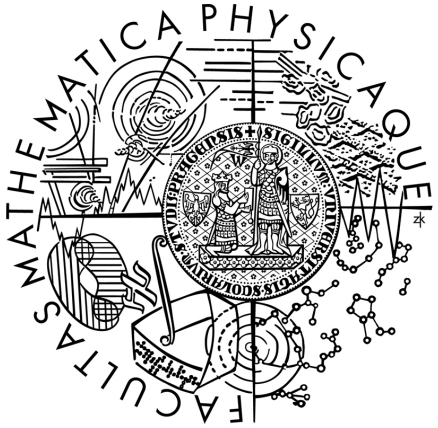
Charles University, Prague
Faculty of Mathematics and Physics

MASTER THESIS



Temporal networks

Rudolf Vlk

Department of Theoretical Computer Science and Mathematical Logic
Supervisor: doc. RNDr. Roman Barták, Ph.D.
Study Program: Computer Science

Acknowledgements

I would like to thank my supervisor doc. RNDr. Roman Barták, Ph.D. for his insights, guidance and patience.

I hereby declare that I wrote this thesis by myself and listed all used sources. I agree with making the thesis publicly available.

Prague, 7.8.2009 Rudolf Vlk

Název práce: Temporal networks
Autor: Rudolf Vlk
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: doc. RNDr. Roman Barták, Ph.D.
e-mail vedoucího: bartak@ktiml.mff.cuni.cz
Abstrakt: Integrace plánování a rozvrhování vyžaduje hledání nových prístupů k problému rozvrhování. Rozvrhovací systém musí být schopen poskytnout užitečné informace plánovači, aby se zabránilo vytváření neuskutečnitelných plánů. Pro rozvrhování založené na splňování omezujících podmínek je možné definovat vlastní filtrační pravidla a tak zefektivnit řešící algoritmus. Pokud filtrační pravidla využívají informace sdílené plánovačem a rozvrhovacím systémem (např. precedenční a nebo temporální podmínky), výstup těchto pravidel je mozné poskytnout plánovači, který je může s výhodou využít. V této práci je navržena filtrační metoda, která využívá temporální vztahy mezi aktivitami alokovanými na jeden nebo více disjunktivních zdrojů. Práce také popisuje sadu propagačních pravidel založených na kombinaci ruzných filtračních technik.
Klíčová slova: planování a rozvrhování, temporální sítě, temporální podmínky, filtrační pravidla

Title: Temporal networks
Author: Rudolf Vlk
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: doc. RNDr. Roman Barták, Ph.D.
Supervisor's e-mail address: bartak@ktiml.mff.cuni.cz
Abstract: Integration of planning and scheduling requires new approaches to the scheduling problem. The scheduler must be able to provide useful information for the planner in order to avoid generation of unfeasible plans. In constraint-based scheduling it is possible to define custom filtering rules that improve the solving procedure. If the filtering rules exploit the information shared by the planner and the scheduler (e.g. precedence or temporal constraints), the outcome of these rules can be used to provide useful hints for the planner. This work presents a filtering technique that exploits temporal relations between a set of activities allocated to one or more disjunctive resources. The work also presents a set of propagation rules for constraint-based scheduling based on various filtering techniqes.
Keywords: planning and scheduling, temporal networks, temporal constraints, filtering rules

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Constraint satisfaction 3
2.2 Scheduling and constraint satisfaction 6
2.3 Introduction to Temporal Networks 8
2.3.1 Temporal constraint satisfaction problem 8
2.3.2 Simple temporal problem 11
2.4 Advanced algorithms for temporal networks 16

3 Related works **17**
3.1 Integrating planning and scheduling 17
3.2 Integrating planning with constraint-based scheduling 19
3.2.1 Energy precedence constraint 21
3.2.2 Filtering rules based on energy precedence constraint . 22

4 Basic concepts **25**
4.1 The Problem 25
4.2 Temporal graph without optional nodes 26
4.3 STP with optional variables 29
4.4 Temporal graph with optional nodes 35

5 Algorithms for temporal graph **36**
5.1 Incremental minimality algorithms 36
5.1.1 Temporal graph without optional nodes 36
5.1.2 Temporal graph with optional nodes 43
5.2 Propagation of resource constraints 47

6 Propagation rules **52**
6.1 Constraint Model 52
6.2 Initialization 53

6.3 Propagation rules for temporal constraints 53
6.4 Enhancing propagation of temporal constraints 56
6.4.1 Making domains of temporal variables finite 56
6.4.2 Detectable precedences 57
6.5 Energy precedence constraint 58

7 Conclusion **64**

Bibliography **67**

Chapter 1

Introduction

Automated planning and scheduling receive a lot of attention in Artificial Intelligence research. The focus of planning is to choose what actions need to be executed in order to accomplish some objectives and scheduling cares about allocation of activities into time and space. Traditional approach to the planning and scheduling problem is sequential. First, the planner finds a set of activities and then the scheduler attempts to schedule execution of the activities within limited time and using limited resources. The drawback of this sequential approach is lack of communication between the planner and the scheduler — a feasible schedule for the plan often does not exist, because the planner does not consider temporal and resource limitations.

One way to alleviate this problem is to integrate planning and scheduling. In this approach, the planner consults the partial plans it creates with the scheduler. This way it is possible to discover and discard the unfeasible partial plans more quickly. Unfortunately, classical schedulers (even though they perform very well on pure scheduling problems) are not suitable for integration with planners and thus new scheduling approaches are necessary to support integration of planning and scheduling.

One such approach is to integrate reasoning on precedence and resource constraints. This allows the scheduler to efficiently prune time windows of activities even when very little commitments are made by the planner and thus the scheduler is able to provide useful feedback.

A natural way of extending the above idea is to consider general temporal constraints instead of simple precedence relations.

In this work we propose a set of filtering rules that integrate reasoning on temporal and resource constraints and that also allow some activities to be optional (meaning that it is possible to flexibly decide whether the activity will be executed or not). It is possible to use these filtering rules in pure scheduling as well as in integrated planning and scheduling.

The work is organized as follows. Chapter 2 provides a brief introduction into the topics that reader should be familiar with in order to understand the rest of the work: the constraint satisfaction technology, constraint-based scheduling and the temporal networks.

In Chapter 3 we present existing works in the area of integration of planning and scheduling and we focus on works that exploit precedence relations between activities in constraint-based scheduling. We describe a filtering technique called "energy precedence constraint" and existing propagation rules based on this technique.

Chapters 4 and 5 provide a theoretical basis for the propagation rules introduced in Chapter 6.

In Chapter 4 we formalize the addressed problem and propose a structure — called a temporal graph — that represents the problem in the propagation rules. We also propose a generalization of temporal networks that combines temporal constraints with optional activities. Finally, we extend the notion of a temporal graph to include optional activities.

In Chapter 5 we propose an incremental algorithm that propagates changes of temporal constraints in the temporal graph. Then we introduce a notion of generalized energy precedence constraint that uses the relationship between precedence and resource constraints to adjust the temporal relations between activities. We also introduce an algorithm that calculates weights of the edges in the temporal graph based on the generalized energy precedence constraint.

In Chapter 6 we propose a set of filtering rules that work with the temporal graph and integrate propagation of temporal and resource constraints between a set of optional activities.

Chapter 2

Preliminaries

2.1 Constraint satisfaction

Constraint satisfaction is a common denominator in the two most important topics discussed in this work. First, the framework for reasoning on temporal relations (called temporal constraint satisfaction problem, TCSP for short) introduced later in this chapter is formulated as an instance of a generic constraint satisfaction problem (CSP). Second, we propose new algorithms that support integration of planning and scheduling. These algorithms are formulated as filtering rules for a scheduler based on a constraint solving technology. Thus, understanding the basic principles of constraint satisfaction is probably the most important prerequisite needed to read our work.

Definition 2.1. *Constraint satisfaction problem* is triple (X, D, C) where:

- $X = \{X_1, \ldots, X_n\}$ is a finite set of variables.
- D is a set of domains for these variables.
- C is a set of constraints restricting possible combinations of the values assigned to variables — every constraint is a relation over the variables' domains.

We say that tuple $(x_1, \ldots, x_n)$ is a *solution* of CSP if the values are from corresponding domains and the assignment $X_1 = x_1, \ldots, X_n = x_n$ satisfies all the constraints in C. Thus the task of constraint solver is to find a value for each variable from the corresponding domain in such a way that all the constraints are satisfied [7].

Example 2.1 (Problem of n-queens). A simple example of a problem that can be formulated as CSP is the problem of n-queens. The task is to place n queens on $n \times n$ board so that the queens do not attack each other.

To create a CSP model of this problem, we must specify, which variables are used, what are their domains and what constraints restrict possible combinations of their values. The aim is to create a model whose every solution would correspond to a solution of the problem of n-queens. Example of such a model follows.

Example 2.2 (A CSP model of the problem of n-queens). An attempt to solve the problem of n-queens is an assignment of a position to every queen. Position of i-th queen is represented by a pair of variables $row(i)$ and $col(i)$. The domain of variable $row(i)$ should contain all allowed row positions for i-th queen, i.e. $\{1, \ldots, n\}$ — the set of rows on the chessboard. The domain of variable $col(i)$ should similarly contain all allowed column positions for i-th queen, which is again the set $\{1, \ldots, n\}$.

The constraints of the model must ensure, that no two queens attack each other. Specifically, every pair of queens must occupy different rows, different columns and different diagonals of the chessboard. Formally, we can say that for every pair of queens $i \neq j$, the following constraint is specified:

$$row(i) \neq row(j) \wedge col(i) \neq col(j) \wedge |row(i) - row(j)| \neq |col(i) - col(j)|$$

The constraint solvers typically solve CSPs using a combination of depth-first search and domain filtering. Domain filtering is a process of removing values from the domains that do not satisfy some constraint. Each constraint has a filtering algorithm assigned to it that does this job for the constraint, and these algorithms communicate via the domains of the variables – if a filtering algorithm shrinks a domain of some variable, the algorithms for constraints that use this variable propagate the change to other variables until a fixed point is reached or until some domain becomes empty. Such a procedure is called a (generalized) arc consistency.

When all domains are reduced to singletons then the solution is found. If some domain becomes empty then no solution exists. In all other cases the search procedure splits the space of possible assignments by adding a new constraint (for example by assigning a value to the variable) and the solution is being searched for in sub-spaces defined by the constraint and its negation (other branching schemes may also be applied).

Listing of Algorithm 2.1 shows an example of how a solving procedure might look.

The procedure *solve*() accepts four arguments — a set of decision variables X, a set of domains for these variables D, a set of constraints that must be satisfied C and set P of variables that had their domains changed by previous invocation of *solve*() and need to be propagated.

The procedure first ensures propagation of domain changes caused by previous invocation by calling the *consistency*() procedure. If the propagation

Algorithm 2.1 Example of a solving procedure written in pseudocode

```
solve(X, D, C, P) →
D' ← consistency(P, D, C)
if D' = fail then
  fail
if All domains in D are singletons then
  return solution
V ← choose_variable(X, D', C)
(D_V^1, D_V^2) ← split_domain(D_V)
D'' ← (D' \ {D_V}) ∪ {D_V^1}
R ← solve(X, D'', C, {V})
if R ≠ fail then
  return R
else
  D''' ← (D' \ {D_V}) ∪ {D_V^2}
  R' ← solve(X, D''', C, {V})
  return R'
```

fails, the procedure also fails and if all domains become singletons, then the solution is found and returned. If neither of these cases occurs, the procedure chooses a variable and splits its domain. Then it recursively calls itself in order to search for solution in subspace where either one or the other part of variable's domain is used.

The consistency procedure invoked by the solving procedure might look like the one listed as Algorithm 2.2.

Algorithm 2.2 Example of generalized consistency procedure

```
consistency(Q, D, C) →
D' ← D
while Q not empty do
  X ← select and remove variable from Q
  for all constraints C_i affected by change in domain of variable X do
    D' ← propagate(C_i, D')
    if some domain in D' is empty then
      fail
    Q ← Q ∪ {Y | Y is a variable affected by propagation of C_i}
return D'
```

In this procedure, Q represents a queue of variables whose domain was changed and not propagated. In every iteration of the while-loop, one va-

riable is chosen and all constraints that may be affected by change in its domain have their corresponding filtering rules invoked (procedure *propagate*()). This rule may prune domains of other variables, which are then added into Q for later processing. The loop ends when a fixed point is reached, in which case the propagation is complete or when some domain becomes empty, which means that no solution exists.

One of the most notable advantages of using constraint satisfaction to solve problems is that with improvements of general CSP solvers, all problems are solved more effectively. The formalism is very generic and thus can be used to model a large scale of problems. Moreover, it is often possible to exploit specific knowledge of the modeled problem and define problem-specific filtering rules in order to help solver prune domains more effectively.

The constraint solvers usually provide an interface for user-defined filtering algorithms so the users may extend the capabilities of the solvers by writing their own filtering algorithms [15]. This interface consists of two parts: triggers and propagators. The user should specify when the filtering algorithm is called — a *trigger*. This is typically a change of domain of some variable, for example when the lower bound of the domain is increased, the upper bound is decreased, or any element is deleted from the domain. The *propagator* then describes how this change is propagated to domains of other variables. The constraint solver provides procedures for access to domains of variables and for operations over the domains (membership, union, intersection, etc.) The output of the propagator is a proposal how to change domains of other variables in the constraint. The algorithm may also deduce that the constraint cannot be satisfied (fail) or that the constraint is entailed (exit).

In this work, we address the problem of scheduling integrated with temporal networks and optional activities and we use constraint satisfaction technology to model and solve this problem. Our main contribution lies in a set of new filtering algorithms that prune domains of decision variables and thus allow a more efficient search for solutions of this problem. These algorithms are written in the form of propagation rules, which allow their straightforward implementation in any constraint solver that supports user-defined global constraints.

2.2 Scheduling and constraint satisfaction

Scheduling is an area of operations research that concerns itself with the problem of allocating a set of activities on available resources and finding an execution time for each activity. The usage of resources is usually restricted in some way (for example, some resources may allow processing of only one

activity at a time, others may diminish with every use and need to be replenished regularly and so on) and there is usually some kind of optimization criterion (like minimalization of makespan, lateness, cost, etc.)

Scheduling has a wide spectrum of applications from school timetabling through logistics to manufacture management. It is also often used in conjunction with planning, we discuss this more thoroughly in the next chapter.

Scheduling problems belong to the area of combinatorial optimisation problems so they can be naturally described as constraint satisfaction problems. One of traditional modelling approaches uses variables to describe the activities. In particular, there are three variables identifying the position of activity in time: the start time, the end time and the processing time (duration). Let A be an activity, we denote these variables S_A, E_A, p_A. We expect the domains for these variables to be discrete (e.g. natural numbers) where the release time and deadline make natural bounds for them. Note that if the processing time of the activity is constant, then one variable is enough to locate the activity in time. We still prefer to use all three variables to simplify description of the constraints [1].

The time variables of every activity are bound by constraint $S_A+p_A = E_A$. Time dependencies between the activities can also be naturally described using constraints like $E_A \leq S_B$ (this constraint means that activity A must be completed before activity B starts).

Besides activities, resources also play a significant role in scheduling. There are three types of resources mentioned in this work: reservoir, discrete and unary resources.

The *reservoir resource* is the most general of the three. A reservoir resource is a multi-capacity resource that can be consumed and/or produced by so called resource events. A reservoir has an integer maximal capacity an may have an initial level. As an example of a reservoir, you can think of a fuel tank.

A *discrete resource* is a special kind of reservoir resource that is used over some time interval: a certain quantity of resource is consumed at the start time of the activity and the same quantity is released at its end time. Discrete resources are also called cumulative or sharable resources in the scheduling literature. A discrete resource has a known maximal capacity profile over time. They allow us, for example, to represent a pool of workers whose availability may change over time.

A *disjunctive resource* (also called a *unary resource*) is a discrete resource with unit capacity. It imposes that all the activities requiring the same unary resource are totally ordered. This is typically the case of a machine that can process only one operation at a time. Unary resources are the simplest and the most studied resources in scheduling as well as in AI planning [11].

If multiple resources are considered in the scheduling problem, variable res_A is used to represent the resource to which activity A is allocated. The resource limitation may also be modeled using constraints. For example, if activities A and B are allocated to the same unary resource, we would add constraint $E_A \leq S_B \vee E_B \leq S_A$.

You can find a much more detailed introduction into constraint-based scheduling in [1].

2.3 Introduction to Temporal Networks

In [8] authors introduced temporal constraint satisfaction problem (TCSP) and its simplified instance, called simple temporal problem (STP). Since temporal networks and their integration with scheduling problem is the main focus of this work we present some basic definitions and some results from their work. If reader is interested in proofs and more detailed discussion, he is kindly asked to refer to the original article.

2.3.1 Temporal constraint satisfaction problem

Definition 2.2. *Temporal constraint satisfaction problem* (TCSP) is tuple (X, D, C) where:

- X is set $\{X_1, \ldots, X_n\}$ of variables representing time points
- D is set $\{D_1, \ldots, D_n\}$ of domains, D_i is domain for variable X_i
- C is set of temporal constraints.

Each temporal constraint is represented by a pairwise disjoint set of intervals:

$$\{l_1, \ldots, l_k\} = \{[a_1, b_1], \ldots, [a_k, b_k]\}$$

There are two types of temporal constraints: unary and binary. Unary constraint T_i restricts the domain of variable X_i to the given set of intervals, i.e. it represents a disjunction

$$a_1 \leq X_i \leq b_1 \vee \ldots \vee a_k \leq X_i \leq b_k$$

Binary constraint T_{ij} restricts the permissible values for the distance $X_j - X_i$, i.e. it represents a disjunction

$$a_1 \leq X_j - X_i \leq b_1 \vee \ldots \vee a_k \leq X_j - X_i \leq b_k$$

Definition 2.3. *Network of binary constraints* (a *binary* TCSP) is TCSP in which only binary constraints are allowed.

In order to represent unary constraints in binary TCSP, a special variable X_0 is introduced to represent "beginning of the world". We treat all unary constraints T_i as binary constraints T_{0i}. For simplicity we assume that $X_0 = 0$.

In this manner we may represent restrictions on domain D_i of variable X_i via binary constraint T_{0i}. Thus D_i needs to specify only general set of permitted values, e.g. whether the domain is continuous or discrete.

The network of binary temporal constraints can be represented by a *directed constraint graph* $G = (V, E)$. Node $i \in V$ represents variable X_i and edge $i \to j$ represents temporal constraint T_{ij}. Edge $i \to j$ is labeled by the set of intervals in T_{ij}.

Example 2.3. John goes to work either by car (30-40 minutes), or by bus (at least 60 minutes). Fred goes to work either by car (20-30 minutes), or in a carpool (40-50 minutes). Today John left home between 7:10 and 7:20 and Fred arrived at work between 8:00 and 8:10. We also know that John arrived at work about 10-20 minutes after Fred left home.

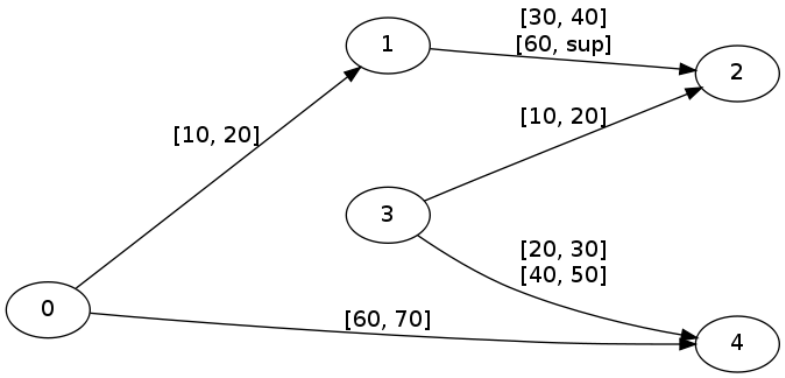
We may represent information given in Example 2.3 using TCSP. The interesting time points are:

- X_1 - time when John leaves home
- X_2 - time when John arrives at work
- X_3 - time when Fred leaves home
- X_4 - time when Fred arrives at work

As discussed before, we also introduce a "beginning of the world" variable, X_0. Let us assume that X_0 corresponds to a time of 7:00. Then the following temporal constraints can be defined:

- $T_{12} = \{[30, 40], [60, \infty]\}$
- $T_{34} = \{[20, 30], [40, 50]\}$
- $T_{01} = \{[10, 20]\}$
- $T_{04} = \{[60, 70]\}$
- $T_{32} = \{[10, 20]\}$

Figure 2.1: Constraint graph for Example 2.3



Example of constraint graph that represents this TCSP is on Figure 2.1.

Note that temporal constraint T_{ij} implies an equivalent temporal constraint T_{ji}. In the above example, $T_{32} = \{[10, 20]\}$ implies constraint $T_{23} = \{[-20, -10]\}$. Only one of these equivalent constraints is usually specified in the constraint graph.

Definition 2.4. We say that tuple $(x_1, \ldots, x_n)$ is a *solution* of a TCSP if the assignment $\{X_1 = x_1, \ldots, X_n = x_n\}$ satisfies all of the temporal constraints.

Definition 2.5. We say that value v is a *feasible value* for variable X_i, if there exists a solution in which $X_i = v$. The set of all feasible values of a variable is called the *minimal domain*.

Definition 2.6. We say that a TCSP is *consistent* if it has at least one solution.

We define a partial order among the binary temporal constraints in the following manner.

Definition 2.7. Let T and S be a pair of binary temporal constraints. We say that T is *tighter* than S, denoted $T \subseteq S$, if for every interval $I \in T$ there exists an interval $J \in S$ such that $I \subseteq J$.

Informally, $T \subseteq S$ if every pair of values allowed by T is also allowed by S.

Minimum of this order (the tightest constraint) is the *empty constraint*, $\emptyset$. If the network contains an empty constraint, then it is trivially inconsistent. Maximum of this order (the most relaxed constraint) is the *universal*

constraint $\{[-\infty, \infty]\}$. Edges corresponding to a universal constraint are usually omitted from the constraint graph (since they are always satisfied, we do not need to take them into consideration).

From the partial order between binary temporal constraints we may derive the partial order between temporal networks.

Definition 2.8. Let T and S be two temporal networks on the same set of variables. We say that T is *tighter* than S, denoted $T \subseteq S$, if for all i, j, $T_{ij} \subseteq S_{ij}$.

Definition 2.9. We say that two temporal networks are *equivalent*, if they have the same set of solutions.

Definition 2.10. Let T be a temporal network. We say that network M is *minimal representation* of T (or *minimal network*), if M is equivalent with T and for every other S equivalent with T, $M \subseteq S$. Constraints specified by the minimal network are called *minimal constraints*.

Definition 2.11. Let $T = (X, D, C)$ be a TCSP and let $Y = \{Y_1, \ldots, Y_k\}$ be a set of variables such that $Y \subseteq X$. We say that $C' = \{T_{ij} | X_i \in Y \wedge X_j \in Y\}$ is a *restriction* of C to Y, denoted $C|Y$.

Definition 2.12. Let $T = (X, D, C)$ be a TCSP and $Y \subseteq X$. Let A be an assignment $\{Y_1 = y_1, \ldots, Y_k = y_k\}$ that assigns a value to every variable in Y. We say that assignment A is *locally consistent*, if it satisfies all constraints from $C|Y$ (i.e. all constraints from C applicable to Y)

Definition 2.13. We say that a TCSP is *decomposable*, if every locally consistent assignment to any subset of variables can be extended to a solution of the TCSP.

If a temporal network is decomposable, then we may construct a solution without backtracking. If value of every variable is chosen with regard to previous assignments, then decomposability guarantees that it will be also possible to find a consistent value for the rest of the unassigned variables.

2.3.2 Simple temporal problem

Definition 2.14. Simple temporal problem (or simple temporal network, STP or STN for short) is a tuple (X, D, C) where X is a set of variables $X_1, \ldots X_n$, D is a set of domains for these variables and C is a set of constraints. Each constraint T_{ij} restricts permissible values for the distance $X_j - X_i$ to be in interval $[a_{ij}, b_{ij}]$. Formally, it represents constraint

$$a_{ij} \leq X_j - X_i \leq b_{ij}$$

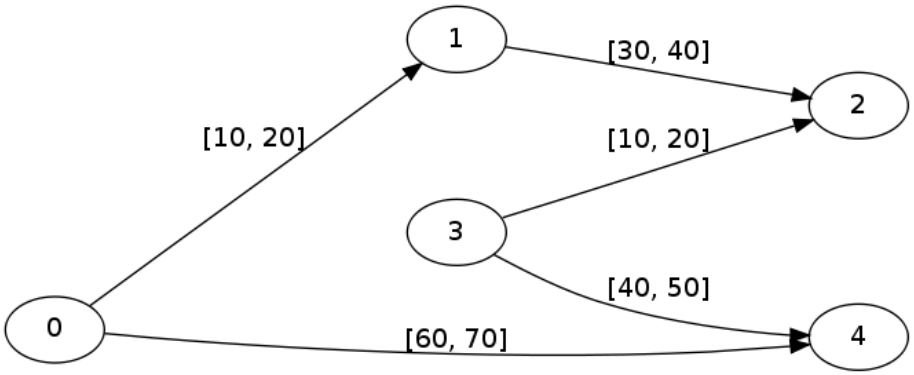
It is easy to see that STP is a special case of binary TCSP in which every constraint specifies a single interval $[a_{ij}, b_{ij}]$. In the constraint graph for an STP, every edge is labeled with exactly one interval.

The following example is a simplified version of Example 2.3, that can be represented using STN.

Example 2.4. John goes to work by car for 30-40 minutes and Fred goes to work in a carpool for 40-50 minutes. Today John left home between 7:10 and 7:20 and Fred arrived at work between 8:00 and 8:10. John arrived at work about 10-20 minutes after Fred left home.

The constraint graph of the temporal network for Example 2.4 is on Figure 2.2.

Figure 2.2: Constraint graph for Example 2.4



Solving an STP amounts to solving the following set of linear inequalities:

$$X_j - X_i \leq b_{ij} \text{ for every } T_{ij}$$

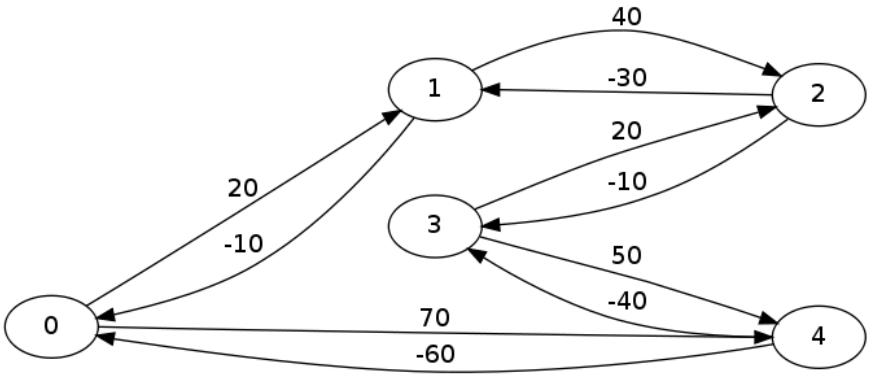
$$X_i - X_j \leq -a_{ij} \text{ for every } T_{ij}$$

A set of linear inequalities can be solved for example using the (exponential) simplex method. However, the special class of linear inequalities characterizing the STP admits a simpler solution. The inequalities are given a convenient graph representation, to which a shortest path algorithm can be applied.

Definition 2.15 (Distance graph)**.** Let T be an STP and let $G = (V, E)$ be its constraint graph. We say that a directed edge-weighted graph $G_d = (V, E_d)$ is a *distance graph* of T if it has the same set of nodes as G and if for every edge in E there are two (directed) edges in E_d. Edge $i \to j$ is labeled by weight c_{ij}, representing linear inequality $X_j - X_i \leq c_{ij}$.

An example of distance graph is on Figure 2.3

Figure 2.3: Distance graph for Example 2.4



Consider a path from i to j in distance graph G_d, $P_{ij} = (i_0, i_1, \ldots, i_k)$, where $i_0 = i$ and $i_k = j$. Edges on the path represent a set of inequalities

$$X_{i_1} - X_{i_0} \leq c_{i_0 i_1}$$

$$X_{i_2} - X_{i_1} \leq c_{i_1 i_2}$$

$$\ldots$$

$$X_{i_k} - X_{i_{k-1}} \leq c_{i_{k-1} i_k}$$

The sum of these inequalities yields

$$X_{i_k} - X_{i_0} \leq \sum_{j=1}^{k} c_{i_{j-1} i_j}$$

Thus, the path P_{ij} induces a new constraint on distance $X_j - X_i$

$$X_j - X_i \leq \sum_{j=1}^{k} c_{i_{j-1} i_j}$$

It is clear, that intersection of induced constraints over all paths from i to j is

$$X_j - X_i \leq d_{ij}$$

where d_{ij} is the length of the shortest path from i to j.

Based on this observation, the following condition on consistency of an STP can be established.

Theorem 2.1. *An STP is consistent if and only if its distance graph contains no negative cycles.*

Proof. Suppose that there is a negative cycle, consisting of nodes $i_1, \ldots, i_k = i_1$. Summing the inequalities along the cycle yields $X_{i_1} - X_{i_1} < 0$, which cannot be satisfied.

Conversely, if there is no negative cycle in distance graph, then the shortest path between each pair of nodes is well-defined. For any pair of nodes, i and j, the shortest paths satisfy $d_{0j} \leq d_{0i} + c_{ij}$ and thus

$$d_{0j} - d_{0i} \leq c_{ij}$$

Hence the tuple $(d_{01}, d_{02}, \ldots, d_{0n})$ is a solution of the STP. □

Definition 2.16. Let T be an STP and let G_d be its distance graph. A complete graph on the same set of nodes as G_d, where every edge $i \rightarrow j$ is labeled with d_{ij} — the length of the shortest path from i to j in G_d — is called *d*-graph of T.

Definition 2.17. Let T be an STP and let G be its *d*-graph. An STP M is called *d-graph-induced* from T, if it has the same set of variables and its temporal constraints are defined as:

$$M_{ij} = [-d_{ji}, d_{ij}]$$

(where d_{ij} is the weight of edge $i \rightarrow j$ in *d*-graph).

A few trivial consequences of the Definition 2.17 are:

- T and M are equivalent (i.e. they are just different formulations of the same problem)
- $M \subseteq T$ (constraints in M are tighter than constraints in T)
- *d*-graph of T is both distance graph and *d*-graph of M

Theorem 2.2 (Decomposability)**.** *Let T be a consistent STP and M its d-graph-induced STP. M is decomposable.*

Theorem 2.2 provides an efficient algorithm for assembling a solution to a given STP T. First we construct the *d*-graph-induced network M. Then we simply assign to each variable any value that satisfies the constraints in M relative to previous assignments (starting with $X_0 = 0$). Decomposability guarantees that such a value can always be found, regardless of the order of assignment. Since M and T are equivalent, solution of M is also a solution of T.

A second by-product of decomposability is that the domains and constraints in M are minimal.

Corollary 2.3. *Let T be a consistent STP. The set of feasible values for variable X_i (or minimal domain of X_i) is $[-d_{i0}, d_{0i}]$ (d_{ij} is the weight of edge $i \to j$ in d-graph of T).*

Proof. According to Theorem 2.2, the assignment $X_0 = 0$ can be extended by assigning any value $v \in [-d_{i0}, d_{0i}]$ to X_i. This assignment can be extended to a full solution. Thus, v is a feasible value. □

Corollary 2.4. *Let T be a consistent STP and M its d-graph-induced STP. M is minimal network equivalent with T.*

The proof of Corollary 2.4 can be found in [8].

Consider the distance graph of Figure 2.3. Since there are no negative cycles, the corresponding STP is consistent. The shortest path distances (or weights of edges in *d*-graph) d_{ij} are shown in table 2.1.

Table 2.1: Shortest path distances for distance graph of Figure 2.3

	0	1	2	3	4
0	0	20	50	30	70
1	-10	0	40	20	60
2	-40	-30	0	-10	30
3	-20	-10	20	0	50
4	-60	-50	-20	-40	0

The minimal domains are: $10 \leq X_1 \leq 20$; $40 \leq X_2 \leq 50$; $20 \leq X_3 \leq 30$; $60 \leq X_4 \leq 70$.

The minimal equivalent network M is given in Table 2.2.

Table 2.2: Minimal network corresponding to Figure 2.3

	0	1	2	3	4
0	[0]	[10, 20]	[40, 50]	[20, 30]	[60, 70]
1	[-20, -10]	[0]	[30, 40]	[10, 20]	[50, 60]
2	[-50, -40]	[-40, -30]	[0]	[-20, -10]	[20, 30]
3	[-30, -20]	[-20, -10]	[10, 20]	[0]	[40, 50]
4	[-70, -60]	[-60, -50]	[-30, -20]	[-50, -40]	[0]

The *d*-graph of an STP can be constructed by applying *Floyd-Warshall's* all-pairs-shortest-paths algorithm to the distance graph. It is listed as Algorithm 2.3. The algorithm runs in time $O(n^3)$, and detects negative cycles

simply by examining the sign of the diagonal elements d_{ii}. Therefore it constitutes a polynomial time algorithm for determining the consistency of an STP, and for computing both the minimal domains and the minimal network. Once d-graph is available, assembling a solution requires only $O(n^2)$ time, because each successive assignment needs to be checked against previous assignments and is guaranteed to remain unaltered. Thus, finding a solution can be achieved in $O(n^3)$ time.

Algorithm 2.3 All-pairs-shortest-paths algorithm

```
for i := 1 to n do
  d_ii ← 0
for i, j := 1 to n do
  d_ij ← c_ij
for k := 1 to n do
  for i, j := 1 to n do
    d_ij ← min(d_ij, d_ik + d_kj)
    if i = j ∧ d_ii < 0 then
      fail
```

2.4 Advanced algorithms for temporal networks

There are many enhancements of reasoning on STP available. The Floyd-Warshall's algorithm introduced in section 2.3.2 can be replaced by Johnson's algorithm [6], which works faster on sparse networks.

Interesting enhancements are available for triangulated networks. One approach in this category is *Partial Path Consistency* introduced in [3]. State-of-the-art solver ΔSTP introduced in [18] is further improved by P^3C in [14]. A method exploiting tree-decomposition structure of triangulated STN is given in [4]

If it is enough to ensure consistency of the temporal network, *directed path consistency* can be used [7].

An efficient algorithm for compiling and dynamically scheduling TCSPs is introduced in [16].

Chapter 3

Related works

This chapter presents the existing works in two fields from which our work derives. The first section describes works that concern themselves with integration of planning and scheduling. Emphasis of this section is on introducing so called energy precedence constraint, which is used in later chapters.

3.1 Integrating planning and scheduling

This work aspires to contribute into a field that receives a lot of attention in Artificial Intelligence research recently — integration of planning and scheduling.

Planning traditionally concerns itself with finding out what actions need to be executed in order to achieve specified objectives. It focuses on causal links between the activities and — being difficult enough as it is — leaves aside the issues of optimization or handling of temporal and resource constraints. Scheduling, on the other hand, expects that there is a fixed set of activities and that they need to be allocated on limited resources and executed in limited time. The scheduler finds such an allocation of activities to resources and such start times for activities that all given restrictions are honored.

Authors of [10] point out the following practical needs related to actions, problem states and resources that traditional approaches to planning do not meet:

- Non-instantaneous duration of actions — planning systems assume that all actions have the same duration and they are instantaneous. This is clearly not true for real problems.
- Action effect persistence — although planning systems work with infi-

nite persistence (an effect is held until another action removes it), the effects do not always behave this way in real situations. For instance, in the action of boiling some water, the effect of 'hot water' has finite persistence. In these cases, it becomes necessary to add explicit constraints to represent this kind of persistence.

- Ordering constraints in action abstract contexts — although planners can manage the execution order of the actions (causal links), this kind of order is only qualitative and quite simple. Nevertheless, in more abstract macro-actions, more complex qualitative constraints, such as overlaps, starts,finishes, equal, etc., and metric constraints among actions can appear.

- Temporal constraints over problem states — planning systems only deal with obtaining the sequence of actions to achieve the goal. However, they do not consider the instant of time when each action is going to be executed or when a state is achieved. Moreover, constraints over the execution of the entire plan, such as due dates which appear in real problems, are also not considered.

- Shared resource management — many of the conflicts among actions that appear during the planning process are due to the non-simultaneous use of shared resources. Since usual planners do not have the necessary knowledge to manage resource availability, they use artificial strategies (such as mutual exclusion relationships, mutexes in Graphplan) to avoid the non simultaneous use of each resource. However, explicit resource management becomes necessary to guarantee the allocation of resources (taking into account constraints on their availability and usage) and to allow us to optimize their usage. For instance, it may be necessary to allocate the resources to be used only during the morning because it implies a lower cost or because they are available only during that period of time.

Besides guaranteeing the plan correctness (i.e. that the correct actions in the correct order have been chosen), it is necessary to guarantee that it is executable (satisfying all the problem constraints and resource availability) and its optimality (optimizing due time, costs, etc). The latter problems can be addressed with scheduling and historically, the first two approaches to a unified planning and scheduling problem were the sequential approach of planning and scheduling and the temporal planning approach [10].

The sequential approach simply divides the solving process into two stages. First, the planner finds activities needed to complete the objective and then

the scheduler makes sure that all problem constraints are satisfied. The main problems of this approach are lack of global optimization criterion (the two processes work separately and do not cooperate) and overload on planning and scheduling process (the planner may produce plans that are not feasible, and thus several iterations of the whole process may be necessary until it is possible to create a schedule for the plan) [10].

Temporal planning approach means that a simple temporal reasoning module is included in the planner. Planners of this kind are called temporal planners and they are able to deal with temporal information and/or to reason on resources. This approach has two important drawbacks [10]:

- The temporal reasoning of the temporal planner is adequate but limited. The produced plans are executable, but may not be optimal or efficient
- It becomes difficult to determine when the system is planning or scheduling. The process is a mixture of planning and scheduling and therefore it is difficult to define common heuristic criteria to improve the performance of the system.

Integrated planning and scheduling is an approach that receives a lot of attention in research recently. It elevates many shortcomings of the previous two methods. In this approach, planner and scheduler cooperate on partial stages during the solving process. Unfeasible plans are quickly excluded from the search space, improving the overall performance of the whole system. Since planner and scheduler are still separate entities in this approach, traditional heuristic criteria can be applied to improve performance of the system.

3.2 Integrating planning with constraint-based scheduling

Focus of this work is on integration of planning with constraint-based scheduling. As we have shown in Section 2.2, the scheduling problem can be naturally modelled using CSP.

Recall that location of activity A in time is represented using variables S_A (start time), E_A (end time or completion time) and p_A (duration or processing time) and that these three variables are bound by constraint $S_A + p_A = E_A$.

One of the ways in which the scheduler can provide useful information to the planner is by removing unfeasible values from domains of variables used to represent the problem (domain pruning). When discussing the filtering

rules, we often need to refer to bounds of domain of a variable. Generally, if V is a variable, the lower bound of its domain is denoted as V^{min} and upper bound of its domain as V^{max}.

We shall often use the following values:

- S_A^{min} is minimum of domain of S_A, also called earliest start time of A
- S_A^{max} is maximum of domain of S_A, also called latest start time of A
- E_A^{min} is minimum of domain of S_A, also called earliest completion time of A
- E_A^{max} is maximum of domain of S_A, also called latest completion time of A

Traditional pruning methods in constraint-based scheduling

The effective domain pruning is one of the most important aspects of efficient constraint-solving procedures. We shall briefly introduce classical techniques used to prune domains of variables in constraint-based scheduling and we shall explain why they are not suitable for integrated planning and scheduling.

One classical pruning technique is called *timetabling* and is applicable to reservoir resources. It computes the minimal resource usage for every time instant t, creating an aggregated demand profile [13]. This profile is maintained during the search and is used to restrict the domains of the start and end times of activities by removing time instants that would necessarily lead to an over-consumption or over-production of the resource.

The main advantage of *timetabling* is its relative simplicity and its low algorithmic complexity. However, it propagates nothing until the time windows of activities become so small that some time instants are necessarily covered by some activity. This means that unless some strong commitments on the time windows of activities are made early in the search, this technique is not able to propagate efficiently.

Another pruning method, called *disjunctive constraint*, is usable only on unary (disjunctive) resources [9]. This algorithm analyzes each pair of activities A, B requiring the same unary resource. Whenever the current time bounds of activities are such that $S_A^{max} < E_B^{min}$, it deduces that, as activity A necessarily starts before the end of activity B, it must be completely executed before B. Thus the solver may add a new constraint $E_A \leq S_B$.

Edge-finding techniques [5, 12] are available for both unary and discrete resources. On a unary resource, edge-finding detects situations where a given

activity A cannot execute after any activity in a set Ω because there would not be enough time to execute all the activities in $\Omega \cup \{A\}$ between the earliest start time of activities in Ω and the latest end time of activities in $\Omega \cup \{A\}$. When such a situation occurs, it means that A must execute before all the activities in Ω and that implies a new upper bound for the end time of A. This instance of edge-finding technique is often referred to as *First* — meaning that activity A must be processed as the first element of set $\Omega \cup \{A\}$.

Similar rules allow us to detect and propagate the fact that a given activity must be processe after all activities in Ω (*Last*), cannot be processed before all activities in Ω (*Not First*) or cannot be processed after all activities in Ω (*Not Last*).

This technique can also be applied (in a generalized form) to discrete resources.

All of the pruning techniques described above (timetabling, disjunctive constraint, edge-finding) are very effective for pure scheduling problems. However, because they consider the absolute position of activities in time rather than their relative position, they will not propagate until the time windows of activities are small enough.

3.2.1 Energy precedence constraint

In [11] two new pruning techniques have been introduced: energy precedence constraint and balance constraint. Both of them exploit existing precedence relations between activities and both are able to prune even when time windows of activities are still very large. Thus they are able to support least-commitment strategies in planning and scheduling.

The *balance constraint* can be applied to a reservoir resource and it calculates lower and upper bounds of the resource level just before and just after some resource event. These bounds can afterwards be used to deduce new constraints or to detect inconsistencies. For a more detailed description of balance constraint, please refer to the original article.

We shall describe the *energy precedence constraint* more thoroughly, because our filtering rules introduced later use it in a slightly generalized form. Before we can proceed, we shall need a few definitions.

We can naturally extend the notion of earliest start time, latest start time, etc. to a set of activities Ω:

- $S_\Omega^{min} = min\{S_A^{min} | A \in \Omega\}$ is earliest start time of Ω
- $S_\Omega^{max} = max\{S_A^{max} | A \in \Omega\}$ is latest start time of Ω
- $E_\Omega^{min} = min\{E_A^{min} | A \in \Omega\}$ is earliest completion time of Ω

- $E_{\Omega}^{max} = max\{E_A^{max} | A \in \Omega\}$ is latest completion time of Ω

The energy precedence constraint is defined on discrete resources only and it is defined only when there are some precedence relations between scheduled activities. The precedence $A \ll B$ means that A must be processed completely before B. If A is an activity and $\Omega \subseteq \{X | X \ll A\}$ is a subset of activities that are constrained to execute before A, then the resource must provide enough energy to execute all activities in Ω between the earliest start time of Ω and S_A^{min}.

Suppose that Q denotes the maximal capacity of the discrete resource over time and that q_A is resource consumption of activity A. For simplicity we assume that both q_A and p_A are known constants (even though in a more general approach we could treat them as decision variables, see [11]). Then the energy precedence constraint for activity A and set Ω is formally defined as:

$$S_A^{min} \geq S_{\Omega}^{min} + \frac{\sum_{B \in \Omega}(q_B \cdot p_B)}{Q}$$

A symmetrical rule can be used to find the latest completion time E_A^{max} by considering the subsets that must execute after A ($\Omega \subseteq \{X | A \ll X\}$).

$$E_A^{max} \leq E_{\Omega}^{max} - \frac{\sum_{B \in \Omega}(q_B \cdot p_B)}{Q}$$

3.2.2 Filtering rules based on energy precedence constraint

In [2] authors propose a set of incremental propagation rules in which pruning of time windows is based on energy precedence constraint for unary resources. Since our work uses a very similar approach, the next paragraphs describe their approach in greater detail.

The paper addresses the problem of allocating n activities to a single disjunctive resource. It assumes that every activity has a corresponding time window in which it must be executed and that there are precedence constraints defined between the activities.

The activities are allowed to be *optional*, which means that it is not known in advance, whether the activities are allocated to the resource or not. Optional activity has one of three states. Either it is *valid*, in which case it must be included in the final schedule; or it is *invalid*, in which case it cannot be included in the final schedule; or it is *undecided*, in which case it is not yet decided whether it will be included in the final schedule or not.

Optional activities are useful for modeling alternative resources for the activities (an optional activity is used for each alternative resource and exactly one optional activity becomes valid) or for modeling alternative processes to accomplish a job (each process may consist of a different set of activities) [2].

To model optional activities, a decision variable $valid_A$ is introduced for every activity A. The value of this variable is 1 if the corresponding activity is valid, 0 if the activity is invalid and $\{0, 1\}$ if the activity is undecided.

The paper introduces a set of filtering rules, that may be separated into two groups: the rules updating the precedence graph and the rules pruning the time windows. The precedence graph is a structure representing precedence relations between activities. All filtering rules have access to this structure and may exploit the information it provides.

The first set of rules is responsible for keeping the precedence graph transitively closed, the second set of rules prunes time windows of activities using various methods, including the energy precedence constraint.

The paper considers activity allocation to a unary resource, where processing times of activities are known in advance. For this case the formulation of energy precedence constraint can be simplified. First, we may define a notion of processing time of a set of activities as

$$p_\Omega = \sum_{A \in \Omega} p_A$$

Since the resource capacity $Q = 1$ and consumption of activities $q_A = 1$, we may rewrite the energy precedence constraint for unary resource as

$$S_A^{min} \geq S_\Omega^{min} + p_\Omega$$

where $\Omega \subseteq \{X | X \ll A \wedge valid_X = 1\}$ is a subset of valid activities that must be processed before A. Note that we must take validity into consideration when optional activities are allowed.

During reasoning on optional activities, it is important to distinguish which activities may influence other activities. In general, undecided activities should not influence other (non-invalid) activities, but they can be influenced by others. Whenever the constraints concerning an undecided activity become inconsistent, the search is not ended with failure; the activity is declared invalid instead. Invalid activities and any constraints tied to them may be completely ignored, because they will not appear in the final schedule.

When using the energy precedence constraint to prune the time window of activity A, we must consider all subsets Ω of valid activities that must be

processed before A. The integrated constraint over all such subsets may be written as:

$$S_A^{min} \geq max\{S_\Omega^{min} + p_\Omega | \Omega \subseteq \{X | X \ll A \wedge valid_X = 1\}$$

Similar integrated constraint may be deduced for latest completion time:

$$E_A^{max} \leq min\{E_\Omega^{max} - p_\Omega | \Omega \subseteq \{X | A \ll X \wedge valid_X = 1\}$$

As shown in [2], the value of S_A^{min} based on energy precedence constraint can be calculated in time $O(n.log\ n)$ where n is number of activities using the procedure listed as Algorithm 3.1.

Algorithm 3.1 est(A) — calculates earliest start time of activity A based on energy precedence constraint

$dur \leftarrow 0$
$end \leftarrow$ inf
for all $Y \in \{X | X \ll A \wedge valid_X = 1\}$ in non-increasing order of S_Y^{min} **do**
  $dur \leftarrow dur + p_Y$
  $end \leftarrow max(end, S_Y^{min} + dur)$

The value of E_A^{max} can be calculated in a symmetrical way in time $O(n.log\ n)$.

Chapter 4

Basic concepts

4.1 The Problem

In this work we address the problem of allocating n activities to m unary resources. Every activity has an associated start time S_A, processing time p_A and resource on which it should be allocated res_A. We assume that activities are not interruptible — once they start, they occupy their resource until they are completed.

Recall that unary (or disjunctive) resource is a resource where activities cannot overlap in time. This means that for any two activities A, B that must be allocated to the same resource either A must be processed completely before B starts or vice versa. Formally, the following condition must hold:

$$(res_A = res_B) \Rightarrow (S_A + p_A \leq S_B \vee S_B + p_B \leq S_A)$$

We assume that every activity has a time window in which it must be executed. Time window $[R_A, D_A]$ specifies that activity A cannot start before R_A (release time) and must finish before D_A (deadline). Formally:

$$R_A \leq S_A \leq D_A - p_A$$

We allow simple temporal constraints between activities. Temporal constraint $A \xrightarrow{[L,U]} B$ means that activity B must start at least L and at most U time units after A has started. Formally:

$$L \leq S_B - S_A \leq U$$

Note that the time window is also a kind of temporal constraint — it restricts start time of an activity with regard to a fixed point in time. If we introduce a special "starting activity" $\overline{S}$ that represents this fixed point,

has processing time $p_{\overline{S}} = 0$ and start time $S_{\overline{S}} = 0$, then we can rewrite time window constraints as

$$R_A \leq S_A - S_{\overline{S}} \leq D_A - p_A$$

In other words, we can model time window $[R_A, D_A]$ as a temporal constraint between $\overline{S}$ and A: $\overline{S} \xrightarrow{[R_A, D_A - p_A]} A$.

Just like authors of [2], we also allow some activities to be optional. For optional activities it is not known in advance whether they will be allocated to their resource (i.e. included in the final schedule). Optional activities can be either *valid* (when they are known to be included in the schedule), *invalid* (when they are known not to be included in the schedule) or *undecided* (when it is not known if activity will be included in the schedule).

We are looking for such an assignment of start times to activities, that activities allocated to one resource do not overlap in time and that all temporal constraints are satisfied. Such an assignment is a solution of our problem.

When there are some undecided activities in the problem, it cannot be solved in the above sense. However, it is possible to provide a partial solution consisting of:

- A list of undecided activities that cannot be part of any solution (and thus must be treated as invalid).
- For every non-invalid activity a list of time points, in which it cannot start.

Such a partial solution can be used by a superior system (e.g. a planner) to make the decisions. The filtering rules that we propose are able to provide approximations of these partial solutions.

4.2 Temporal graph without optional nodes

In [2] and [11] authors used a structure called precedence graph to represent precedence relations between the activities. We use a similar structure (called temporal graph) to represent temporal relations between activities.

Activities $A_1, \ldots, A_n$ and temporal constraints between these activities define a simple temporal problem $T = (X, D, C)$, where

1. $X = (S_{A_1}, \ldots, S_{A_n})$
2. $D = \{D_{A_1}, \ldots, D_{A_n}\}$ and $D_{A_i} = [\inf, \sup]$ is domain for variable S_{A_i}
3. C is a set of temporal constraints between pairs of activities

Every solution of T provides an assignment of start times to activities $A_1, \ldots, A_n$, which respects the temporal constraints between the activities, but which may not respect the fact that activities are allocated to unary resources.

First we define a notion of temporal graph without optional variables. This structure can be used to represent a simple temporal problem induced by temporal constraints between a set of activities. Then we introduce a notion of simple temporal problem with optional variables and the temporal graph with optional nodes. These are used to represent temporal constraints between a set of optional activities.

Definition 4.1 (Temporal graph without optional nodes)**.** *Temporal graph without optional nodes* is a complete directed edge-weighted graph, where nodes represent start times of activities and weights of edges represent temporal constraints between them. If A_i and A_j are activities, then they are represented by nodes i and j in the temporal graph and weight of edge $i \to j$ is denoted as d_{ij}. Edges $i \to j$ and $j \to i$ represent temporal constraint $A_i \xrightarrow{[-d_{ji}, d_{ij}]} A_j$, i.e. an inequality:

$$-d_{ji} \leq S_{A_j} - S_{A_i} \leq d_{ij}$$

For every node i, edge $i \to i$ is also present in the graph.

When we speak about the "temporal graph" in this section, we always mean the "temporal graph without optional nodes".

Note that the semantics of edge weights is the same as in distance graph (see Definition 2.15). In fact, the temporal graph is always a distance graph of the STP it represents. When the constraints defined by the temporal graph are minimal, it is also a *d*-graph (see Definition 2.16) of the STP it represents. In such a case we say that the temporal graph is minimal.

Definition 4.2. Let G be a temporal graph. Let P_{ij} be path $i = i_0, i_1, \ldots, i_n = j$ from node i to node j. Length (or accumulated weight) of P_{ij} is defined as

$$d_{P_{ij}} = \sum_{k=1}^{n} d_{i_{k-1} i_k}$$

Definition 4.3. We say that edge $i \to j$ in the temporal graph is *minimal* if for every path P_{ij} from i to j

$$d_{ij} \leq d_{P_{ij}}$$

We say that the temporal graph is *minimal* if its every edge is minimal.

Intuitively, a temporal graph is minimal if every edge $i \to j$ is labeled with the length of the shortest path from i to j. It is easy to see that the temporal graph is d-graph of STP it represents if and only if it is minimal.

Theorem 4.1. *Temporal graph G is minimal, if and only if inequality*

$$d_{ij} \leq d_{ik} + d_{kj} \tag{4.1}$$

holds for every three nodes i, j, k.

Proof. If G is minimal, then the inequality (4.1) trivially holds for every i, j, k.

For the other implication, suppose that the inequality (4.1) holds for every i, j, k. We will show by induction on the number of nodes in path, that every edge $i \to j$ in G is minimal. Consider a path P_{ij}^3 from i to j, that contains three nodes i, j, k. The inequality

$$d_{ij} \leq d_{P_{ij}^3} = d_{ik} + d_{kj}$$

trivially holds (from inequality (4.1)). Now suppose (for induction) that for every path P_{ij}^{n-1} that contains $(n-1)$ nodes, the inequality

$$d_{ij} \leq d_{P_{ij}^{n-1}}$$

holds for every i, j. Let $P_{ij}^n = (i = i_1, i_2, \ldots, j = i_n)$ be a path from i to j that contains n nodes. Then

$$d_{ij} \leq d_{ii_2} + d_{i_2 j} \leq d_{ii_2} + d_{P_{i_2 j}^{n-1}} = d_{P_{ij}^n}$$

proves the induction step. The first inequality follows from (4.1) and the second inequality follows from induction hypothesis. The third inequality follows from the fact that the lenght of a path from i to j can be calculated as lentgh of edge $i \to i_2$ plus the length of a path from i_2 to j. □

We can represent different STPs (on the same set of activities) using the same temporal graph by adjusting weights of edges. When weight d_{ij} is decreased, then the temporal constraint between nodes i and j becomes more strict. When the weight is increased, the temporal constraint becomes more relaxed.

We shall use the temporal graph to represent temporal constraints between a set of activities during constraint-based scheduling. During the reasoning we shall deduce new temporal constraints (e.g. by considering relative positions of activities and resource restrictions). The search procedure looking for the final schedule will also introduce new constraints in order

reduce domains of all variables to singletons. Both of these changes make the temporal constraints more strict and thus are manifested as the decrease of weights of some edges in the temporal graph. Weights of edges in the temporal graph are never increased during the constraint reasoning. This observation becomes important in Section 6.1, where we create a constraint model for the temporal graph.

4.3 STP with optional variables

We allow activities to be optional — every activity is either valid, invalid or undecided. Valid activities are always included in the final schedule, invalid activities are always excluded from it and undecided activities may end up either way. We introduce a notion of simple temporal problem with optional variables, which will be a formal basis for our reasoning on the temporal constraints between a set of optional activities.

Definition 4.4. Simple temporal problem with optional variables (STPOV) is pair (S, M) where S is a simple temporal problem (without optional variables) and M is a mapping that labels every variable of S as valid, invalid or undecided.

The semantics of these labels is motivated by our intention to model optional activities. In particular, the valid variables represent valid activities, which are always included in the final schedule and so the variables must respect all relevant constraints. The invalid variables represent invalid activities, which do not appear in the final schedule. Thus, the invalid variables should be treated as nonexistent: they should not affect the other variables in any way and do not have to obey any constraints. The undecided variables represent undecided activities and these may or may not appear in the final schedule. Thus undecided variables may either be treated as nonexistent, or they may be required to respect all associated constraints.

Definition 4.5. Let T be an STPOV. We denote the set of its valid variables as $valid(T)$, the set of its invalid variables as $invalid(T)$ and the set of its undecided variables as $undecided(T)$.

The semantics of undecided variables disallows definition of a solution for general STPOV. For this reason we distinguish two types of STPOV: those that contain some undecided variables and those that do not. For the latter type (called ground STPOV), it is possible to define a notion of solution that is compatible with the notion of solution of a standard STP.

Definition 4.6. We say that an STPOV is *ground* if none of its variables are labeled as undecided.

Definition 4.7. Let $T = ((X, D, C), M)$ be an STPOV and let $Y = \{Y_1, \ldots, Y_k\}$ be a set of variables such that $Y \subseteq X$. We say that $C' = \{T_{ij} | X_i \in Y \wedge X_j \in Y\}$ is a *restriction* of C to Y, denoted $C|Y$.

Definition 4.8 (Solution)**.** Let $T = ((X, D, C), M)$ be a ground STPOV, where $X = \{X_1, \ldots, X_n\}$. We say that tuple $(x_1, \ldots, x_n)$ is a *solution* of T if the assignment

$$\{X_1 = x_1, \ldots, X_n = x_n\}$$

satisfies all of the temporal constraints in $C|valid(T)$.

A solution of a ground STPOV assigns a value to every one of its variables. The valid variables must satisfy only the temporal constraints that involve other valid variables. The invalid variables may be assigned arbitrary values, but since we are not interested in their values, we require (without loss of generality) that the invalid variables are consistently assigned a value of 0.

Note that the valid variables and the temporal constraints between them form a simple temporal problem in which we have no reason to consider optional variables. Thus we can associate every ground STPOV with an STP.

Definition 4.9 (Associated STP)**.** Let $T = ((X, D, C), M)$ be a ground STPOV. We say that STP $T' = (X', D', C')$ is an *associated STP* of T if:

- $X' = valid(T)$
- D' is a set of corresponding domains for variables in X'
- $C' = C|valid(T)$ is a set of constraints from T that involve only variables in X'

If T is a ground STPOV and T' is its associated STP, then it is clear, that there is a one-to-one mapping between solutions of T and T'.

Lemma 4.2. *Let T be a ground STPOV and T' its associated STP. Then from every solution of T we can derive a solution of T' and every solution of T' can be extended to a solution of T. The mapping between solutions of T and T' is bijective.*

Proof. Obvious. □

The Lemma 4.2 allows us to solve ground STPOV with algorithms designed to solve STP.

Definition 4.10 (Consistency)**.** We say that a ground STPOV is *consistent* if it has at least one solution.

Corollary 4.3. *A ground STPOV is consistent if and only if its associated STP is consistent.*

Proof. Trivial consequence of Lemma 4.2 and Definition 2.6. □

We do not define the notion of a solution of a non-ground STPOV. Instead, we consider every non-ground STPOV to be a "generalization" of a set of ground STPOV.

Definition 4.11. Let $T = (S, M)$ be an STPOV. Let M' be a mapping, such that it labels every variable of S as valid or invalid, and that

$$(M(X_i) \neq \mathit{undecided}) \Rightarrow (M(X_i) = M'(X_i))$$

holds for every i. Then STPOV $T' = (S, M')$ is called an *instance* of T.

Thus, T' is instance of T if T' is ground and if they differ only in labels of variables in $\mathit{undecided}(T)$.

Definition 4.12. Let T be an STPOV. We denote the set of all instances of T as $\mathit{inst}(T) = \{T' | T' \text{ is instance of } T\}$

Definition 4.13. Let T and T' be a pair of ground STPOV on the same set of variables. We say that T and T' are *equivalent* if they have the same set of solutions, i.e. if the following equivalence holds:

$$x \text{ is a solution of } T \Leftrightarrow x \text{ is a solution of } T'$$

Definition 4.14. Let T and T' be a pair of non-ground STPOV on the same set of variables. We say that T and T' are equivalent if for every $U \in \mathit{inst}(T)$ such that $U = (S, M)$ and for every $U' \in \mathit{inst}(T')$ such that $U' = (S', M')$ the following implication holds:

$$(M = M') \Rightarrow (x \text{ is a solution of } U \Leftrightarrow x \text{ is a solution of } U')$$

We may define consistency of a non-ground STPOV through its instances.

Definition 4.15. We say that a non-ground STPOV is *consistent* if at least one $T' \in \mathit{inst}(T)$ is consistent.

Thus, we consider a non-ground STPOV consistent, if there exists such an instantiation of its undecided variables, that it produces a consistent ground STPOV.

It is possible to extend the definition of constraint graph and distance graph (see Section 2.3) to include optional nodes. These graphs can represent STPOV, just like the constraint graph and the distance graph without optional nodes represent a classical STP.

Definition 4.16. Let G be a graph with optional nodes. We say that cycle $i_1, \ldots, i_k = i_1$ is *unbreakable*, if all of its nodes are valid.

Theorem 4.4. *An STPOV is consistent if and only if its distance graph contains no unbreakable negative cycles.*

Proof. Let us first consider a case when T is a ground STPOV. Let T' be an STP associated with T. The following statements are equivalent:

1. The distance graph of T contains no unbreakable negative cycles.
2. The distance graph of T' contains no negative cycles.
3. T' is consistent.
4. T is consistent.

Equivalence (1) $\Leftrightarrow$ (2) is trivial consequence of Definition 4.9. Equivalence (2) $\Leftrightarrow$ (3) follows from Theorem 2.1. Equivalence (3) $\Leftrightarrow$ (4) follows from Corollary 4.3. Equivalence (1) $\Leftrightarrow$ (4) is what we need to prove.

Now let us consider a case when T is non-ground. A non-ground STPOV is consistent iff there is some consistent $S \in inst(T)$. If there is no unbreakable negative cycle in T, then $S \in inst(T)$ such that all undecided variables of T are invalid in S also does not contain unbreakable negative cycle and thus S is consistent. Conversely, if there is a ground STPOV $S \in inst(T)$ such that it is consistent, then it cannot contain unbreakable negative cycle and thus T cannot contain unbreakable negative cycle either, because it does not have more valid variables than S. □

If an STPOV is inconsistent, all $S \in inst(T)$ are also inconsistent. Theorem 4.4 allows us to detect inconsistency for all $S \in inst(T)$ by looking for unbreakable negative cycles in the distance graph of T. It also allows us to detect undecided variables that must be invalid in every consistent instance of T.

Corollary 4.5. *Let T be an STPOV, let T' be an STPOV that has the same variables and constraints as T. Suppose that all variables except X_i are labeled in the same manner in both problems and that variable X_i is undecided in T and invalid in T'. If there is a negative cycle $i = i_0, i_1, \ldots, i_k = i$ in distance the graph of T such that $i_1, \ldots, i_{k-1}$ are all valid, then T and T' are equivalent.*

Proof. We have to show T and T' have the same set of consistent instances and that the corresponding consistent instances have the same sets of solutions.

To show that T and T' have the same set of consistent instance it is sufficient to show that the instances of T in which X_i is valid are all inconsistent. However, if X_i is valid then the cycle $i = i_0, i_1, \ldots, i_k = i$ is unbreakable and consequently (by Theorem 4.4) the instances in which X_i is valid are inconsistent.

Since the temporal constraints are the same in all instances of both T and T', the solution sets of corresponding consistent instances must be the same. □

Corollary 4.5 allows us to detect variables that cannot be valid in any consistent instance of an STPOV. When we find such a variable we may declare it invalid, thus acquiring an STPOV which is more specific (contains less undecided variables) without losing any solutions.

Let T be an STPOV and let G be its distance graph. Edge $i \to j$ in G has weight c_{ij} and represents inequality $X_j - X_i \leq c_{ij}$. If both i and j are valid then this inequality must hold in all $S \in inst(T)$. If either i or j is invalid, then the inequality is not required to hold in any $S \in inst(T)$. If either i or j is undecided, then the inequality must hold in those instances, where X_i and X_j are both valid.

Now consider a path from i to j in G, $P_{ij} = (i_0, i_1, \ldots, i_k)$, where $i_0 = i$ and $i_k = j$. Edges on the path represent a set of inequalities

$$X_{i_1} - X_{i_0} \leq c_{i_0 i_1}$$
$$X_{i_2} - X_{i_1} \leq c_{i_1 i_2}$$
$$\ldots$$
$$X_{i_k} - X_{i_{k-1}} \leq c_{i_{k-1} i_k}$$

If all nodes in path P_{ij} are valid, then these inequalities must hold in all instances of T. In such a case we may conclude that the sum of these inequalities yields a constraint

$$X_{i_k} - X_{i_0} \leq \sum_{j=1}^{k} c_{i_{j-1} i_j}$$

which must also hold in all instances of T. Thus, the path P_{ij} consisting only of valid nodes induces a new constraint on distance $X_j - X_i$

$$X_j - X_i \leq \sum_{j=1}^{k} c_{i_{j-1}i_j} \tag{4.2}$$

Let us now consider a situation when X_i is undecided and all other nodes in P_{ij} are valid. The constraint (4.2) holds in all instances of T where X_i is valid, because in these instances all nodes of P_{ij} are valid. Similarly, the constraint (4.2) is ignored in all instances of T, where X_i is invalid, just like all the other constraints that involve X_i. The similar reasoning can be applied when X_j is undecided or when both X_i and X_j are undecided.

This means that even if variables X_i and X_j are undecided, the path-induced constraint (4.2) holds in all instances of T where both X_i and X_j are valid.

Based on these observations, we define a notion of a valid path. Valid paths are those that induce "useful" temporal constraints in an STPOV.

Definition 4.17. Let P_{ij} be a path from i to j. We say that nodes i and j are *outer* nodes of P_{ij}, all the other nodes are its *inner* nodes.

Definition 4.18. Let G be a graph with optional nodes. Path P_{ij} from i to j in G is called *valid* if all of its inner nodes are valid.

Note that we consider every edge to be a valid path.

We say that edge $i \to j$ is *non-invalid* if neither i nor j are invalid.

Every valid path $P_{ij} = (i_0, i_1, \ldots, i_k)$ induces the following constraint on distance $X_j - X_i$:

$$X_j - X_i \leq \sum_{j=1}^{k} c_{i_{j-1}i_j}$$

The intersection of induced constraints over all valid paths from i to j is

$$X_j - X_i \leq d_{ij}$$

where d_{ij} is the length of the shortest valid path from i to j.

Definition 4.19. Let T be an STPOV and let G be its distance graph. A complete graph on the same set of nodes as G, where every non-invalid edge $i \to j$ is labeled with d_{ij} — the length of the shortest valid path from i to j in G — is called d-graph of T.

Note that in the definition of d-graph of an STPOV, we require only non-invalid edges to be labeled with the length of the shortest path.

4.4 Temporal graph with optional nodes

Now we may extend the notion of temporal graph to include optional nodes.

Definition 4.20 (Temporal graph with optional nodes)**.** *Temporal graph with optional nodes* is the same graph as temporal graph without optional nodes (see Definition 4.1), but every node is additionally labeled as valid, invalid or undecided (always with regard to validity of activity it represents).

The temporal graph with optional nodes can be used to represent STPOV. We shall represent a sequence of various STPOV on the same set of activities by adjusting the weights of edges and by changing validity of nodes.

Typically, we decrease weights of edges in temporal graph (thus making the temporal constraints more strict) and change undecided nodes either to valid or invalid (thus making the STPOV more specific). These types of changes are consistent with typical approach in constraint-based reasoning.

Definition 4.21. We say that edge $i \to j$ in temporal graph with optional nodes is *minimal* if for every valid path P_{ij} from i to j

$$d_{ij} \leq d_{P_{ij}}$$

We say that temporal graph with optional nodes is minimal if its every non-invalid edge is minimal.

The temporal graph is minimal iff it is the d-graph of the STP it represents.

Theorem 4.6. *Temporal graph G is minimal, if and only if inequality*

$$d_{ij} \leq d_{ik} + d_{kj} \tag{4.3}$$

holds for every three nodes i, j, k such that i and j are not invalid and k is valid.

Proof. The proof of Theorem 4.1 can be easily extended to consider optional nodes and slightly different definition of minimality of the temporal graph with optional nodes. □

Chapter 5

Algorithms for temporal graph

5.1 Incremental minimality algorithms

We use the temporal graph without optional nodes to represent simple temporal constraints between a set of activities. Our aim is to always keep the temporal graph minimal (see Definition 4.21), since minimal constraints significantly reduce the search space and when there are no optional nodes, they even guarantee backtrack-free generation of solutions (see Theorem 2.2).

Since the temporal graph is also a distance graph of the STP it represents, we can use Floyd-Warshall's algorithm (Algorithm 2.3) to find the shortest paths in the temporal graph. However, we typically deal with a situation where the temporal graph is minimal and we decrease weight of only one edge (or only a few edges). The Floyd-Warshall's algorithm recalculates the shortest paths between all nodes in the temporal graph from scratch. This is hardly necessary since major part of the graph typically needs no update.

For this reason, we introduce an algorithm that recalculates the shortest paths in the temporal graph more efficiently. First, we introduce a version of this algorithm that works on a temporal graph without optional nodes. Then we introduce an enhanced version that works on a temporal graph with optional nodes. The enhanced version of the algorithm serves as a basis for one of the filtering rules proposed in later chapters.

5.1.1 Temporal graph without optional nodes

In this section we do not consider optional nodes. Wherever we write "temporal graph", we mean "temporal graph without optional nodes".

When we change the weights of edges in the temporal graph, we change the temporal constraints of the underlying STP. The algorithms we present in

this work always decrease weights in the temporal graph (and never increase them) and thus make the temporal constraints in the temporal graph more strict.

We distinguish two types of assignments. An *equivalence-preserving* assignment is such, that the temporal problems represented by the temporal graph before and after the assignment are equivalent. If the assignment is not equivalence-preserving, we say that it is *equivalence-breaking*.

We use equivalence-preserving assignments whenever it is necessary to preserve the set of solutions of the underlying temporal problem and we use equivalence-breaking assignments whenever it is necessary to introduce new temporal constraints into the temporal graph.

For example, if the temporal graph is minimal, then any assignment decreasing the weight of an edge must be equivalence-breaking. If the temporal graph is not minimal, then the assignment that decreases the weight of an edge to the length of the shortest path between its nodes is equivalence-preserving.

Let i and j be the nodes in the temporal graph that represent activities A and B. The edges $i \to j$ and $j \to i$ and their weights, d_{ij} and d_{ji}, represent constraint $-d_{ji} \leq S_B - S_A \leq d_{ij}$. Suppose that we deduce (for example by analyzing the relative positions of activities and resource constraints), that the start times of the activities must also satisfy constraint $L \leq S_B - S_A \leq U$. The constraint $max(-d_{ji}, L) \leq S_B - S_A \leq min(d_{ij}, U)$ is an intersection of the original and the deduced constraint. The following assignments update edges $i \to j$ and $j \to i$ to represent the new intersected constraint:

$$d_{ij} \leftarrow min(d_{ij}, U)$$

$$d_{ji} \leftarrow min(d_{ji}, -L)$$

The procedure listed as Algorithm 5.1 assumes, that it works with a minimal temporal graph. It decreases the weight of edge $i \to j$ to a new value and updates the weights of other edges as necessary to make the temporal graph minimal again.

We will show that Algorithm 5.1 is correct, always finishes and detects negative cycles.

Definition 5.1. Let G be a temporal graph. If G contains no negative cycles after the weight of edge $i \to j$ is decreased, we say that the weight has been *consistently decreased*.

Proposition 5.1. *If Algorithm 5.1 consistently decreases value of d_{ij} in a minimal temporal graph G, then G is minimal after the algorithm finishes. All assignments inside the while-loop are equivalence-preserving.*

Algorithm 5.1 Decrease weight of $i \to j$ to w

```
1:  Q ← ∅
2:  if d_ij > w then
3:      d_ij ← w
4:      enqueue(Q, i → j)
5:  while Q is not empty do
6:      (i → j) ← dequeue(Q)
7:      if i = j ∧ d_ii < 0 then
8:          return failure
9:      for k := 1 to n do
10:         if d_ik > d_ij + d_jk then
11:             d_ik ← d_ij + d_jk
12:             enqueue(Q, i → k)
13:         if d_kj > d_ki + d_ij then
14:             d_kj ← d_ki + d_ij
15:             enqueue(Q, k → j)
```

Proof. Temporal graph G was minimal before the algorithm begins and thus all the triangles (subgraphs on three nodes) of G are minimal (by Theorem 4.1). After weight d_{ij} is decreased by the assignment $d_{ij} \leftarrow w$ on line 3, some triangles that contain edge $i \to j$ may no longer be minimal. Particularly, the inequalities $d_{ik} \leq d_{ij} + d_{jk}$ and $d_{kj} \leq d_{ki} + d_{ij}$ may no longer hold for some k. The algorithm checks whether these inequalities hold for every k (lines 10 and 13) and if they do not, then the proper weights are decreased accordingly by the assignments on lines 11 and 14. When the for-loop ends, every triangle i, j, k is minimal.

Every assignment that decreases weight of an edge in G may cause some triangles that contain the edge to become non-minimal. Thus, by restoring minimality in one triangle, the algorithm may break the minimality in up to n other triangles. This problem is solved by adding all the edges whose weight was decreased into Q. In subsequent iterations of the while-loop, these edges are removed from Q and processed in the same manner as edge $i \to j$ (i.e. minimality of every triangle that contains the edge is restored).

If Q is empty, the algorithm ends. We will show that if this happens, all the triangles in G are minimal (and consequently G is minimal, see Theorem 4.1). Suppose for contradiction that the algorithm has ended and there is a triangle that is not minimal, say a, b, c. Without loss of generality

$$d_{ac} > d_{ab} + d_{bc}$$

is the inequality that breaks the minimality. The triangle a, b, c was minimal

before the algorithm began, and so either d_{ab} or d_{bc} had to be decreased by the algorithm. Consider the last assignment that decreased either d_{ab} or d_{bc}. Without loss of generality, suppose that this last assignment decreased d_{ab}. After the assignment, edge $a \to b$ is inserted into Q. Since the algorithm has ended, the queue Q must be empty and so some iteration of the while-loop must have processed the edge $a \to b$. During this processing, the triangle a, b, c for every c is made minimal and so $d_{ac} \leq d_{ab} + d_{bc}$. The weights d_{ab} and d_{bc} cannot be decreased by any later iterations of the while-loop (because we considered the last assignment). Thus the inequality $d_{ac} \leq d_{ab} + d_{bc}$ holds after the algorithm finishes, which is a contradiction with the assumption of non-minimality.

What remains to prove is that the assignments that restore minimality are equivalence-preserving. Without loss of generality, consider the assignment

$$d_{ik} \leftarrow d_{ij} + d_{jk}$$

The weight d_{ik} represents constraint $S_{A_j} - S_{A_i} \leq d_{ik}$. If an assignment decreases value of d_{ik}, the set of feasible values for distance $S_{A_j} - S_{A_i}$ can be reduced, which would mean that the assignment is equivalence-breaking. We will show that in this case it is not so.

Consider the path $i \to j$, $j \to k$. Weights of edges on this path represent the following constraints:

$$S_{A_j} - S_{A_i} \leq d_{ij}$$
$$S_{A_k} - S_{A_j} \leq d_{jk}$$

The sum of these inequalities yields:

$$S_{A_k} - S_{A_i} \leq d_{ij} + d_{jk}$$

Thus, decreasing value of d_{ik} to $d_{ij} + d_{jk}$ cannot reduce the set of feasible values for distance $S_{A_k} - S_{A_i}$ and the assignment is equivalence-preserving. □

Lemma 5.2. *Let i, j and k be any three nodes in a temporal graph that contains no negative cycles. Then:*

1. *There is always at least one shortest path from j to k that does not go through edge $i \to j$.*
2. *There is always at least one shortest path from k to i that does not go through edge $i \to j$.*

Proof. We prove only the first part, the proof for the second part is very similar.

Every path from j to k going through edge $i \to j$ can be expressed as

$$j \to \ldots \to i, i \to j, j \to \ldots \to k$$

where the last section of the path $j \to \ldots \to k$ does not go through $i \to j$. Let us exclude cycle $j \to \ldots \to i, i \to j$. What remains is a path from j to k that is not longer than the original path (because the graph does not contain negative cycles) and does not go through $i \to j$. □

Corollary 5.3. *If the weight of edge $i \to j$ (d_{ij}) in the minimal temporal graph is consistently decreased, then:*

1. *Edge $j \to k$ for any k remains minimal even after weight d_{ij} is decreased.*
2. *Edge $k \to i$ for any k remains minimal even after weight d_{ij} is decreased.*

Proof. We prove only the first part, the proof for the second part is very similar.

Edge $j \to k$ is minimal before d_{ij} is decreased. The length of the shortest path from j to k changes after d_{ij} is decreased only if every shortest path from j to k goes through $i \to j$. However, that contradicts Lemma 5.2 and thus edge $j \to k$ remains minimal. □

We have shown that the edges of the minimal temporal graph beginning in j or ending in i remain minimal even after the weight of edge $i \to j$ is decreased.

Proposition 5.4. *If Algorithm 5.1 consistently decreases value of d_{ij} in a minimal temporal graph G, the algorithm finishes in at most n^2 iterations of the while-loop.*

Proof. After lines 2-4 are executed, the value of d_{ij} is decreased to a new value and the edge $i \to j$ is the only element of queue Q. Let K be a set of edges in G that are not minimal, $K = \{k \to l | k \to l \text{ is not minimal}\}$. For every $k \to l$ from K, the shortest path from k to l must go through $i \to j$, because the edges were minimal before the assignment on line 3.

We know that the edges starting in j and the edges ending in i cannot be in K (by Corollary 5.3).

We will show that for every edge $k \to l$ in K, the length of the shortest path from k to l is $d_{ki} + d_{ij} + d_{jl}$. We know that the shortest path from k to

j goes through $i \to j$. The shortest path from k to i has length d_{ki}, because $k \to i$ is minimal (it is not in K). Similarly, the shortest path from j to l has length d_{jl}, because $j \to l$ is also minimal. The shortest path from i to j is obviously d_{ij}. Thus, the sum $d_{ki} + d_{ij} + d_{jl}$ is the length of the shortest path from k to l.

For the specific case of an edge $i \to l$ in K, the length of the shortest path from i to l is $d_{ij} + d_{jl}$ and for the specific case of an edge $k \to j$ in K, the length of the shortest path from k to j is $d_{ki} + d_{ij}$.

We will show that every non-minimal edge is removed from K by exactly one assignment (i.e. that the edge is rendered minimal by exactly one assignment).

First we will show that all edges starting in i and all edges ending in j are removed from K in the first iteration of the while-loop.

The first iteration of the while-loop removes $i \to j$ from Q. Then it checks for every node k, whether the edge $i \to k$ belongs to K (i.e. whether it is non-minimal).

If $i \to k$ is non-minimal, the assignment $d_{ik} \leftarrow d_{ij} + d_{jk}$ makes it minimal (because $d_{ij} + d_{jk}$ is the length of the shortest path from i to k). After this assignment, the edge $i \to k$ is no longer in K.

The first iteration of the while-loop also checks for every node k, whether the edge $k \to j$ belongs to K and eventually makes in minimal.

We have shown that all edges starting in i and all edges ending in j are removed from K in the first iteration of the while-loop. Thus, after the first iteration of the while-loop, K contains no edges that start or end in i or j. All of the edges that were removed from K are now in the queue Q.

If Q is not empty, then every subsequent iteration removes one edge from Q and processes it just like the first iteration processed the edge $i \to j$. In these subsequent iterations, the edges that neither start nor end in i or j can have their weight decreased by an assignment. Suppose that $k \to l$ is such an edge. Then the assignment that changes its weight can have two forms:

$$d_{kl} \leftarrow d_{ki} + d_{il}$$

$$d_{kl} \leftarrow d_{kj} + d_{jl}$$

The former case occurs, when d_{kl} is changed during processing of edge $i \to l$, the latter case occurs when processing edge $k \to j$. However, since $d_{il} = d_{ij} + d_{jl}$ and $d_{kj} = d_{ki} + d_{ij}$ (these are values assigned in the first iteration of the while-loop), both cases can be written as

$$d_{kl} \leftarrow d_{ki} + d_{ij} + d_{jl}$$

As we have shown before, the sum $d_{ki} + d_{ij} + d_{jl}$ is the length of the shortest path from k to l. Thus, the edge $k \to l$ is minimal after any of the above two assignments.

We have proven that every edge is removed from K by one assignment. Thus every edge of G enters queue Q at most once and thus the while-loop has at most n^2 iterations. Moreover, since $2n$ edges can never appear in K, the while-loop is executed at most $n^2 - 2n$ times. □

The Algorithm 5.1 is designed to decrease weight of one edge and propagate this change into the rest of the temporal graph. However, its practical use often requires that several weights are decreased and then all of these changes are propagated into the other edges. The proof of correctness can be easily extended for this modification, but unfortunately the proof of time complexity no longer holds in this case.

Until now we have assumed that the Algorithm 5.1 is not forced to introduce a negative cycle into G. The next proposition describes behavior of the algorithm when a negative cycle is introduced into the temporal graph.

Proposition 5.5. *If Algorithm 5.1 introduces a negative cycle into a minimal temporal graph by decreasing the weight of edge $i \to j$, then the algorithm finishes with failure after a finite number of while-loop iterations.*

Proof. Let $i, j, k_1, \ldots, k_n, i$ be a negative cycle in temporal graph after d_{ij} is decreased, i.e.

$$d_{ij} + d_{jk_1} + \ldots + d_{k_n i} < 0$$

Then the algorithm must perform (at least) the following sequence of assignments:

$$d_{ik_1} \leftarrow d_{ij} + d_{jk_1}$$
$$d_{ik_2} \leftarrow d_{ik_1} + d_{k_1k_2}$$
$$\ldots$$
$$d_{ik_n} \leftarrow d_{ik_{n-1}} + d_{k_{n-1}k_n}$$
$$d_{ii} \leftarrow d_{ik_n} + d_{k_n i}$$

The first assignment must be performed because $d_{ik_1} > d_{ij} + d_{jk_1}$. Indeed, if this was not true, then $d_{ik_1} \leq d_{ij} + d_{jk_1}$ and

$$d_{ik_1} + d_{k_1k_2} + \ldots + d_{k_n i} \leq d_{ij} + d_{jk_1} + \ldots + d_{k_n i} < 0$$

But that would mean that there was a negative cycle in temporal graph even before d_{ij} was decreased, which is a contradiction.

The second assignment must be performed because $d_{ik_2} > d_{ik_1} + d_{k_1k_2}$ (using similar reasoning as above) and d_{ik_1} is decreased by the first assignment.

All other assignments must be performed for similar reasons, the last one assigns the length of the negative cycle to d_{ii}. This is detected when the edge $i \to i$ is removed from Q and the algorithm ends with failure. □

5.1.2 Temporal graph with optional nodes

In this section we present a modification of Algorithm 5.1 that works with temporal graph with optional nodes. When we write about the "temporal graph" in this section, we mean the "temporal graph with optional nodes".

Algorithm 5.1 introduces a new temporal constraint into minimal temporal graph (without optional variables) and makes the graph minimal again. We present a modification of this algorithm that introduces a new temporal constraint into temporal graph with optional variables. It is listed as Algorithm 5.2.

Algorithm 5.2 Decrease weight of $i \to j$ to w

```
Q ← ∅
if d_ij > w then
  d_ij ← w
  enqueue(Q, i → j)
while Q is not empty do
  (i → j) ← dequeue(Q)
  if i = j ∧ d_ii < 0 then
    if i is valid then
      return failure
    else
      make i invalid
  if j is valid then
    for k := 1 to n do
      if k is not invalid ∧ d_ik > d_ij + d_jk then
        d_ik ← d_ij + d_jk
        enqueue(Q, i → k)
  if i is valid then
    for k := 1 to n do
      if k is not invalid ∧ d_kj > d_ki + d_ij then
        d_kj ← d_ki + d_ij
        enqueue(Q, k → j)
```

We will show that Algorithm 5.2 decreases the value of d_{ij} to requested

value, always finishes and makes the temporal graph minimal (unless decreasing the value of d_{ij} introduces an unbreakable negative cycle into the temporal graph, in which case the algorithm ends with failure).

Definition 5.2. Let G be a temporal graph. If G contains no unbreakable negative cycles after the weight of edge $i \to j$ is decreased, we say that the weight has been *consistently decreased*.

Proposition 5.6. *If Algorithm 5.2 consistently decreases value of d_{ij} in a minimal temporal graph G, then G is minimal after the algorithm finishes. All assignments inside the while-loop are equivalence-preserving.*

Proof. The proof is a variation of proof of Proposition 5.1. The steps of the proof are the same, it is sufficient to apply dual definitions and theorems for optional variables. □

Lemma 5.7. *Let i, j and k be any three nodes in a temporal graph that contains no unbreakable negative cycles.*

1. *There is always at least one shortest valid path from j to k that does not go through edge $i \to j$.*
2. *There is always at least one shortest valid path from k to i that does not go through edge $i \to j$.*

Proof. We prove only the first part, the proof for the second part is very similar.

Every valid path from j to k going through edge $i \to j$ can be expressed as:

$$j \to \ldots \to i, i \to j, j \to \ldots \to k$$

where the last section of the path $j \to \ldots \to k$ does not go through $i \to j$. Let us exclude cycle $j \to \ldots \to i, i \to j$. Since the path is valid, all of its inner nodes must be valid. Thus the excluded cycle is unbreakable and what remains is a path from j to k that is surely not longer than the original path (because the cycle cannot be negative) and does not go through $i \to j$. □

Corollary 5.8. *If the weight of edge $i \to j$ (d_{ij}) in the minimal temporal graph is consistently decreased, then:*

1. *Every non-invalid edge $j \to k$ remains minimal even after weight d_{ij} is decreased.*
2. *Every non-invalid edge $k \to i$ remains minimal even after weight d_{ij} is decreased.*

Proof. We prove only the first part, the proof for the second part is very similar.

Edge $j \to k$ is minimal before d_{ij} is decreased. The length of the shortest valid path from j to k would change after d_{ij} is decreased only if every shortest valid path from j to k goes through $i \to j$. However, that contradicts Lemma 5.7 and thus edge $j \to k$ remains minimal. □

We have shown that non-invalid edges of minimal temporal graph beginning in j or ending in i remain minimal even after weight of edge $i \to j$ is decreased. As before, we use this argument to prove time complexity of Algorithm 5.2 in the following proposition.

Proposition 5.9. *If Algorithm 5.2 consistently decreases value of d_{ij} in a minimal temporal graph G, the algorithm finishes in at most n^2 iterations of the while-loop.*

Proof. The proof is almost the same as for Proposition 5.4. The steps of the proof are the same, it is sufficient to apply dual definitions and theorems for optional variables. □

Proposition 5.10. *If Algorithm 5.2 introduces an unbreakable negative cycle into a minimal temporal graph by decreasing weight of edge $i \to j$, then the algorithm finishes with failure after finite number of while-loop iterations.*

Proof. The proof is very similar to proof of Proposition 5.5. □

Proposition 5.11. *If Algorithm 5.2 introduces a negative cycle, in which all nodes except one are valid and the remaining node is undecided, into a minimal temporal graph by decreasing weight of edge $i \to j$, then the algorithm detects this cycle and breaks potential unbreakable negative cycle by declaring the undecided node invalid.*

Proof. Let k be the undecided node in the negative cycle. The path from k to k along the negative cycle is a valid path. Thus the length of the shortest path from k to k is negative and the algorithm will eventually assign $d_{kk} \leftarrow c$, where c is the length of the negative cycle. d_{kk} becomes negative after this assignment and this is eventually detected by the algorithm. Since k is undecided, the algorithm declares it invalid. □

Just like we needed a mechanism to introduce new temporal constraints into the temporal graph, we need a mechanism that will allow superior system to declare undecided variables either valid or invalid. When a previously undecided variable is declared invalid, then it is not necessary to make any additional changes in the temporal graph. When a variable becomes valid,

then new valid paths may appear in the temporal graph and the temporal graph may no longer be minimal.

We propose an algorithm that makes the selected node in the temporal graph valid and ensures that the graph remains minimal.

Algorithm 5.3 Consistent validation of node k

```
make k valid
for i, j := 1 to n; i, j ≠ k do
  if i → j is non-invalid then
    d_ij ← min(d_ij, d_ik + d_kj)
    if i = j ∧ d_ii < 0 then
      if i is valid then
        return failure
      else
        make i invalid
```

Proposition 5.12. *Let G be a minimal temporal graph. Algorithm 5.3 makes node k in G valid and after the algorithm finishes, if validation of k did not introduce unbreakable negative cycle into G, G is minimal again. Otherwise the algorithm reports failure.*

Proof. After node k becomes valid, new valid paths are introduced into G. Assume that non-invalid edge $i \to j$ is no longer minimal. Since it was minimal before the algorithm made k valid, the shortest path from i to j must now go through k. The length of the shortest path from i to j is $d_{ik} + d_{kj}$, because $i \to k$ and $k \to j$ must be minimal, as we will show later. The algorithm checks every non-invalid edge in G and decreases its weight accordingly to make it minimal. Thus G must be minimal after the algorithm finishes.

What remains to show is that all edges starting or ending in k remain minimal even after k is made valid. First, consider edge $i \to k$ for some i. If it is not minimal, then all shortest paths from i to k now go through k (because $i \to j$ was minimal before k was valid) and $d_{ik} > d_{P_{ik}}$, where P_{ik} is some shortest path from i to k going through k. Length of path P_{ij} can be expressed as

$$d_{P_{ik}} = d_{ii_1} + \ldots + d_{i_nk} + d_{ki_{n+1}} + \ldots + d_{i_{n+m}k} \geq d_{ii_1} + \ldots + d_{i_nk}$$

And thus $d_{ik} > d_{ii_1} + \ldots + d_{i_nk}$ which is a contradiction with $i \to k$ being minimal before k was made valid. Proof for edge $k \to i$ for any i is very similar. □

Algorithm 5.3 detects and handles negative cycles in the temporal graph in the same manner as Algorithm 5.2.

5.2 Propagation of resource constraints

The temporal graph represents temporal constraints between a set of activities and the activities are allocated to one or more unary resources. In Chapter 3 we introduced a notion of energy precedence constraint for unary resources. It is a pruning technique that allows us to reduce time windows of activities based on their relative positions.

In the temporal graph, both time windows and precedence relations are represented by the temporal constraints. Time window of activity A is represented by the temporal constraint between A and the special starting activity $\overline{S}$. Specifically, the earliest start time of activity A is $est_A = -d_{A\overline{S}}$ and the latest completion time of A is $lct_A = d_{\overline{S}A} + p_A$. The precedence relations can also be deduced from the temporal constraints: if $d_{AB} < 0$, then $B \ll A$.

Thus, we can easily design a procedure (a modification of the procedure in Algorithm 3.1) that calculates the earliest start time of an activity based on the energy precedence constraint. The procedure is listed as Algorithm 5.4.

Algorithm 5.4 calculate_est(A) — calculates the earliest start time of activity A based on energy precedence constraint

```
dur ← 0
end ← inf
for all Y ∈ {X|d_AX < 0 ∧ valid_X = 1 ∧ res_X = res_A} in non-increasing
order of −d_Y S̄ do
  dur ← dur + p_Y
  end ← max(end, −d_Y S̄ + dur)
return end
```

When the procedure finishes, the variable *end* contains the value of earliest start time for activity A based on the energy precedence constraint. To include this information into the temporal graph we may assign

$$d_{A\overline{S}} \leftarrow -calculate\_est(A)$$

We can rewrite this procedure into a form that integrates more naturally with the temporal graph. The procedure listed as Algorithm 5.5 does the same job as procedure in Algorithm 5.4, but it does not use inverted values of weights in the temporal graph.

When this modified procedure finishes, the value in variable *end* can be directly used as a new weight of edge $A \rightarrow \overline{S}$:

$$d_{A\overline{S}} \leftarrow calculate\_est(A)$$

Algorithm 5.5 calculate_est(A) — calculates the earliest start time of activity A based on energy precedence constraint

$dur \leftarrow 0$
$end \leftarrow \sup$
for all $Y \in \{X | d_{AX} < 0 \wedge valid_X = 1 \ \wedge res_X = res_A\}$ in non-decreasing order of $d_{Y\overline{S}}$ **do**
  $dur \leftarrow dur + p_Y$
  $end \leftarrow min(end, d_{Y\overline{S}} - dur)$
return end

We can calculate the latest completion time of an activity in very similar manner.

The presented procedures calculate new temporal constraints between activities $\overline{S}$ and A by exploiting the precedence relations between activities. The activities that must be processed before A affect the earliest start time of A and activities that must be processed after A affect its latest completion time.

The presented concept can be used to calculate new temporal constraints between any pair of activities, not just pairs with $\overline{S}$. We propose a generalization of the notion of the energy precedence constraint that is applicable to any pair of activities.

First, we need to generalize the concept of the earliest start time and the latest completion time. Symbol est_B^A represents the earliest start time of B with regard to A. Formally, if temporal constraint between A and B is defined as $A \xrightarrow{[L,U]} B$, then $est_B^A = L$. Symbol lct_B^A represents the latest completion time of B with regard to A. Formally $lct_B^A = U + p_A$. Note that est_B^A and lct_B^A can be simply expressed in terms of temporal graph weights, namely $est_B^A = -d_{BA}$ and $lct_B^A = d_{AB} + p_A$.

Analogically, the latest start time of activity B with regard to activity A is $lst_B^A = d_{AB}$ and the earliest completion time of B with regard to A is $ect_B^A = -d_{BA} + p_A$

Note that $est_A = est_A^{\overline{S}}$, $lct_A = lct_A^{\overline{S}}$, etc. The activity $\overline{S}$ plays a role of global time reference point. In our generalized approach, any activity can be in the role of reference point for other activities.

The introduced notation can be extended in a natural way to sets of activities. Let A be an activity and let Ω be a set of activities, then:

- $est_\Omega^A = min\{est_B^A | B \in \Omega\}$
- $lst_\Omega^A = max\{lst_B^A | B \in \Omega\}$

- $ect^A_\Omega = min\{ect^A_B | B \in \Omega\}$
- $lct^A_\Omega = max\{lct^A_B | B \in \Omega\}$
- $p_\Omega = \sum\{p_B | B \in \Omega\}$

We define the following sets of activities to simplify the notation.

Set MustBeBefore(A) contains all activities that are valid or undecided, must be allocated to the same resource, and must start before A starts. Formally, it is defined as follows:

$$\text{MustBeBefore(A)} = \{B | valid_B \neq 0 \wedge res_A = res_B \wedge d_{AB} < 0\}$$

Set MustBeAfter(A) contains all valid or undecided activities that must be allocated to the same resource and must start after A. Formally:

$$\text{MustBeAfter(A)} = \{B | valid_B \neq 0 \wedge res_A = res_B \wedge d_{BA} < 0\}$$

Set MustBeAfterBoth(A, B) contains all valid or undecided activities that must be allocated to the same resource as A and B and must be processed after both A and B. If A and B are not allocated to the same resource, the set is empty.

$$\text{MustBeAfterBoth(A, B)} = \text{MustBeAfter(A)} \cap \text{MustBeAfter(B)}$$

Set MustBeBeforeBoth(A, B) contains all valid or undecided activities that must be allocated to the same resource as A and B and must be precessed before both A and B. If A and B are not allocated to the same resource, the set is empty.

$$\text{MustBeBeforeBoth(A, B)} = \text{MustBeBefore(A)} \cap \text{MustBeBefore(B)}$$

Set MustBeBetween(A, B) contains all valid or undecided activities that must be allocated to the same resource as A and B and must be processed after A and before B. If A and B are not allocated to the same resource, the set is empty.

$$\text{MustBeBetween(A, B)} = \text{MustBeAfter(A)} \cap \text{MustBeBefore(B)}$$

Consider a pair of activities A, B such that $A \ll B$. Let

$$\Omega \subseteq \{X | X \in \text{MustBeBetween(A, B)} \wedge valid_X = 1\}\}$$

All activities in Ω are valid and must be executed after activity A and before activity B. Thus, if A starts in time s_A, then B cannot start sooner than

$$s_A + est^A_\Omega + p_\Omega$$

So the set Ω determines the earliest start time of B with regard to A as

$$est_B^A \leftarrow est_\Omega^A + p_\Omega$$

If we consider all sets Ω, we get

$$est_B^A \leftarrow max\{est_\Omega^A + p_\Omega | \Omega \subseteq \{X | X \in \text{MustBeBetween(A, B)} \wedge valid_X = 1\}\}$$

Similarly, let A and B be activities such that $A \ll B$ and

$$\Omega \subseteq \{X | X \in \text{MustBeAfterBoth(A, B)} \wedge valid_X = 1\}\}$$

All activities in Ω are valid and executed after B, so B cannot finish later than

$$lct_\Omega^A - p_\Omega$$

Thus, the latest completion time of B with regard to A is

$$est_B^A \leftarrow lct_\Omega^A - p_\Omega$$

For all sets Ω we get

$$lct_B^A \leftarrow min\{lct_\Omega^A - p_\Omega | \Omega \subseteq \{X | X \in \text{MustBeAfterBoth(A, B)} \wedge valid_X = 1\}\}$$

These rules are a natural generalization of similar rules introduced in Section 3.2.2. The only difference is that the role of the fixed point does not have to be played by the activity $\overline{S}$. The procedure that calculates est_B^A is thus very similar to the procedure that calculates est_B (see Algorithm 5.5). The new procedure is listed as Algorithm 5.6.

Algorithm 5.6 calculate_est(B, A) — calculates the value of d_{BA} based on energy precedence constraint

---

$dur \leftarrow 0$
$end \leftarrow$ sup
for all $C \in$ MustBeBetween(A, B) such that $valid_C = 1$ in non-decreasing order of d_{CA} **do**
  $dur \leftarrow dur + p_C$
  $end \leftarrow min(end, d_{CA} - dur)$
return end

Even though the procedure does not in fact calculate est_B^A (it calculates the value of weight $d_{BA} = -est_B^A$), we prefer to name it $calculate\_est(B, A)$ to emphasize the idea behind the calculation.

Algorithm 5.7 calculate_lct(A, B) — calculates the value of d_{AB} based on energy precedence constraint

```
dur ← 0
end ← sup
for all C ∈ MustBeAfterBoth(A, B) such that valid_C = 1 in non-decreasing order of d_AC + p_C do
    dur ← dur + p_C
    end ← min(end, d_AC + p_C − dur)
return end
```

A similar procedure can be used to calculate the value of lct_B^A, it is listed as Algorithm 5.7.

This procedure calculates $lct_B^A = d_{AB} + p_B$.

Note that both $calculate\_est(B, A)$ and $calculate\_lct(A, B)$ calculate useful results only when $A \ll B$.

The procedure $epc(A, B)$ listed as Algorithm 5.8 calculates the value of d_{AB} based on the energy precedence constraint. If B must be processed before A, then procedure $calculate\_lct(A, B)$ calculates no useful result (note though, that $calculate\_lct(B, A)$ would calculate a useful result, but for weight d_{BA}) and thus there is no need to call it. Hence, the procedure $epc(A, B)$ calls only $calculate\_est(A, B)$ when $B \ll A$.

Using similar reasoning, we conclude that if A must be processed before B, then it is sufficient to call $calculate\_lct(A, B)$. Since $calculate\_lct(A, B)$ returns the value of $lct_B^A = d_{AB} + p_B$, it is necessary to subtract p_B from the final result.

If the order of the activities is not determined, the energy precedence constraint does not filter anything and the procedure returns ∞.

Algorithm 5.8 epc(A, B) — calculates the value of d_{AB} based on energy precedence constraint

```
if B ∈ MustBeBefore(A) then
    return calculate_est(A, B)
else if A ∈ MustBeBefore(B) then
    return calculate_lct(A, B) − p_B
else
    return ∞
```

Chapter 6

Propagation rules

6.1 Constraint Model

In this section we present a CSP model of the temporal graph with optional nodes.

Every node in the temporal graph is marked either as valid, invalid or undecided. We introduce variable $valid_A$ for the node representing activity A. Domain of variable $valid_A$ is $\{0, 1\}$ for undecided variables, $\{1\}$ for valid variables and $\{0\}$ for invalid variables (see [2]). This choice reflects the fact that domains of constraint satisfaction variables gradually shrink as the solving process advances — every undecided activity must be declared either valid or invalid before a solution can be found.

Suppose that activities A and B are represented by nodes a and b in the temporal graph. Then edges $a \rightarrow b$ and $b \rightarrow a$ represent temporal constraint between A and B. We use constraint variable D_{AB} to represent weight d_{ab} and variable D_{BA} to represent weight d_{ba}. Since the temporal graph is a complete graph (and it also contains the edge d_{aa} for every node a), we have to use n^2 variables to represent all edges in the temporal graph.

Domain of D_{AB} contains feasible values for d_{ab}. As the reasoning progresses, d_{ab} can be decreased without limits, but it can never be increased. Thus the lower bound of the domain is inf (negative infinity) and upper bound is the current value of d_{ab}, i. e. domain of D_{AB} is $[\inf, d_{ab}]$. As reader may know, infinite domains are generally not desirable for constraint satisfaction solving over finite domains. This issue is addressed by one of our propagation rules and is described later in greater detail. For now it should suffice to say that if both A and B are valid, then this rule guarantees that domains of both D_{AB} and D_{BA} are finite.

We refer to variables $valid_A$ as *validity variables* and to variables D_{AB} as

temporal variables.

We denote the maximum allowed value (the upper bound of domain) of variable V as V^{max} and we denote the minimum allowed value (the lower bound of domain) of variable V as V^{min}.

6.2 Initialization

The initialization phase serves to set up the initial domains of constraint variables. We assume that the temporal graph initially contains no temporal constraints, i.e. that every temporal constraint represented by the temporal graph is the universal constraint $[\inf, \sup]$. Thus, the weights of the edges in the temporal graph are initially:

- $d_{ab} = \sup$, for edges between every pair of nodes $a \neq b$.
- $d_{aa} = 0$

Domains of the temporal variables are thus:

- $dom(D_{AB}) = [\inf, \sup]$ for every pair of activities A and B.
- $dom(D_{AA}) = [\inf, 0]$ for every activity A.

We also assume that all nodes in the temporal graph are initially undecided. Thus, the domain of every validity variable $valid_A$ is initially the set $\{0, 1\}$.

If some temporal constraints are known in advance, they can be posted using the rule listed as Algorithm 6.1 later in this chapter. If some activities are initially known to be valid, it is possible to simply post the constraint $valid_A \leftarrow 1$ for every such activity. The rule listed as Algorithm 6.3 is triggered by this assignment and updates the temporal constraints in the temporal graph as necessary.

6.3 Propagation rules for temporal constraints

After the initialization, the temporal graph contains no temporal constraints. Thus, all known constraints must be explicitly posted afterwards. Recall that to introduce a new temporal constraint into the temporal graph, weight of $i \rightarrow j$ and/or $j \rightarrow i$ needs to be decreased accordingly. We present the rule in Algorithm 6.1 as a convenience for addition of new temporal constraints (for example, it can be used by a superior system (e.g. a planner) to add a new temporal constraint during the reasoning).

All filtering rules are written in the form *trigger* → *propagator*. Recall that the trigger informs the constraint solver when to execute the propagator. For example, the rule in Algorithm 6.1 is triggered whenever the superior system requests addition of a new temporal constraint, the propagator then appropriately adjusts the weights of edges in the temporal graph.

Also, two special commands may appear in the propagator: **exit** informs the solver that the constraint associated with the propagator is entailed (and thus further invocations of the propagator would not prune anything), **fail** informs the solver that no solution exists.

Algorithm 6.1 Rule for addition of a new temporal constraint

$A \xrightarrow{[L,U]} B$ is added →
$D_{AB}^{max} \leftarrow min(D_{AB}^{max}, \mathrm{U})$
$D_{BA}^{max} \leftarrow min(D_{BA}^{max}, -\mathrm{L})$

Recall that D_{AB}^{max} represents the current value of d_{ab}. When this value is decreased, the temporal graph may no longer be minimal. In the previous chapter we introduced Algorithm 5.2 that restores the minimality of the temporal graph after the weight of some edge is decreased. The filtering rule listed as Algorithm 6.2 is is based on this algorithm.

Algorithm 6.2 Propagation of temporal constraints after weight of some edge is decreased

D_{AB}^{max} is decreased →
if $valid_A = 0 \vee valid_B = 0$ **then**
  exit
if $A = B \wedge D_{AA}^{max} < 0$ **then**
  if $valid_A = 1$ **then**
    fail
  else
    $valid_A \leftarrow 0$
    exit
else
  if $valid_A = 1$ **then**
    for all C such that $valid_C \neq 0$ **do**
      $D_{CB}^{max} \leftarrow min(D_{CB}^{max}, D_{CA}^{max} + D_{AB}^{max})$
  if $valid_B = 1$ **then**
    for all C such that $valid_C \neq 0$ **do**
      $D_{AC}^{max} \leftarrow min(D_{AC}^{max}, D_{AB}^{max} + D_{BC}^{max})$

The rule is triggered whenever D_{AB}^{max} is decreased, i.e. when minimality of the temporal graph is potentially broken.

The rule first checks whether both A and B are still valid or undecided. If one of them is invalid, then the constraint represented by D_{AB} does not affect any solutions and thus may be safely ignored.

Then the rule performs check for negative cycles. If the value of D_{AA}^{max} is negative, then A is a part of a negative cycle. If A is valid, then it is a part of an unbreakable negative cycle and the problem has no solution — the filtering rule ends with a failure. If A is undecided, then it is a part of a breakable negative cycle and the rule breaks it by declaring A invalid.

After that the change in weight d_{ab} is propagated into weights of other edges in the temporal graph. Note that the rule uses only a part of Algorithm 5.2, namely the body of the while-loop. The filtering rule does not need to implement the queue, because whenever it changes domain of some temporal graph variable, the constraint solver triggers the rule in Algorithm 6.2 for that variable automatically.

Proposition 6.1. *The filtering rule Algorithm 6.2 and its subsequent executions make the temporal graph minimal after D_{AB} is decreased (in a previously minimal temporal graph).*

Proof. Direct consequence of Proposition 5.6. □

When validity of some activity changes from undecided to valid, new valid paths are introduced into the temporal graph and thus the graph may no longer be minimal. The filtering rule listed as Algorithm 6.3 is triggered by instantiation of domain of variable $valid_A$. It restores minimality of the temporal graph using the principles introduced in Algorithm 5.3.

Algorithm 6.3 Propagation of temporal constraints after a node is validated

$valid_A$ is instantiated $\rightarrow$
if $valid_A = 1$ **then**
  for all B such that $B \neq A \wedge valid_B \neq 0$ **do**
    for all C such that $C \neq A \wedge valid_C \neq 0$ **do**
      $D_{CB}^{max} \leftarrow min(D_{CB}^{max}, D_{CA}^{max} + D_{AB}^{max})$
exit

Proposition 6.2. *The filtering rule Algorithm 6.3 makes the temporal graph minimal after activity A becomes valid.*

Proof. Direct consequence of Proposition 5.12. □

The Algorithm 6.3 does not explicitly check the negative cycles in the temporal graph after it decreases the value of D_{CB}^{max}. It is not necessary, since the check is performed by Algorithm 6.2 (which is trigerred automatically after the value of D_{CB}^{max} decreases).

6.4 Enhancing propagation of temporal constraints

In this section we introduce a few simple propagation rules, that enhance domain pruning of the temporal variables.

6.4.1 Making domains of temporal variables finite

The temporal variables proposed in section 6.1 always have infinite domains. The reason for this is that the value of weight d_{ab} can be theoretically decreased without limit — thus the variable D_{AB} must have the lower bound of its domain set as negative infinity.

However, we know that valid negative cycles in the temporal graph indicate inconsistency in the temporal constraints. Consider the cycle $a \rightarrow b$, $b \rightarrow a$ in the temporal graph. As long as $d_{ab} + d_{ba} \geq 0$, the cycle is not negative. Thus the weights d_{ab} and d_{ba} limit each other's ability to decrease. For instance, the inequality $d_{ab} \geq -d_{ba}$ must always hold, otherwise there is an inconsistency in the temporal graph. Consequently, we may deduce that if A and B are two valid activities, then $D_{AB} \geq -d_{ba}$, i.e. $D_{AB} \geq -D_{BA}^{max}$. This gives the lower bound for the domain of D_{AB}:

$$D_{AB}^{min} \leftarrow -D_{BA}^{max}$$

The relationship is symmetrical:

$$D_{BA}^{min} \leftarrow -D_{AB}^{max}$$

Whenever a domain of a temporal variable becomes empty due to one of these assignments, then there must be unbreakable negative cycle in the temporal graph and so the failure of solving procedure caused by the empty domain is inevitable and correct.

Note that using the reasoning above, we can prune domain of variable D_{AA} for a valid activity A to a single value: $\{0\}$.

We propose two filtering rules based on these observations. The first one (listed as Algorithm 6.4) reacts to a situation when the weight of an edge in the temporal graph is decreased.

Algorithm 6.4 Pruning lower bounds of temporal variables

D_{AB}^{max} is decreased $\rightarrow$
if $valid_A = 0 \vee valid_B = 0 \vee A = B$ **then**
  exit
if $valid_A = 1 \wedge valid_B = 1$ **then**
  $D_{BA}^{min} \leftarrow -D_{AB}^{max}$

The rule is applicable only to the weight of the edge between a pair of valid activities A and B (it does not have to propagate anything if either of the activities is invalid or if A and B are the same activity). The rule propagates the new upper bound of D_{AB} into the lower bound of D_{BA}.

The second filtering rule is listed as Algorithm 6.5.

Algorithm 6.5 Pruning lower bounds of temporal variables

$valid_A$ is instantiated $\rightarrow$
if $valid_A = 1$ **then**
  for all B such that $valid_B = 1$ **do**
    $D_{AB}^{min} \leftarrow -D_{BA}^{max}$
    $D_{BA}^{min} \leftarrow -D_{AB}^{max}$
exit

When activity A becomes valid, the rule sets the lower bounds for weights of all edges that connect A to other valid activities.

6.4.2 Detectable precedences

So far we have pruned domains of temporal variables only with regard to temporal constraints. But since the activities are allocated to unary resources, we may induce additional temporal constraints based on the fact that no two activities may be processed at the same time on the same resource. Recall that every activity A has a (constant) processing time p_A and a resource res_A on which it must be allocated.

Example 6.1. Consider two activities A, B, both of them allocated to the same resource and a temporal constraint $A \xrightarrow{[-30,5]} B$. Activity B can start at most 30 time units before A starts and it may start at most 5 time units after A starts. Let us assume that $p_A = 10$ and $p_B = 7$. After A starts, it occupies the resource for 10 time units and since B can start at most 5 time units after A starts, it is clear that B must be processed before A. Since B is going to be processed for 7 time units we may deduce temporal constraint $B \xrightarrow{[7,\text{sup}]} A$,

i.e. $A \xrightarrow{[\inf,-7]} B$. By combining the new and old temporal constraint we get $A \xrightarrow{[-30,-7]} B$ or $B \xrightarrow{[7,30]} A$.

This deduction is application of the generalized *detectable precedence*. The concept of detectable precedences has been introduced in [17], where it is used in a situation where time windows and simple precedence relations between activities are known, and the activities are allocated to one unary resource. Whenever time windows of activities A and B are such that they do not allow execution of A after B, then we may deduce precedence $A \ll B$.

We generalize this concept to a situation where temporal constraints are defined between every pair of activities. If temporal constraint between activities A and B that must be allocated to the same resource does not allow B to be processed after A, then B must be processed completely before A starts. Formally, whenever $res_A = res_B$ and $d_{AB} < p_A$ then B must be processed before A and so $d_{AB} \leftarrow min(d_{AB}, -p_B)$

Algorithm 6.6 Detectable precedence

D_{AB}^{max} is decreased $\rightarrow$
if $valid_A = 0 \vee valid_B = 0 \vee A = B$ **then**
  exit
if $res_A = res_B \wedge D_{AB}^{max} < p_A$ **then**
  $D_{AB}^{max} \leftarrow min(D_{AB}^{max}, -p_B)$
exit

The filtering rule in Algorithm 6.6 is triggered when D_{AB}^{max} $(= d_{AB})$ is decreased. The rule is relevant only for a pair of different non-invalid activities — this is the first thing it checks after it is triggered. Then it checks, whether there is enough time to process activity B after activity A. If not, then B must be processed before A and the rule sets d_{AB} accordingly.

6.5 Energy precedence constraint

In this section we introduce the filtering rules that prune domains of temporal variables according to the generalized energy precedence constraint introduced in Section 5.2. First, we define a procedure that calculates the bound for D_{AB}^{max}. This procedure is listed as Algorithm 6.7 and it is an implementation of the procedure listed as Algorithm 5.8.

When activity B must be processed before A, the outcome of procedure $epc(A, B)$ depends on the values of D_{CB}^{max} for valid activities C from set MustBeBetween(B, A). In the other case, when activity A must be processed

Algorithm 6.7 epc(A, B) — calculates value of D_{AB}^{max} based on energy precedence constraint

$end \leftarrow D_{AB}^{max}$
if $B \in$ MustBeBefore(A) **then**
  // B is before A
  $dur \leftarrow 0$
  for all $C \in$ MustBeBetween(B, A) such that $valid_C = 1$ in non-decreasing order of D_{CB}^{max} **do**
    $dur \leftarrow dur + p_C$
    $end \leftarrow min(end, D_{CB}^{max} - dur)$
else if $A \in$ MustBeBefore(B) **then**
  // A is before B
  $dur \leftarrow 0$
  for all $C \in$ MustBeAfterBoth(A, B) such that $valid_C = 1$ in non-decreasing order of $D_{AC}^{max} + p_C$ **do**
    $dur \leftarrow dur + p_C$
    $end \leftarrow min(end, D_{AC}^{max} + p_C - dur)$
  $end \leftarrow end - p_B$
return end (as new D_{AB}^{max})

before B, the outcome depends on values of D_{AC}^{max} for valid activities C from set MustBeAfterBoth(A, B). Thus, the value calculated by $epc(A, B)$ may change whenever:

- temporal constraints change in such a way that a new valid activity becomes a member of MustBeBetween(B, A) or MustBeAfterBoth(A, B)
- value of D_{CB}^{max} for a valid activity C in MustBeBetween(B, A) decreases
- value of D_{AC}^{max} for a valid activity C in MustBeAfterBoth(A, B) decreases
- some activity in set MustBeBetween(B, A) or MustBeAfterBoth(A, B) becomes valid

Activity C can become member of MustBeBetween(B, A) in two ways: either it is a member of MustBeAfter(B) and becomes a member of MustBeBefore(A), or it is a member of MustBeBefore(A) and becomes a member of MustBeAfter(B). Similarly C can become a member of MustBeAfterBoth(A, B) by becoming a member of either MustBeAfter(A) or MustBeAfter(B). In all of these cases, C must be allocated to the same resource as A and B, it cannot be invalid and one of values D_{AC}^{max}, D_{CA}^{max}, D_{CB}^{max}, D_{BC}^{max} must be decreased below 0.

The following filtering rules recalculate the bounds induced by the energy precedence constraint whenever any of the situations mentioned above occurs.

Algorithm 6.8 Propagation of the energy precedence constraint after a new precedence appears in the temporal graph

D^{max}_{AB} is decreased →
if $res_A \neq res_B$ **then**
  exit
if $valid_A = 0 \vee valid_B = 0$ **then**
  exit
if $D^{max}_{BA} < 0$ **then**
  exit
if $D^{max}_{AB} < 0$ **then**
  if $valid_A = 1$ **then**
    for all $C \in$ MustBeBetween(B, A) **do**
      $D^{max}_{BC} \leftarrow epc(B, C)$
    for all $C \in$ MustBeAfterBoth(B, A) **do**
      $D^{max}_{CB} \leftarrow epc(C, B)$
  if $valid_B = 1$ **then**
    for all $C \in$ MustBeBeforeBoth(B, A) **do**
      $D^{max}_{AC} \leftarrow epc(A, C)$
  exit

The rule in Algorithm 6.8 handles the situation when a new precedence $B \ll A$ appears in the temporal graph, that is, whenever value of some D^{max}_{AB} is decreased below 0. This rule only applies to a weight between a pair of non-invalid activities that are allocated to the same resource. If these two conditions are not met, the rule propagates nothing and the associated constraint becomes entailed. Also, if the precedence $A \ll B$ already exists in the temporal graph, the rule no longer needs to wait for $B \ll A$ to appear, since that would introduce a negative cycle into the temporal graph and the solving procedure would either fail or invalidate A or B.

If precedence $B \ll A$ appears, then B becomes a member of MustBeBefore(A) and A becomes a member of MustBeAfter(B) and:

1. Activity A becomes a member of set MustBeAfterBoth(B, C) for C such that $C \in$ MustBeBetween(B, A) and thus if A is valid, then D^{max}_{BC} should be recalculated for all such C (see Figure 6.1).

2. Activity A becomes a member of set MustBeBetween(B, C) for C such that $C \in$ MustBeAfterBoth(B, A) and thus if A is valid, then D^{max}_{CB} should be recalculated for all such C (see Figure 6.2).

3. Activity B becomes a member of set MustBeBetween(C, A) for C such that $C \in$ MustBeBeforeBoth(B, A) and thus if B is valid, then D_{AC}^{max} should be recalculated for all such C (see Figure 6.3).

Figure 6.1: Situation 1. after precedence $B \ll A$ appears

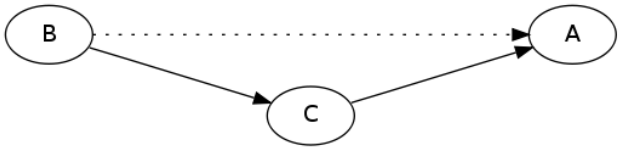


Figure 6.2: Situation 2. after precedence $B \ll A$ appears

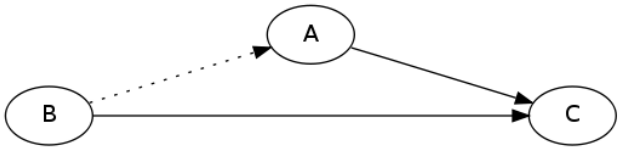
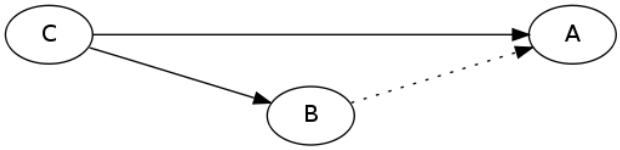


Figure 6.3: Situation 3. after precedence $B \ll A$ appears



This is exactly what the rule does when it detects precedence $B \ll A$ (by checking whether $D_{AB}^{max} < 0$). After that, the global constraint associated with the rule becomes entailed, because every precedence can appear at most once.

The filtering rule listed in Algorithm 6.9 reacts to a situation when the value D_{AB}^{max} decreases and recalculates the energy precedence constraint for all pairs of activities that could be affected by this change. Value D_{AB}^{max} is used to calculate energy precedence constraint in two cases. First, it is used to calculate epc(C, B) if $B \in$ MustBeBefore(C). The value D_{AB}^{max} is used for all valid A such that $A \in$ MustBeBetween(B, C). Thus, if A is valid, value

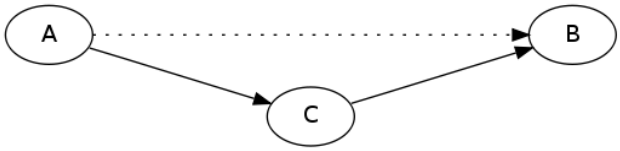
Algorithm 6.9 Propagation of the energy precedence constraint after a weight of an edge in the temporal graph is decreased

D_{AB}^{max} is decreased $\rightarrow$
if $res_A \neq res_B$ **then**
  exit
if $valid_A = 0 \vee valid_B = 0$ **then**
  exit
if $valid_A = 1 \wedge A \in$ MustBeAfter(B) **then**
  for all $C \in$ MustBeAfterBoth(A, B) **do**
    $D_{CB}^{max} \leftarrow epc(C, B)$
if $valid_B = 1 \wedge B \in$ MustBeAfter(A) **then**
  for all $C \in$ MustBeBetween(A, B) **do**
    $D_{AC}^{max} \leftarrow epc(A, C)$

of D_{AB}^{max} is relevant for all C such that $C \in$ MustBeAfterBoth(B, A) (this is the situation depicted in Figure 6.2).

Second, the value of D_{AB}^{max} is used to calculate epc(A, C) if $A \in$ MustBeBefore(C). It is used for all valid B such that $B \in$ MustBeAfterBoth(A, C) and thus if B is valid, value of D_{AB}^{max} is relevant for all C such that $C \in$ MustBeBetween(A, B) (see Figure 6.4).

Figure 6.4: Here, the value of $epc(A, C)$ can change after D_{AB}^{max} is decreased



The filtering rule in Algorithm 6.10 is triggered when an activity A becomes valid. If activity A is a member of MustBeBetween(B, C) for some B, C, then it is necessary to recalculate the value of D_{CB}^{max} (the situation in Figure 6.2). If activity A is a member of MustBeAfterBoth(B, C) and $B \ll C$, then it is necessary to recalculate D_{BC}^{max} (the situation in Figure 6.1).

Algorithm 6.10 Propagation of the energy precedence constraint after an activity is declared valid

$valid_A$ is instantiated $\rightarrow$
if $valid_A = 1$ **then**
  for all $B \in$ MustBeBefore(A) **do**
    for all $C \in$ MustBeAfterBoth(B, A) **do**
      $D_{CB}^{max} \leftarrow epc(C, B)$
    for all $C \in$ MustBeBetween(B, A) **do**
      $D_{BC}^{max} \leftarrow epc(B, C)$
exit

Chapter 7

Conclusion

The contribution of this work can be summarized as follows.

First, we have introduced an extension of the simple temporal problem. The proposed "simple temporal problem with optional variables" can be used to represent temporal constraints between a set of optional activities.

Second, we have proposed an incremental algorithm that guarantees minimality of an STP. We have provided versions of this algorithm for both classical STP and STP with optional variables.

Third, we have introduced the notion of a generalized energy precedence constraint that can be used to adjust temporal constraints between activities. It exploits existing temporal relations and resource constraints in the problem.

Fourth, we have created a set of propagation rules that utilize the above theoretical results. These rules can be implemented as global constraints in a constraint-based scheduling system. They may be used both in pure scheduling to enhance the search procedure and in integrated planning and scheduling to provide useful feedback for the planner.

By integrating the reasoning on temporal relations with resource constraints and optional actitivities the work also brings the concept of simple temporal network closer to practical usage.

The achieved results can be extended in several ways. First, we assume that the activities are allocated to one or more unary resources. In reality, the situation is much more complex and may require that activities are allocated to a variety of different resource types.

Also, there exist several efficient algorithms that can ensure minimality of a triangulated temporal network. We did not use these algorithms because the proposed filtering rules frequently request the temporal constraint between an arbitrary pair of activities. The temporal graph is a complete graph and so this information is always directly available. In the triangu-

lated graph it would have to be calculated. On the other hand, the gains in speed and lower memory usage of triangulated graphs might outweight the disadvantages. Further research is necessary to explore the possibilities of increasing time and space efficiency of temporal constraint propagation.

Next, the algorithm that calculates new bounds for temporal constraints based on the generalized energy precedence constraint is designed as a monolithic algorithm. The efficiency of presented filtering rules could be improved if they used an incremental version of this algorithm.

It is also possible, that we did not discover the full potential of the generalized energy precedence constraint when it comes to pruning of the temporal constraints between activities allocated to unary resources. Further research could yield results in the form of improved pruning.

Bibliography

[1] R. Barták. Constraint-based scheduling: An introduction for newcomers. In *Intelligent Manufacturing Systems*, pages 69–74. IFAC Publications, Elsevier Science, 2003.

[2] R. Barták and O. Čepek. Incremental filtering algorithms for precedence and dependency constraints. In *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*. IEEE Press, 2006.

[3] Christian Bliek and Djamila Sam-haroud. Path consistency on triangulated constraint graphs. In *Proceedings of IJCAI*, pages 456–461, 1999.

[4] H. H. Bui, M. Tyson, and N. Yorke-Smith. Efficient message passing and propagation of simple temporal constraints. In *Proceedings of AAAI 2007 Workshop on Spatial and Temporal Reasoning*, page 9–15, Vancouver, Canada, jul 2007.

[5] J. Carlier and E. Pinson. A practical use of jackson's preemptive schedule for solving the job shop problem. *Annals of Operations Research*, 26(1-4):269–287, 1990.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[7] R. Dechter. *Constraint Processing*. Morgan Kauffman, 2003.

[8] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[9] J. Erschler. *Analyse sous contraintes et aide à la décision pour certains problèmes d'ordonnancement*. PhD thesis, Université Paul Sabatier, Toulouse, France, 1976.

[10] Antonio Garrido and Federico Barber. Integrating planning and scheduling. *Applied Artificial Intelligence*, 15(5):471–491, 2001.

[11] Philippe Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results. *Artif. Intell.*, 143(2):151–188, 2003.

[12] W. Nuijten. *Time and resource constrained scheduling: A constraint satisfaction approach.* PhD thesis, Eidhoven University of Technology, 1994.

[13] Claude Le Pape. Implementation of resource constraints in ILOG schedule: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3:55–66, 1994.

[14] Léon Planken, Mathijs de Weerdt, and Roman van der Krogt. P3C: A new algorithm for the simple temporal problem. In *ICAPS*, pages 256–263, 2008.

[15] C. Schulte. *Programming Constraint Services, High-Level Programming of Standard and New Constraint Services.* Springer Verlag, 2002.

[16] Julie A. Shah and Brian C. Williams. Fast dynamic scheduling of disjunctive temporal constraint networks through incremental compilation. In *ICAPS*, pages 322–329, 2008.

[17] P. Vilím and R. Barták. Filtering algorithms for batch processing with sequence dependent setup times. In *Proceedings of the 6th AIPS International Conference on Artificial Intelligence Planning and Scheduling*, pages 312–320. AAAI Press, 2002.

[18] Lin Xu and Berthe Y. Choueiry. A new efficient algorithm for solving the simple temporal problem. *International Syposium on Temporal Representation and Reasoning*, 0:212, 2003.