of positive length. There are two possibilities: either all edges in this positive cycle are of non-negative length, or at least one edge (i, j) in the cycle is of negative length.

In the first case, this means that for jobs i and j for which (i, j) is an edge in this cycle, there is a path from j to i with length $D_{ji} > 0$. In addition, we know $d_{ij} \geq 0$.

$D_{ji} > 0 \Rightarrow S_i > S_j$, while $d_{ij} \geq 0 \Rightarrow S_j \geq S_i$. Since this can't both be true, we have a contradiction.

In the second case, there are jobs i and j for which (i, j) is an edge in this cycle and $d_{ij} < 0$. Since the cycle as a whole has positive length, the path from j to i has a length longer than $-d_{ij}$: $D_{ji} > -d_{ij}$. This implies

$$S_i \geq S_j + D_{ji} \Rightarrow S_j \leq S_i - D_{ji} < S_i + d_{ij}. \tag{4}$$

Since (4) breaks definition 2.3, our assumption of a cycle of positive length is incorrect.

$\Leftarrow$
Take a time-constrained project schedule whose graph of time lengths only contains cycles of non-positive length. We obtain a new graph by multiplying all edge lengths by -1 and, due to our assumption, this graph now only contains cycles of non-negative length. This means that Bellman's algorithm ([1]) can be performed on this graph - the algorithm works for any graph where negative cycles do not exist - using the node corresponding to job 0 as the source node.

This algorithm gives us the shortest path from the source node to each other node, which is the same as the longest path in the original graph (obtained by multiplying both the edges as the optimal solution by -1).

A job can start once all of the time constraints related to this job have been complied with. This is determined by the longest (chain of) time constraint(s), so the distance from the source to a node determined by Bellman's algorithm gives a feasible start time for that node. Since the algorithm can find a distance for each connected node if the cycles in the original graph are of non-positive length, this gives a feasible start-time for each node, and thus a feasible solution for the problem. $\square$

Definition 2.6. For every job $j \in J$, e_j is the *Earliest Feasible Starting Time* for job j. A way to determine its value is explained in corollary 2.7.

Corollary 2.7. *As a consequence of above proof, we find that the earliest time a job can possibly start (its* Earliest Feasible Start Time*) is equal to its distance from the source node found by Bellman's algorithm performed on the negated time-lag graph as described above. This value is used in algorithm 2 in order to* *construct the so-called min-cut graph, which in turn is used to solve the project scheduling problem*

We do not take into account the possibility that the processing time of a job added to its earliest feasible start time exceeds the time horizon, since we assume that time horizon T is never the cause of infeasibility. The computation of the Earliest Start Times does give us a lower bound on T however:

$$\min T = \max_{j \in J} (e_j + p_j), \tag{5}$$

since T can never be lower than the highest time needed for every job to complete as quickly as possible.

2.2 Formulation as an Integer Programming Problem

If we assume a time-feasible solution for a time-constrained project scheduling problem exists, we can formulate it as an integer programming problem in order to try and solve it. We do so as follows.

First we introduce variables x_{jt} where $j \in J, t \in \{0, ..., T\}$.

$$x_{jt} = \begin{cases} 1 & \text{if job } j \text{ starts at time } t, \\ 0 & \text{otherwise.} \end{cases}$$

These x_{jt} are used to define a (potentially unfeasible) schedule. This then allows us to formulate the following integer linear program:

$$\begin{aligned}
\text{minimize} \quad & w(x) = \sum_j \sum_t w_{jt} x_{jt} && && \text{(6a)} \\
\text{subject to} \quad & \sum_t x_{jt} = 1, && j \in J, && \text{(6b)} \\
& \sum_{s=t}^{T} x_{is} + \sum_{s=0}^{t+d_{ij}-1} x_{js} \leq 1, && (i,j) \in L, t = 0, ..., T, && \text{(6c)} \\
& x_{jt} \geq 0, && j \in J, t = 0, ..., T, && \text{(6d)} \\
& x_{jt} \text{ integer}, && j \in J, t = 0, ..., T. && \text{(6e)}
\end{aligned}$$

In the integer programming problem above, $w(x)$ indicates the cost of schedule x (w_{jt} is included in the sum iff job j starts at time t, through the x_{jt}). Constraint (6b) enforces each job to get performed exactly once, and constraint (6c) enforces the temporal constraints, by making sure that the time period between S_i and $S_i + d_{ij}$ does not contain S_j.

2.2.1 Feasibility and a Total Unimodular constraint matrix

The reason we are able to easily find a solution to above problem, in the way that will be described in §2.3, is because of the Total Unimodularity of the constraint matrix in the above programming problem. This ensures us of an integer optimal solution.

2.3 Solving the Integer Programming Problem

As displayed in [6, §2.2], a scheduling problem can be transformed into a directed graph, through which an optimal solution can be found by finding the minimum cut of this graph.

What follows is a complete algorithm for this process, useful for numerical solving of Project Scheduling Problems. This will be addressed in §3.

Before we construct the directed graph through which we can determine an optimal solution, we must first calculate earliest feasible start times $e(j)$ and latest feasible start times $l(j)$ for all jobs $j \in J$. These values indicate, as their name suggest, the earliest and latest times $t \in \{0, ..., T\}$ at which a job can start while still maintaining the possibility of a feasible solution.

2.3.1 Earliest feasible start times

As explained in corollary 2.7, the Earliest feasible start times of a schedule are automatically determined when we check whether a feasible solution exists using Bellman's algorithm. Therefore, when we are at this step in the process it is no longer necessary to calculate these e_j again.

2.3.2 Latest feasible start times

In order to determine the latest feasible start times $l(j)$ for $j \in J$, we have developed the following algorithm. It works by repeatedly looking at a job i and one of its successors j. Note that job j is a successor of i if $(i, j) \in L$. If $l(i)$ potentially forces job j to start beyond $l(j)$ (so $l(i) + d_{ij} > l(j)$), we know that $l(i)$ is too high. Therefore, if such a situation is found, $l(i)$ is reduced appropriately.

Note that on line 6 in algorithm 1, "shift" is the act of taking the first element from a set, and removing it from the set itself.

Algorithm 1 Determining Latest Feasible Start Times

```
 1: for jobs $j \in \{1, ..., n-1\}$ do
 2:     $l(j) \leftarrow (T - (p_j - 1))$ # Initialization so that job $j$ never finishes after $T$
 3: end for
 4: processSet = $\{1, ..., n-1\}$                # Track which $l(j)$'s need updating
 5: while processSet $\neq \emptyset$ do
 6:     PROCESS(shift(processSet))
 7: end while
 8: procedure PROCESS($i$)
 9:     for $(i, j) \in L$ do
10:         if $l(i) + d_{ij} > l(j)$ then
11:             $l(i) \leftarrow l(j) - d_{ij}$   # Reduce $l(i)$ so it doesn't interfere with job $j$
12:             processSet $\leftarrow$ processSet $\cup \{k | (k, i) \in L\}$        # We update an
    $l(i)$ based on $l(j)$, where $j$ is a successor to $i$. Therefore, if we change $l(i)$,
    we need to reprocess its predecessors.
13:         end if
14:     end for
15: end procedure
```

We know this algorithm always terminates, since the algorithm is only carried out if a feasible solution exists. This means that the $l(i)$ can never be lowered beyond $e(i)$ (since the latest feasible start time cannot occur before the earliest feasible start time), so at some point each job will have a correct latest starting time ($l(i) + d_{ij}$ will never be higher than $l(j)$ for some successor j).

The complexity of this algorithm is easily determined. Each job will be processed a maximum of T times (If its earliest and latest feasible start times are both 0, it gets initialized to T, and every processing iteration only reduces the latest feasible start time by 1). Since there are $n - 1$ jobs, this gives us a complexity $\mathcal{O}(nT)$. In practice the complexity will be much lower however, since with most time constraints the latest feasible start time will not be too far from the initial upper limit of $T - (p_j - 1)$.

2.3.3 Constructing the min-cut graph

Now we have determined the earliest and latest feasible start times, we are able to construct a directed graph the minimum cut of which will give us an optimal solution to the project scheduling problem. The nodes and arcs will be created as described in [6, §2.2].

Algorithm 2 Determining the Min-Cut graph

```
1: Create nodes $a$ and $b$ representing the virtual start and end jobs
2: for $j \in \{1, ..., n-1\}$ do
3:     for $e(j) \leq t \leq l(j) + 1$ do
4:         Create a node $v_{jt}$
5:     end for
6: end for
7: for $j \in \{1, ..., n-1\}$ do
8:     Create an edge between $a$ and $v_{j,e(j)}$ with infinite capacity
9:     Create an edge between $v_{j,l(j)+1}$ and $b$ with infinite capacity
10:    for $e(j) \leq t \leq l(j)$ do
11:        Create an edge between $v_{jt}$ and $v_{j,t+1}$ with capacity $w_{jt}$
12:    end for
13:    for $i \in \{1, ..., n-1\}$ do
14:        if $(i, j) \in L$ then
15:            if $e(i) + 1 \leq t \leq l(i)$ AND $e(j) + 1 \leq t + d_{ij} \leq l(j)$ then
16:                Create an edge between $v_{it}$ and $v_{j,t+d_{ij}}$ with infinite capacity
17:            end if
18:        end if
19:    end for
20: end for
```

Here, every edge of finite capacity represents a job being performed at a certain time. When a (minimum) cut has been found, the cut is related to a solution of (6) by defining:

$$x_{jt} = \begin{cases} 1 & \text{if } (v_{jt}, v_{j,t+1}) \text{ is in the cut,} \\ 0 & \text{otherwise.} \end{cases}$$

In addition, note the constraints placed on the temporal edges of infinite capacity, namely placing an edge of infinite capacity between nodes v_{it} and $v_{j,t+d_{ij}}$ when $e(i) + 1 \leq t \leq l(i)$ `AND` $e(j) + 1 \leq t + d_{ij} \leq l(j)$.

The reason this works can be best visualised by, for example, looking at the min-cut graph generated by example 2.8 (see figure 2). How this min-cut graph is used is by finding finite capacity cuts that split the graph. The infinite capacity temporal edges work to enforce the temporal constraints defined in the problem.

As proven in [6, Theorem 1], the capacity of the cut is equal to the value $w(x)$ of the corresponding scheduling solution. Therefore, the minimum cut gives us an optimal solution of the problem.

Example 2.8 (From scheduling problem to min-cut graph)**.** The best way to understand the process described in §2.3.1, §2.3.2 and §2.3.3 is by seeing the process on an example. Take the following graph, with nodes representing jobs and edge lengths representing time lags:

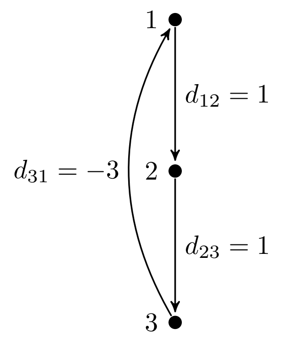

Figure 1: Given jobs with their time lags laid out in graph form

In addition, we have been given job processing times ($p_1 = 2, p_2 = 1, p_3 = 1$) and costs for starting a job at a certain time, see the following table of w_{it}:

w_{it}	$t = 0$	$t = 1$	$t = 2$	$t = 3$
$i = 1$	30	2	15	4
$i = 2$	50	1	10	5
$i = 3$	90	9	13	2

Performing Bellman's algorithm gives us the following information:

```
Earliest feasible starting time of job 1: 0
Earliest feasible starting time of job 2: 1
Earliest feasible starting time of job 3: 2
```

Then, performing algorithm 1 on the graph using the processing time information gives us:

```
Latest feasible starting time of job 1: 1
Latest feasible starting time of job 2: 2
Latest feasible starting time of job 3: 3
```

Finally, we can perform algorithm 2. This results, finally, in the following min-cut graph:

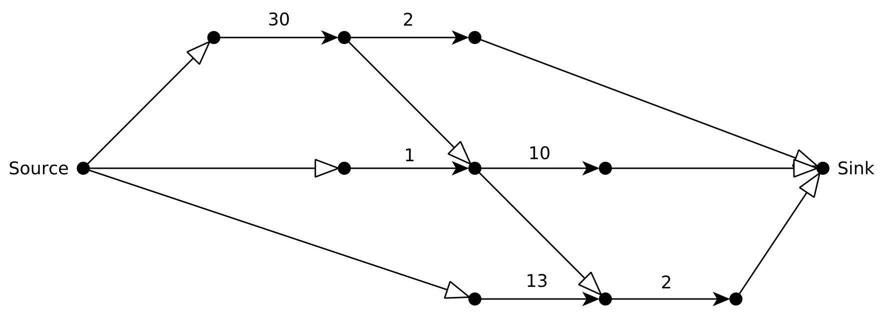

Figure 2: The min-cut graph, obtained from algorithm 2

In figure 2, an edge indicated by a black-headed arrow has its capacity indicated above of it, while a white-headed arrow indicates an edge of infinite capacity. Note that there are no infinite capacity edges from job 3 to job 1 to enforce d_{31} anywhere. This is the result of that, in this example, d_{31} can never be broken without violating time-horizon T.

2.3.4 Calculating the solution from the min-cut graph

Now we have found the min-cut graph, we will determine the optimal solution by calculating the maximum flow using the push-relabel algorithm [4]. First, we will study example 2.8 to find out what we expect the optimal solution of that example to be. Secondly, we will look at how the push-relabel algorithm actually calculates the maximum flow within a graph. Lastly, we prove that the maximum flow in the min-cut graph indeed corresponds to a general optimal solution for the project scheduling problem.

Example 2.8 (Continued)**.** To determine the optimal solution of this example, we need to understand the meaning of the nodes and edges in figure 2. Essentially, every edge indicates a certain job starting at a certain time. An edge of finite capacity reflects the cost of starting a certain job at the time corresponding to the originating node. Finding a schedule can therefore be seen as making a cut in the graph, where the edges in the cut determine at what times jobs are executed, while the sum of the edges cut gives us the total cost. The temporal constraints being modeled using edges of infinite capacity prevent us from breaking these constraints (Since this would mean cutting an infinite capacity edge, and thus making the costs of our schedule 'infinite', so infeasible). There are four valid cuts, shown in figure 3.

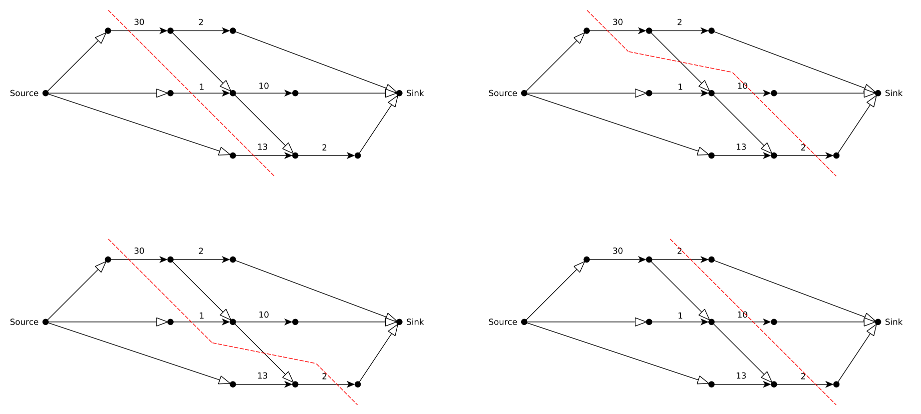

Figure 3: Red lines indicate valid cuts for the min-cut graph from example 2.8

Note that the two cuts which appear to cut the temporal edges of infinite capacity are valid, because the direction in which the edges are cut means that the infinite capacity edge travels from the side of the sink to the side of the source, so it does not affect the solution. This gives us four possible solutions, where the bottom-right image shows us the solution of minimal capacity: 14. This occurs when we start job 1 at $t = 1$, job 2 at $t = 2$ and job 3 at $t = 3$.

2.3.4.1 The Push-Relabel algorithm

In order to calculate the maximum flow in the min-cut graph, we use the Push-Relabel algorithm. The generic version of this algorithm, which we will be using, has a time complexity of $\mathcal{O}(V^2 A)$. The version used in [6] has a time complexity of $\mathcal{O}(VA\log(V^2/A))$, but since this algorithm is less efficient in practice [4] and significantly more cumbersome to implement, we will be using the generic version.

If we refer to our min-cut graph as $G(V, A)$, the actual algorithm is described in algorithm 3 below:

Algorithm 3 The Push-Relabel algorithm

```
1: for $(u, v) \in A$ do
2:     $f(u, v) \leftarrow 0$                    # $f(u, v)$ indicates the flow over an edge
3: end for
4: for $(s, v) \in A$, where $s$ is the source node do
5:     $f(s, v) \leftarrow c(s, v)$              # $c(u, v)$ indicates the capacity of an edge
6: end for
7: for $u \in V$ do
8:     $h(u) \leftarrow 0$                       # $h(u)$ is the *height* of a node
9:     $e(u) \leftarrow f(s, u)$    # $e(v)$ is the nodes excess, which can only come from
   flow from the source
10: end for
11: for Source node $s$ do
12:     $h(s) \leftarrow |V|$
13:     $e(s) \leftarrow \infty$
14: end for
15: while We can perform a Push or Relabel operation do
16:     Perform this operation
17: end while
18: procedure PUSH$(u, v)$
19:     if $e(u) > 0$ AND $h(u) = h(v) + 1$ then
20:         $\Delta \leftarrow \min\{e(u), c(u, v) - f(u, v)\}$
21:         $f(u, v) \leftarrow f(u, v) + \Delta$
22:         $f(v, u) \leftarrow f(v, u) - \Delta$
23:         $e(u) \leftarrow e(u) - \Delta$
24:         $e(v) \leftarrow e(v) + \Delta$
25:     end if
26: end procedure
27: procedure RELABEL$(u)$
28:     if $e(u) > 0$ AND $h(u) \le h(v)\ \forall v$ with $f(u, v) < c(u, v)$ then
29:         $h(u) \leftarrow \min\{h(v) + 1 \mid v \text{ with } f(u, v) < c(u, v)\}$
30:     end if
31: end procedure
```

In words, the algorithm works in a few phases:

1. Every edge going away from the source is *saturated*, meaning that for source node s, the flow $f(s, v)$ over edge (s, v) is set to capacity $c(s, v)$.

2. Every node gets an attribute known as its *height*. This is initialized to $|V|$ (the number of nodes) for the source, and 0 for every other node. In addition, we define the *excess* of a node v as

$$e(v) = \begin{cases} \sum_{u \in V} f(u, v), & \forall v \in V \backslash \{\text{source}\} \\ \infty & v = s. \end{cases}$$

Since we start with a flow of $c(s,v)$ for all edges (s,v) where s is the source, our excesses are initialised as

$$e(v) = \begin{cases} \sum_{(s,v)\in V} c(s,v), & \forall v \in V\backslash\{\text{source}\}, \\ \infty & v = s, \end{cases}$$

which conforms to the previous definition. We call any node $v \in V\backslash\{\text{source, sink}\}$ *active* if $e(v) > 0$, since the amount of incoming flow is more than the outgoing amount of flow. Therefore, it still needs to be processed in some way. While performing the algorithm, the Push operation slowly reduces the excess of all nodes to 0.

3. *Push* and *relabel* operations are performed on every active node. The push operation effectively looks at how much excess a node v has, and distributes this over nodes that can be reached from v. The end result of these operations is that if saturating all starting edges gave us too much flow (which is almost always the case), the push and relabel operations reduce the amount of flow that is sent from the source.

Once the algorithm completes, we can simply look at the flow over edges coming from the source, and this will give us the maximum flow from the source to the sink. The proof of the correctness of this algorithm can be found in [8]. If we perform this algorithm on the graph from example 2.8, we find that this indeed gives us a maximum flow of 14. What follows is the proof that the maximum flow indeed always gives us the optimal solution to the project scheduling problem.

Theorem 2.9. *The optimal solution to a project scheduling problem with temporal constraints is equal to the maximum flow in its corresponding min-cut graph. The source nodes of the edges in the cut indicate the start time of each job in this optimal schedule.*

Proof. When the maximum flow of a min-cut graph has been determined, we know from the Max-flow Min-cut theorem that this gives us the minimum capacity cut that, when removed in a specific way from the network, causes the situation that no flow can pass from the source to the sink.[7]

Since every job provides a path from the source to the sink (with an edge from source to $v_{i,e(i)}$, through all nodes v_{it} and finally from $v_{i,l(i)+1}$ to sink), we know that at least one edge from each job needs to be removed in order to block flow from passing from the source to the sink. This means that any minimum cut will — for every job $j \in J$ — include at least one edge $(v_{jt}, v_{j,t+1})$.

Once this observation is made, [6, Lemma 1] and [6, Theorem 1] prove the theorem. □

3 Resource-constrained project schedules

After looking at project scheduling problems that are purely temporally constrained, we now expand our problem to include resource constraints. In addition to their temporal constraints, jobs now need resources while they process. These resources are potentially needed by multiple jobs, meaning that previously feasible solutions now become infeasible due to jobs being processed in parallel competing for a specific resource.

In this model, there is a finite set R of resources, and the capacity of resource $k \in R$ is denoted by R_k. Note that in our model, R_k is time-independent: At every point in time, R_k of resource k is available. In addition the resources are renewable, so when a job is finished using a resource, the resource again becomes available for other jobs to use.

The resource constraints are attached to a job by defining r_{jk} as the amount of resource k needed by job j during its processing. Like the resources themselves, these constraints are time-independent.

Finally, we redefine our objective function. In this scheduling problem, instead of aiming to reduce our costs, we aim to complete our project as quickly as possible.

In §2 we defined objective function $w_S = \sum_{j \in J} w_{j,S_j}$ (1). We will keep using the notation w_S as the 'value' of a schedule S, but we will now redefine it to

$$w_S = S_n. \tag{7}$$

Recall that job n starting indicated that every job had been completed, so we want to reduce S_n as much as possible. Of course, our objective $w(J)$ remains the same - we still want to minimize w_S, so equation (2) remains valid.

For the resource-constrained project scheduling problem, there is no polynomial time algorithm to find an optimal solution, unless NP = ZPP (the class of so-called Zero-error Probabilistic Polynomial time problems)[3].

3.1 Formulation as an Integer Programming Problem

Like the purely temporally constrained problem, we are going to formulate the resource constrained problem as an integer programming problem, and use this to try and solve the problem.

As we are now trying to minimize our project makespan, we obtain the objective function of

$$\text{minimize} \qquad w(x) = \sum_t t x_{nt} \tag{8}$$

where n is our artificial 'last job'.

Since the time constraints and other restrictions remain valid and part of the problem, the objective function is subject to (6b), (6c), (6d) and (6e).

In addition, we now of course have our resource constraints. These are modeled by

$$\sum_j r_{jk} \left(\sum_{s=t-p_j+1}^{t} x_{js} \right) \leq R_k, k \in R, t = 0, ..., T. \tag{9}$$

These inequalities ensure that all jobs being processed at time t simultaneously do not consume more resources than available.

Note that in (9) — as well as all following equations where we walk over s — the starting index of s needs to be 0 or greater. A starting index of $s = t - p_j + 1$ can therefore always be read as $s = \max\{t - p_j + 1, 0\}$.

3.2 Lagrangian Relaxation

As mentioned above, the resource-constrained project scheduling problem has no polynomial-time solution.

This is because, unlike the purely temporally-constrained problem, where the constraints are Totally Unimodular as explained in §2.2.1 which guarantee a polynomial time solvable problem, the resource constraints take away this guarantee.

However, by introducing Lagrangian multipliers $\lambda = (\lambda_{tk}), t \in \{0, ..., T\}$ we obtain a Lagrangian relaxation of our problem. Since a relaxation allows us to violate some of our constraints (in this case the resource constraints), the optimal solution of the relaxation is potentially infeasible in the original problem. However, these solutions do give us lower bounds on our solution, and we will eventually use these lower bounds and solutions of the relaxed problem in order to find an actual feasible solution.

What we do with our Lagrangian relaxation is take some of our constraints, in this case the resource constraints, and incorporate them into our objective function while scaling them using our multipliers λ. This in order to create a matrix of constraints that is again Totally Unimodular. Since our λ are non-negative, any resource constraints that are broken generally increase our solution. Hence they function as a 'penalty' on our minimization.

We rewrite our resource constraints as follows:

$$\begin{aligned} \sum_j r_{jk}\left(\sum_{s=t-p_j+1}^{t} x_{js}\right) \leq R_k, k \in R, t = 0, ..., T \\ \sum_j r_{jk}\left(\sum_{s=t-p_j+1}^{t} x_{js}\right) - R_k \leq 0, k \in R, t = 0, ..., T. \end{aligned} \tag{10}$$

Then, instead of requiring every inequality to be met, we instead try to minimize them, which is the same as trying to minimize the sum of all inequalities:

$$\text{minimize } \sum_j r_{jk}\left(\sum_{s=t-p_j+1}^{t} x_{js}\right) - R_k, k \in R, t = 0, ..., T.$$

Now we sum over the entire time horizon, and incorporate our multiplier λ

$$\text{minimize } \sum_t \lambda_{tk}\left(\sum_j r_{jk}\left(\sum_{s=t-p_j+1}^{t} x_{js}\right) - R_k\right), k \in R.$$

Next, we seperate the equation into two sums, resulting in

$$\text{minimize } \sum_t \left(\sum_j r_{jk}\left(\sum_{s=t-p_j+1}^{t} x_{js}\right)\right)\lambda_{tk} - \sum_t \lambda_{tk} R_k, k \in R.$$

Finally we also sum over all $k \in R$, giving

$$\text{minimize } \sum_{k\in R}\sum_t \left(\sum_j r_{jk}\left(\sum_{s=t-p_j+1}^{t} x_{js}\right)\right)\lambda_{tk} - \sum_{k\in R}\sum_t \lambda_{tk} R_k,$$

which can be rewritten to the equal

$$\text{minimize } \sum_j\sum_t \left(\sum_{k\in R} r_{jk} \sum_{s=t}^{t+p_j-1} \lambda_{sk}\right) x_{jt} - \sum_t\sum_{k\in R} \lambda_{tk} R_k. \tag{11}$$

If we add (11) to (8), we obtain the following Lagrangian subproblem:

$$\text{minimize } \sum_t t x_{nt} + \sum_j\sum_t \left(\sum_{k\in R} r_{jk} \sum_{s=t}^{t+p_j-1} \lambda_{sk}\right) x_{jt} - \sum_t\sum_{k\in R} \lambda_{tk} R_k, \tag{12}$$

again subject to (6b), (6c), (6d) and (6e).

If we now introduce weights

$$w_{jt} = \begin{cases} \sum_{k \in R} r_{jk} \sum_{s=t}^{t+p_j-1} \lambda_{sk} & \text{if } j \neq n, \\ t & \text{if } j = n, \end{cases}$$

we can rewrite (12) as

$$\text{minimize } \sum_j \sum_t w_{jt} x_{jt} - \sum_t \sum_{k \in R} \lambda_{tk} R_k \tag{13}$$

subject to (6b), (6c), (6d) and (6e).

For a given λ, the term $\sum_t \sum_{k \in R} \lambda_{tk} R_k$ is constant, so we are purely minimizing over the job weights and start times. If we compare this to (6a), we see that (13) is a project scheduling problem with temporal constraints and start-time dependent costs, just as the problem discussed in §2. In addition, since weights w_{jt} depend on λ, which are non-negative, the weights are non-negative as well, which allows us to solve (13) using the techniques discussed before.

3.3 Relating the Lagrangian relaxation and the resource-constrained project scheduling problem

For any λ, the optimal solution of (13) is a lower bound on the value of our resource-constrained project scheduling problem defined by (6b), (6c), (6d), (6e), (8) and (9): either the optimal solution of the relaxation complies with all resource constraints and it is also an optimal solution to the project scheduling problem, or some resource constraints are broken and the optimal solution of the relaxation is lower than the optimal solution of the resource-constrained project.

We shall denote the value of an optimal solution for the Lagrangian relaxation as w_λ for a fixed λ. From this, we define the *Lagrangian dual* as $\max\limits_{\lambda \geq 0} w_\lambda$.

In addition to being a lower bound on (13), the Lagrangian dual is in general also a lower bound on the LP-relaxation of (13), since this relaxation is still constrained by the resource restrictions.

However, since the time constraints are Totally Unimodular (see §2.2.1), we find that the optimal solution of the LP-relaxation in fact *equals* the Lagrangian dual [5, Corollary 9.1].

Although knowing the value of the Lagrangian dual does not give us the λ that produces this value, this result is still important. Since the optimal solution of the LP-relaxation can be determined in polynomial time, it gives us the means to determine how close w_λ is to the Lagrangian dual for a certain λ.

3.4 Computing the Lagrangian Multipliers

Our objective now is to compute our multipliers λ so that the Lagrangian relaxation using this value approaches its maximum, and is hence as close as possible to the optimal solution of the resource-constrained project.

This computation is done in two steps. First, we calculate the target value of the Lagrangian relaxation (which from now on we will call w^*). Secondly, we use this value to find a value for λ that gets us in the neighborhood of this target value.

3.4.1 Computing the target value

As explained in §3.3, this the target value of the Lagrangian relaxation is the value of the Lagrangian dual, which is equal to the LP-relaxation of (13).

Example 2.8 (Continued)**.** We turn again to our previous example. Now, however, we need to add extra information, namely our resource constraints. We define one resource constraint, with $R_1 = 10$. In addition, we define $r_{11} = 6, r_{21} = 6, r_{31} = 2$. Note how this implies jobs 1 and 2 cannot execute simultaneously, since this would demand 12 units from resource 1, while only $R_1 = 10$ units are available.

When only time constraints were involved, an optimal solution to our new objective function (8) would have trivially been to start every job on its earliest feasible starting time. In this case, that would have resulted in $w(x) = 3$. However, because of the choice of our resource constraints this is no longer possible.

Recall that $p_1 = 2$ and $p_2 = 1$. In addition, $e(1) = 0$ and $e(2) = 1$ (see §2.3.3). This would mean our resource constraints are violated, since jobs 1 and 2 are running simultaneously at $t = 1$. In fact, manually looking for feasible solutions by studying the valid cuts in figure 3 shows us that we in fact only have *one* solution that complies with our resource constraints, namely where job 1 starts at $t = 0$, job 2 at $t = 2$ and job 3 at $t = 3$, which would be complete at $t = 4$.

Therefore, when we calculate optimal value of the LP-relaxation, we expect $w^* \leq 4$.

This calculation was performed by the `linprog()` function in `Matlab`.

This calculation resulted in

```
fval =

    3.7862
```

or $w^* = 3.7862$, which meets our expectations that $w^* \leq 4$.

3.4.2 Computing the actual multiplier

Once w^* has been calculated as in §3.4.1, we can now begin actually calculating our λ by making use of the method described in [6, §3.5], which itself is based on a standard subgradient method described in [2, §6.3]

This is an iterative process where, starting with a λ^0 (which we take to be the matrix of ones), we calculate $\lambda^{i+1} := \left[\lambda^i + \delta^i g^i\right]^+$, whereby:

- $[\cdot]^+$ indicates the nonnegative part of a vector. In other words, every negative value is changed to 0.
- $g^i_{k,t} = \sum_{j\in J} r_{jk} \left(\sum_{s=t-p_j+1}^{t} x^i_{js} \right) - R_k \left(1 - \sum_{s=0}^{t} x^i_{ns} \right).$
- $\delta^i = \dfrac{\delta(w^* - w_{\lambda^i}(x^i))}{||g^i||^2}$. Here, δ is a parameter that is slowly reduced as the improvement of our w_λ slows, w^* is the value of the Lagrangian dual as calculated in §3.4.1, and $||g^i||^2$ is the sum of squares of our elements of g^i. This denominator is important, as it normalizes the product $\delta^i g^i$ based on the size of our resource constraints.

This is the point where all of the theory described in §2 comes into play. In the definition of δ^i, we see the term $w_{\lambda^i}(x^i)$. This term indicates the optimal solution of (13) constrained by (6b), (6c), (6d), (6e), (8) and (9), as described in §3.3. However, since the non-constant term of (13) is a purely time constrained problem, we find a solution of $w_{\lambda^i}(x^i)$ by applying the method described in §2.3.

Example 2.8 (Continued)**.** In order to put the above theory into practice, a program was written to perform the above calculations for a number of iterations — in this case 10.

However, due to the limited number of available options and the small scale of the problem, we find that the results on this problem are quite uninteresting: in the first two iterations, the solution that is found is the same as the optimal solution found is the one where each job starts at its Earliest Feasible Start Time. This is indeed the best solution when looking only at temporal constraints, since it is minimizes w_S. However, it is clearly not valid since it violates our only resource constraint.

However, from iteration 3 onwards we immediately jump to the only valid solution in our problem:

```
Start time of job 1: 0
Start time of job 2: 2
Start time of job 3: 3
```

after which the generated solutions no longer change in new iterations.

This is clearly not a very interesting solution. Indeed, since the fact that our found solution is also feasible in the resource constraints means that we also no longer have to use any of the techniques discussed in [6, §4] to transform the found solution into a feasible one. Therefore, as also mentioned in §4, a next step in this research should be to look at a problem where the found solution of the Lagrangian relaxation is less obvious.

4 Conclusion

In this thesis, we studied part of the technique proposed in [6] to solve NP-hard project scheduling problems, specifically those involving both temporal and resource constraints. This technique involves an algorithm to solve problems that are purely temporally constrained, and algorithm is then used in order to efficiently solve an approximation of the resource constrained problem obtained by performing a Lagrangian relaxation.

In doing so, proofs were given for unbacked claims made in the article and the technique was applied to a basic but insightful example.

The application of the technique on this simple example, however, does not produce any results that cannot be easily observed at a simple glance. Follow up research could look at the application of the described technique on a more sizable project scheduling problem, and look at complications found in doing so. In addition, no real attention was paid to the choice of our initial λ, for which [6] also does not provide any insights. This is a second angle in which follow-up research could be performed.

Since the solutions of the Lagrangian relaxations are generally infeasible for the actual problem because of broken resource constraints, a final step in the finding of a solution for a resource constrained problem is taking the solution of the Lagrangian relaxation and transforming it into a feasible solution to the original problem. This process is discussed in [6, §4]. The study of this process, as well as their description in the article, could be a final way in which follow up research on this thesis could be performed.

References

[1] Richard Bellman. On a routing problem. Technical report, DTIC Document, 1956.

[2] Dimitri P Bertsekas. *Nonlinear programming.* Athena scientific Belmont, 1999.

[3] Uriel Feige and Joe Kilian. Zero knowledge and the chromatic number. In *Computational Complexity, 1996. Proceedings., Eleventh Annual IEEE Conference on*, pages 278–287. IEEE, 1996.

[4] Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[5] Lodewijk CM Kallenberg. Besliskunde 4, 2009.

[6] Rolf H Möhring, Andreas S Schulz, Frederik Stork, and Marc Uetz. Solving project scheduling problems by minimum cut computations. *Management Science*, 49(3):330–350, 2003.

[7] Wikipedia. Max-flow min-cut theorem — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Max-flow_min-cut_theorems`, 2014. [Online; accessed october–december 2014].

[8] Wikipedia. Pushrelabel maximum flow algorithm — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm#Correctness`, 2015. [Online; accessed october 2014–february 2015].