Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Marek Vlk

Dynamic Scheduling

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	prof. RNDr. Roman Barták, Ph.D.
Study programme:	Informatics
Specialization:	Theoretical Computer Science

Prague 2014

I would like to thank Mr. Roman Barták for supervising this thesis, and members of the FlowOpt project for providing their source code.

Název práce: Dynamic Scheduling

Autor: Marek Vlk

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: prof. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Jedním z problémů rozvrhování výroby v reálném životě je dynamičnost výrobních prostředí zahrnující nové výrobní požadavky a rozbíjející se zařízení během vykonávání rozvrhu. Prosté přerozvržení od nuly v reakci na neočekávané události, které nastávají v provozu, může vyžadovat nadměrný výpočetní čas. Obnovený rozvrh může být navíc neúnosně odchýlený od toho probíhajícího. Tato práce podává přehled o stávajících přístupech v oblasti dynamického rozvrhování a navrhuje postupy jak upravit rozvrh při vyrušení, jako je například selhání zdroje, příchod naléhavé objednávky nebo její zrušení. Důraz je kladen na rychlost navržených procedur i na minimální modifikaci původního rozvrhu. Rozvrhovací model vychází z projektu FlowOpt, který je založen na temporálních sítích s alternativami. Algoritmy jsou napsány v jazyce C#.

Klíčová slova: Rozvrhování, opravy rozvrhů, přerozvržení, pediktivní-reaktivní rozvrhování

Title: Dynamic Scheduling

Author: Marek Vlk

Department: Department of Theoretical Computer Science and Mathematical Logic

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

Abstract: One of the problems of real-life production scheduling is dynamics of manufacturing environments with new production demands and breaking machines during the schedule execution. Simple rescheduling from scratch in response to unexpected events occurring on the shop floor may require excessive computation time. Moreover, the recovered schedule may be prohibitively deviated from the ongoing schedule. This thesis reviews existing approaches in the field of dynamic scheduling and proposes techniques how to modify a schedule to accommodate disturbances such as resource failure, hot order arrival or order cancellation. The importance is put on the speed of suggested procedures as well as on a minimum modification from the original schedule. The scheduling model is motivated by the FlowOpt project, which is based on the Temporal Networks with Alternatives. The algorithms are written in the C# language.

Keywords: Scheduling, schedule updates, rescheduling, predictive-reactive scheduling

Contents

Introduction

Scheduling is a decision-making process that is used in manufacturing and service industries. It plays an important role in procurement and production, in transportation and distribution, and in information processing and communication. The aim of scheduling is to allocate limited resources to the activities, which has to be done in such a way that the company optimizes its objectives and achieves its goals. Resources may be machines in a workshop, runways at an airport, crews at a construction site, or processing units in a computing environment. Activities can be operations in a workshop, take-offs and landings at an airport, stages in a construction project, or computer programs that have to be executed. Objectives may take many different forms, such as minimizing the time to complete all activities (makespan), minimizing the number of activities that are completed after due dates, and so forth. [1, 2]

In manufacturing models, a resource is usually referred to as a machine, and a task that has to be processed on a machine is often called a job. In production processes, a job may be a single operation or a collection of operations that have to be done on various machines. In manufacturing environment, developing a detailed schedule of the tasks to be performed helps maintain efficiency and control of operations.

In the real world, however, manufacturing systems face uncertainty due to unexpected events occurring on the shop floor. Machines break down, operations take longer than anticipated, personnel do not perform as expected, urgent orders arrive, others are cancelled, etc. These disturbances render the ongoing schedule infeasible. In such case, a simple approach is to collect the data from the shop floor when the disruption occurs and to generate a new schedule from scratch. Gathering the information and total rescheduling involve excessive amount of time which may lead to failure of the scheduling mechanism and thus have far-reaching consequences.

For these reasons, reactive scheduling, which may be understood as the continuous correction of precomputed predictive schedules, is becoming more and more important. On the one hand, reactive scheduling has certain things in common with some predictive scheduling approaches, such as iterative improvement of some initial schedule. On the other hand, the major difference between reactive and predictive scheduling is the on-line nature and associated real-time execution requirements. The schedule update must be accomplished before the running schedule becomes invalid, and this time window may be very small in a complex manufacturing environment.

The aim of this thesis is to propose a new technique to tackle disturbances, such as machine breakdown and new order arrival, in the manufacturing system. Given an ongoing schedule and an unexpected event occurring at a certain time making the schedule infeasible, the intention is to find a feasible schedule as similar to the original one as possible, and as fast as possible.

In this work we borrow the scheduling model from the FlowOpt project [3]. Simply said, a schedule consists of activities, resources and constraints. Activities require resources to process them and all resources may perform at most one activity at a time. Possible positions of activities in time are restricted by temporal constraints.

The scope of dynamic scheduling is miscellaneous and there is no standard classification scheme. Moreover, some terms are used ambiguously. The following chapter provides a short trip into the field of dynamic scheduling. The chapter 2 describes some existing schedule repair techniques. The chapter 3 explains some existing techniques that are employed further in the algorithms. The formal definition of our scheduling model is given in chapter 4. Complexity of resource failure recovery and our proposed method are presented in chapter 5. Other disturbances and their solutions are described in chapter 6, and summary and future work suggestions are given in Conclusion.

1. Terminology

In this chapter we gather basic definitions concerning dynamic scheduling, and contextualize this thesis.

1.1 Basic Definitions

According to [4] a *manufacturing system* is "the collection of operations and processes used to produce a desired product".

A *production schedule* specifies, for each activity established to be executed, the planned start and end times along with a set of resources, to which the activity is assigned.

Scheduling is the process of creating a production schedule for a given set of orders and resources.

Rescheduling is the process of updating an existing production schedule in response to changes in the manufacturing system. This includes the arrival of new orders, machine breakdowns, and machine repairs.

1.2 Rescheduling Framework

A *rescheduling framework* is presented in [5]. The framework includes *rescheduling environments*, identifying the sets of jobs to schedule, *rescheduling strategies*, describing whether or not schedules are generated, *rescheduling policies*, specifying when to reschedule, and *rescheduling methods*, describing how to generate and update schedules.

1.2.1 Rescheduling Environment

While *static environments* have a finite set of jobs, *dynamic environments* have infinite set of jobs. Static environments can be either deterministic, where is no uncertainty, or stochastic, taking into account uncertainty of variables.

Dynamic environments consider infinite set of jobs arriving over an infinite time horizon. Each job must be scheduled before it can be executed. Dynamic environments are divided according to variability in the arrival process. First, if there is no uncertainty in the arrival process, jobs are known in advance, and the production schedule is continuously repeated, we talk about *cyclic production.* Second, if there is some uncertainty in job arrivals, but their routes through the manufacturing system equal, and the arrival rate is steady, it is usually referred to as *flow shop.* Third, *job shops* involve process flow variability as well as the variability in job arrivals.

1.2.2 Rescheduling Strategies

There are two basic strategies for controlling production in dynamic environments with uncertain job arrivals. The first strategy does not create production schedules, but decentralized production control methods dispatch jobs when necessary, using information available at the moment of dispatching. Jobs waiting for processing at a resource are chosen by using a dispatching rule or other heuristics. This strategy is often referred to as *completely reactive scheduling*, on-line scheduling, or dynamic scheduling. Note that this is not the prime scope of this thesis, whose title Dynamic Scheduling has been picked to refer to non-static scheduling approaches in general.

The second strategy is usually called *predictive-reactive scheduling.* It has two primary steps. The first one generates a production schedule; the second one corrects the schedule in response to disturbances and other changes within the environment.

According to [6], predictive-reactive scheduling is based on schedule modifications considering only shop efficiency. In such case the new schedule may be fundamentally deviated from the original one, which may cause poor performance owing to affecting other planning activities based on the original schedule. Therefore, the two more following strategies (approaches) may be distinguished.

Robust predictive-reactive scheduling takes a focus on generating schedules to minimize the impacts of disruptions on the performance. A usual solution is to consider not only schedule efficiency, but mainly deviation from the original schedule (termed *stability*).

In some cases the disruptions are predictable or even expected, which may be exploited when computing a predictive (original) schedule. This is the aim of *robust pro-active scheduling.* Such methods usually add some temporal slack among operations of a job in order to absorb some level of uncertainty without rescheduling.

To make matters more confusing, robust pro-active scheduling is also referred to as robust predictive-reactive scheduling and then robust predictive-reactive scheduling amalgamates with predictive-reactive scheduling.

1.2.3 Rescheduling Policies

Implementing a rescheduling strategy (except for a completely reactive approach) requires a rescheduling policy. At least three policies may be distinguished. First, *periodic policy* updates a schedule periodically according to a given frequency. Second, *event-driven policy* reschedules the system when a disruption occurs. It can happen repeatedly in dynamic environments, or it can be just a single event to revise a schedule in a static system. Third, *hybrid policy* combines both former approaches.

1.2.4 Rescheduling Methods

Rescheduling methods are divided into two groups: *complete rescheduling*, and *schedule repair.* Complete rescheduling computes a new schedule from scratch. This might principally give the best solution in terms of optimality, but this solution may require excessive computation time that is usually unavailable, and also may result in instability and lack of continuity, leading to extra production costs.

The goal of this thesis is to propose a new schedule repair method in the field of event-driven robust predictive-reactive scheduling, working as fast as possible while giving reasonably good solution. For this reason, we review schedule repair methods in the following chapter in more details.

2. Related Methods

When dealing with disturbances, reactive scheduling systems usually attempt to revise only necessary parts of the ongoing schedule to avoid rescheduling from scratch, because schedule repair methods have the advantage in terms of computation time and stability. Such methods are reviewed in this chapter, based on the classification in [6, 7].

The predictive schedules are usually based on optimisation principles. It is obvious that any correction will cause a deviation from the predictive schedule and thus performance measures will no longer be optimal. Therefore, the focus has been to find a technique that handles the schedule recovery without deterioration in the quality.

On the other hand, the schedule must be recovered and ready to replace the ongoing schedule while it is still active, i.e. before it becomes infeasible or obsolete. Otherwise the manufacturing process fails. For these reasons, the time span needed by the algorithm to react to unexpected events must also be considered when evaluating the schedule repair performance.

2.1 Heuristics

Heuristic-based approaches do not guarantee to find an optimal solution, but respond in a short time. The simplest schedule repair technique is the *right shift rescheduling*. This algorithm shifts the operations globally to the right on the time axis in order to cope with disruptions. When it arises from machine breakdown, the method introduces gaps in the schedule, during which the machines are idle. It is obvious that this approach results in schedules of bad quality, and can be used only for environments involving minor disruptions.

The shortcomings of total rescheduling and right shift rescheduling gave rise to another approach: *affected operation rescheduling*, also referred to as partial schedule repair. The idea of this algorithm is to reschedule only the operations directly and indirectly affected by the disruption in order to minimize the deviation from the initial schedule.

The *Prec-rep* algorithm proposed in [8] is worth mentioning although it is designed not for the recovery of schedules being executed, but for repairing violated precedence and resource constraints in manually altered schedules. The algorithm sweeps over the violated constraints and cures them in such a way that the activity that is to precede another one is shifted to the left and the succeeding one to the right. This gives significantly better results when compared to right shift rescheduling.

2.2 Meta-heuristics

Meta-heuristics such as simulated annealing or genetic algorithms are high level heuristics guiding local search methods to escape from local optima. Local search methods are based on probing into neighbourhoods, i.e. the algorithms start from some given solution and iteratively improve it using move operators.

Thus, when such algorithm reaches a solution that cannot be directly improved, which means that the algorithm gets stuck in a local optimum, it terminates. Meta-heuristics help these techniques to jump from local optima by (occasional) accepting worse solutions or by generating better initial solutions for local probing in some sophisticated way.

Since the local search heuristics may be trapped in a poor local optimum, using meta-heuristics is a good way to their enhancement. On the other hand, meta-heuristics require higher computational effort, which holds especially for genetic algorithms with an increasing problem size.

An example of integrating local search and heuristic procedures is the *iterative flattening search* [9], which has been designed for finding predictive schedules minimizing makespan. The algorithm iterates two steps. First, in the relaxation step, some precedence constraints, which have been added into the model so as to resolve resource restrictions, are retracted. Second, in the flattening step, some precedence constraints are added into the model in order to make the schedule feasible again. The iterative flattening search is reported to have been enhanced by tabu search meta-heuristic technique to achieve fine-grained exploration, and by introducing partial order schedules aimed at increasing temporal flexibility in the temporary solutions [10].

2.3 Artificial Intelligence Approaches

Some techniques from the field of artificial intelligence and knowledge-based systems are also used in dynamic scheduling problems. *Case-based reasoning* [11] is applicable to domain specific problems and allows continuous learning from past cases, but requires an extensive experience database, which involves time-consuming search through. *Fuzzy logic* [12] requires the knowledge of the domain to be built into the algorithm and learning of the algorithm is impossible. *Neural networks* [13] provide very fast responses and predict the repair strategy according to past experience, which, however, may require excessive re-training time.

Another approach, which is rather an independent branch, is *multi-agent based architectures* [14, 15]. In multi-agent systems independent agents cooperate in order to achieve a common goal. Albeit this is one of the most promising approaches to building complex, robust, and effective scheduling systems owing to their distributed, autonomous and dynamic nature, but the coordination among the agents is hard to achieve.

2.4 Minimal Perturbation Problem

As mentioned above, predictive schedules are based on optimisation principles. Regardless of what the optimisation objective is, it is often desired, in order to handle changes in the model definition, to find a schedule that is as similar as possible to the initial one. This problem is formulated as a *minimal perturbation problem* [16].

The minimal perturbation problem is designed for example to reconstruct school timetable when new requirements come, in such a way that the impact of the timetable modification on people and lectures is minimized. Methods seeing schedule repair as the minimal perturbation problem are described in [17]. Unfortunately, due to the huge computational burden, this approach cannot be used in manufacturing systems in which the time frame for schedule update is very small.

2.5 Repair-DTP

The algorithms discussed in the scheduling literature usually do not consider presence of temporal constraints (minimal and maximal time lags) in the scheduling model. Therefore the methods described above cannot be simply implemented in the manufacturing systems involving minimal and maximal distance constraints.

The *Repair-DTP* algorithm proposed in [18] tackles a problem the most similar to ours, however, it is designed to correct violated constraints in manually edited schedules. The model involves precedence constraints and synchronization constraints, but not temporal distance constraints. Nonetheless, the Repair-DTP algorithm employs *simple temporal networks* [19] and *incremental full path consistency* algorithm [20] to reduce searching space. If a feasible correction exists, the algorithm tries to find the most similar schedule to the initial one through only shifting activities in time. Since the Repair-DTP algorithm does not try changes in resource selection, it cannot be used to deal against machine failure. Moreover, the main shortcoming of the algorithm is searching through disjunctions, introduced by hierarchical nature of the model and by resource unarity. This leads to excessive (exponentially growing) amount of temporal networks that are inspected, which requires unacceptable amount of time.

3. Basic Terms and Solving Techniques

This chapter briefly describes basic concepts and methods that are employed in this thesis.

3.1 Constraint Satisfaction Methods

A *Constraint Satisfaction Problem* (CSP) [21] is a triple (X, D, C), where

- $X = \{x_1, ..., x_n\}$ is a set of variables,
- $D = \{D_1, ..., D_n\}$ is a set of nonempty domains of values for each variable,
- $C = \{C_1, ..., C_m\}$ is a set of constraints.

Each variable x_i can take on the values from the domain D_i. Constraint C_j involves some subset of the variables and specifies the allowable combinations of values for that subset. However, we work only with constraints concerning just two variables (=a binary CSP).

A state of the problem is an *assignment* (or an instantiation) of values to some or all of the variables $(x_i = a_i, x_j = a_j, ...)$. An assignment that does not violate any of the constraints is called *feasible* or *consistent.* An assignment that includes all variables is called *complete.* A *solution* of a CSP is a complete and feasible assignment.

CSPs are typically solved using a form of search. The most used techniques are variants of *backtracking* (described next), and *constraint propagation* (of which the aim is to prune the domains of variables).

3.1.1 Backtracking and Backjumping

Backtracking assigns variables one after another in the given order, and when a particular variable has been unsuccessfully tried to assign all values from its domain, it goes back to the most recently assigned variable and tries another value, and so on. More formally, let $x_1, ..., x_n$ be the variables and $x_1 = a_1, ..., x_i = a_i$ for $i < n$ be the partial assignment. If all values of variable x_{i+1} have been tried without finding a solution, there is no solution that extends the current assignment $x_1 = a_1, ..., x_i = a_i$. Backtracking then goes back to the variable x_i and, if possible, changes its value, otherwise backtracks further.

The cause why variable x_{i+1} could not have been successfully assigned is often not variable x_i itself, so that attempting to assign other values to x_i may be pointless. There usually exists an index $j < i$ such that the partial assignment $x_1 = a_1, ..., x_j = a_j$ cannot be extended to make up a solution with any value of x_{i+1}. In this case it is useful to try another value directly for x_j instead of x_i.

The efficiency of such a *backjump* depends on how far it is able to jump. It is of course important not to skip any solution. If a backjump does not skip over any solution, then it is called a safe jump. More precisely, if an algorithm jumps back from x_{i+1} to x_j such that $x_1 = a_1, ..., x_j = a_j$ cannot be extended to form a solution with any value of x_{i+1}, the jump is safe.

Backjumping algorithms usually carry out the longest backjump that can be proved to be safe. Various methods are used to determine safety of a jump. Gaschnig backjumping [22] takes into account which constraints actually caused a conflict, but jumps only once — when it succeeds in assigning a value to a variable, it then goes back only one level like in chronological backtracking. Graph-directed backjumping [23] can conduct a number of backjumps, but is directed only by the structure of constraints, which means that it neglects satisfying or violating a particular constraint.

Advantages of both methods are taken in the Conflict-directed backjumping algorithm (CBJ) [24], which maintains a conflict set for each of variables. When a value of the current variable x_i conflicts with an assigned value of some past variable x_j, the variable x_j is added to the conflict set of x_i. When all values of the current variable x_i have been tried, the algorithm jumps back to the most recently assigned variable x_j in the conflict set of x_i, and, also, the variables in the conflict set of x_i (except for x_j) are added to the conflict set of x_j, thus all information about conflicts is kept.

3.1.2 Backmarking

Another way to speed up the naïve backtracking is to eliminate redundant consistency checks. The point of backmarking [25] is firstly to avoid checking consistency when it is known it would fail, and secondly to avoid consistency check when it is known it would succeed.

Suppose consistency checking is conducted as follows. When a variable x_i is being instantiated, the variable x_1 is checked against x_i. If this succeeds, then x_2 is checked against x_i, and so on up to x_{i-1} is checked against x_i, or until some test fails. To reduce the number of consistency checks, two structures must be introduced.

First, let *Mark*[1] be an $n \times m$ array, where n is the number of variables, and m is the size of the largest domain. All elements are initially set to 0. During the computation, suppose x_i is instantiated with value k, and x_h $(h < i)$ is checked against x_i. If the test fails, $Mark[i, k]$ is set to h. If all tests succeed, $Mark[i, k]$ is set to $i - 1$. Therefore, if variable x_i cannot be instantiated with value k, then $Mark[i, k]$ links to the lowest-indexed variable that precludes value k from being assigned to variable x_i.

Second, let *BackTo*[2] be an array of size n, where n is the number of variables. All elements are initially set to 0. During the computation, when we backtrack from x_i to x_{i-1}, $BackTo[i]$ is set to $i - 1$, and for all j, $i < j \leq n$, $BackTo[j]$ is set to the minimum of $BackTo[j]$ and $i - 1$. Therefore, $BackTo[i]$ records the lowest-indexed variable whose value has been re-instantiated since x_i was last instantiated.

When value k is assigned to variable x_i, the following test is performed.

- If $Mark[i, k] < BackTo[i]$, it follows that the consistency check has already failed against some variable x_h, where $h < BackTo[i]$, and now the consis-

[1] *Mark* is in some literature denoted as *mcl* – maximum checking level

[2] *BackTo* is in some literature denoted as *mbl* – minimum backtracking level

tency check would fail again because x_h has not changed its value. This is in literature referred to as *type A saving.*

- If $Mark[i,k] \geq BackTo[i]$, it follows that the consistency checks have already passed with all variables x_h, where $h < BackTo[i]$, and that these variables still have the same values, therefore the consistency checks have to be conducted only for variables x_g, where $BackTo[i] \leq g < i$. This is referred to as *type B saving.*

Consequently, thanks to these added comparisons, backmarking checks for consistency with only such variables that the result is not known.

3.1.3 Conflict-Directed Backjumping with Backmarking

While the aim of backjumping is to reduce the number of nodes visited in the search tree generated by a backtracking algorithm, the aim of backmarking is to omit redundant consistency checks. Both these methods may be easily combined into one [25]. However, in order to retain advantages of both methods to the best, a slight modification is useful [26]. For this time, let *BackTo* be an $n \times m$ array, where n is the number of variables and m the size of the biggest domain. Now each $Mark$ entry has a corresponding *BackTo* entry. The value of $BackTo[i,k]$ points to the lowest-indexed variable whose instantiation has changed since variable x_i was last assigned value k.

This algorithm, which is actually an underlying coat of a part of the STN-recovery algorithm described later on, is depicted in algorithm 1.

The outer while cycle in the code ensures that all n variables are set – the variable i serves as a pointer to the variable $x[i]$ that is being assigned. The array $newVals$ stores for each variable a value that is to be assigned when the corresponding variable is visited. Initially, $newVals[i]$ is 0 for all i. In the inner while cycle, the assignment attempt is conducted as follows. Firstly, the values of $Mark$ and *BackTo* are compared, as explained in the previous section, in order to determine whether it makes sense to try to assign the value $newVal$ to the variable $x[i]$. If it is known that the consistency check would fail, the algorithm proceeds to assigning the incremented value of $newVal$. Otherwise, the value is assigned to variable $x[i]$ and the consistency check is conducted in the for cycle. If the consistency check fails, it, similarly, proceeds to another value, otherwise the flag $successful$ is set to $true$ in order to jump out of the while cycle.

When the line behind the inner while cycle is reached, it may be because of two causes. First, in case of fail, which means that all values from the domain of variable $x[i]$ have been unsuccessfully tried, the algorithm finds the maximum element of $cs[i]$ (which is a conflict set of $x[i]$), denoted j, merges $cs[i]$ into $cs[j]$ except j, updates the values of *BackTo*, and, finally, jumps back to the variable $x[j]$. Oppositely, in case of success, which means that some value has been successfully assigned to variable $x[i]$, the algorithm stores the variable $newVal$, from which the search should continue in the case that $x[i]$ is revisited, and proceeds to the variable $x[i+1]$. When the value of i exceeds the number of variables n, it means that the solution has been found.

Algorithm 1 Conflict-Directed Backjumping with Backmarking

```
function CBJ-BM
    $i \leftarrow 1$
    while $i \leq n$ do
        $newVal \leftarrow newVals[i]$                                    ▷ initially 0
        $successful \leftarrow false$
        while $\neg successful$ & $newVal < m$ do
            if $Mark[i][newVal] < BackTo[i][newVal]$ then
                $cs[i] \leftarrow cs[i] \cup \{Mark[i][newVal]\}$
                $newVal \leftarrow newVal + 1$
                continue                                                  ▷ type A saving
            end if
            $x[i] \leftarrow newVal$
            $fail \leftarrow false$
            for $j \leftarrow BackTo[i][newVal]$ to $i - 1$ do                ▷ type B saving
                if $\neg Consistent(x[j], x[i])$ then
                    $cs[i] \leftarrow cs[i] \cup \{j\}$
                    $Mark[i][newVal] \leftarrow j$
                    $fail \leftarrow true$
                    break
                end if
            end for
            if $\neg fail$ then
                $Mark[i][newVal] \leftarrow i - 1$
                $successful \leftarrow true$
            end if
            $BackTo[i][newVal] \leftarrow i$
            $newVal \leftarrow newVal + 1$
        end while
        if $\neg successful$ then
            $j \leftarrow Max(cs[i])$                ▷ latest index in $cs[i]$ is where to jump
            $cs[j] \leftarrow cs[j] \cup cs[i] \setminus \{j\}$
            for $k \leftarrow j + 1$ to $n$ do
                for $v \leftarrow 0$ to $m - 1$ do
                    $BackTo[k][v] \leftarrow Min(BackTo[k][v], j)$
                end for
            end for
            while $i > j$ do                                              ▷ jump back to j
                $newVals[i] \leftarrow 0$
                $cs[i] \leftarrow \emptyset$
                $i \leftarrow i - 1$
            end while
        else
            $newVals[i] \leftarrow newVal$
            $i \leftarrow i + 1$
        end if
    end while
end function
```

3.2 Temporal Reasoning

3.2.1 Simple Temporal Networks

Simple Temporal Network (STN) [19] is a pair (V, C), where V is a set of time points $v_1, ..., v_n$, and C is a set of constraints $c_1, ..., c_m$ limiting time distance between two time points. Each constraint can be written as $v_j - v_i \leq w_{ij}$, where $v_i, v_j \in V$ and $w_{ij} \in \mathbb{Z}$. The number w_{ij} is often called a *weight* of the constraint and it expresses maximal time distance between the time points, i.e. v_j occurs at most w_{ij} time units after v_i occurs. Minimal time distance between the time points v_i and v_j can be obtained through reverse constraint, i.e. $v_j - v_i \geq -w_{ji}$.

It is also useful to introduce a special time point v_0 that constitutes the origin of the time horizon. The time point v_0 is referred to as the *global predecessor*. It allows introducing unary constraints, i.e. $v_i - v_0 \leq w_{0,i}$ means that the maximal possible value of the time point v_i is $w_{0,i}$, and oppositely $v_i - v_0 \geq -w_{i,0}$ means that the minimal possible value of v_i is $-w_{i,0}$.

The solution of an STN is an assignment of values to all variables in the set V, such that all constraints C are satisfied. An STN is called consistent if it has at least one solution.

An important term to define is an oriented path from v_{i_1} to v_{i_h}:

$$P_{i_1,i_h} = v_{i_1}, v_{i_2}, ..., v_{i_h}$$

and its weight

$$w(P_{i_1,i_h}) = \sum_{k=1}^{h-1} w_{i_k,i_{k+1}}$$

It is often necessary to know the maximal distance of two time points even if there is not a constraint between these time points. This is the motivation to have the STN with a property all pairs shortest path (APSP), which means that for each pair of distinct time points v_i and v_j the following holds. If there exists a path from v_i to v_j, then $c_{ij} \in c$ and w_{ij} equals the weight of minimal path from v_i to v_j, i.e. $w_{ij} = \min_{P_{ij}} w(P_{ij})$; otherwise w_{ij} is infinity.

3.2.2 Incremental Full Path Consistency

Once we have an STN with the APSP property, it is also useful to be able to add constraints to the STN while conserving the APSP property. This is what the Incremented Full Path Consistency (IFPC) algorithm [20] does. It adds a constraint to the STN and ensures that the modified STN still satisfies the APSP property (see algorithm 2).

The input of the algorithm is a new constraint $v_b - v_a \leq wNew$ (or the new weight for the constraint already present in the STN). The algorithm in the first for cycle updates the paths from v_k to v_a and from v_b to v_k, i.e. the paths that have the updated constraint at their start or at their end. In the second for cycle, the algorithm updates all paths emanating from and emanating to the already updated time points, i.e. paths that may contain the updated constraint at their middle. Since the paths emanating to each of the time points in $Jset$ include the updated constraint, it suffices to update only the paths emanating to some time point in *Jset* that traverse the time point v_a – it is guaranteed that the join of the two paths includes the updated constraint.

Algorithm 2 Incremental Full Path Consistency

```
function IFPC(int $a$, int $b$, long $wNew$)
    if $w[a,b] \leq wNew$ then
        return redundant
    end if
    if $wNew + w[b,a] < 0$ then
        return inconsistent
    end if
    $w[a,b] \leftarrow wNew$
    $Iset \leftarrow \emptyset$
    $Jset \leftarrow \emptyset$
    for $k \leftarrow 0$ to $n$ do
        if $k \neq a$ & $k \neq b$ then
            if $w[k,b] > w[k,a] + w[a,b]$ then
                $w[k,b] \leftarrow w[k,a] + w[a,b]$
                $Iset \leftarrow Iset \cup \{k\}$
            end if
            if $w[a,k] > w[a,b] + w[b,k]$ then
                $w[a,k] \leftarrow w[a,b] + w[b,k]$
                $Jset \leftarrow Jset \cup \{k\}$
            end if
        end if
    end for
    for all $i \in Iset$ do
        for all $j \in Jset$ do
            if $i \neq j$ & $w[i,j] > w[i,a] + w[a,j]$ then
                $w[i,j] \leftarrow w[i,a] + w[a,j]$
            end if
        end for
    end for
    return consistent
end function
```

Owing to the nested loop, the worst case complexity of IFPC is quadratic in the number of time points, but whether a constraint addition is consistent, inconsistent or redundant, is decided in constant time via the first two conditions.

An STN with the APSP property is created through iterative adding the constraints into the initially empty STN via the IFPC algorithm. This STN build-up is done in $\mathcal{O}(mn^2)$ time, where n is the number of time points in the STN, and m is the number of constraints to be added.

4. Scheduling Model

This chapter describes the scheduling model we deal with.

4.1 Informal Description

The scheduling model is motivated by the FlowOpt project [3], which contains a tool for designing and editing *manufacturing workflows.* Workflows in the FlowOpt model match up the structure of Nested Temporal Networks with Alternatives [27]. In this thesis, however, we take into consideration only activities, so that it is not necessary to run into details of the workflows. Therefore, a scheduling problem consists of activities that are planned to be executed.

Constraints

A mutual position in time of two distinct activities may be limited by a *temporal constraint.* In the FlowOpt model, the constraints include precedences (one activity has to be accomplished before the execution of another activity can begin), synchronizations (one activity has to start/end exactly when another activity starts/ends), and temporal distances (determining minimal or maximal time lags between activities).

Notice that what these constraints do is imposing minimal and/or maximal temporal distance between two activities. A precedence constraint between activities A_1 and A_2 determines that the distance of the start time of activity A_2 from the start time of activity A_1 is at least the duration time of A_1. As to synchronizations, for example the constraint enforcing activities A_1 and A_2 to start at the same time determines that the distance between the start time of activity A_1 and the start time of activity A_2 is (at least and at most) 0. Other constraints involve maximal and minimal distances in the same manner.

Further, as explained in section 3.2.1, a minimal distance between two time points may be expressed as a maximal distance of two time points of a negative value. More precisely, if a time point, say $Start(A_2)$, is set to occur at least w time units after another time point, say $Start(A_1)$, i.e. $Start(A_2) - Start(A_1) \geq w$, then it is exactly the same as $Start(A_1) - Start(A_2) \leq -w$. Therefore, in order to simplify and generalize working with the constraints, we understand in the rest of this thesis temporal constraints just as constraints determining maximal distance between start times of two distinct activities.

There are also logical constraints in the FlowOpt model, which restrict mutual occurrence of activities in the resulting schedule. However, since we take into account only activities that are scheduled to be performed, logical constraints are not of concern for our purposes.

Resources

Activities are processed on *resources.* A resource may be for example a worker, a machine, a tool, etc. All resources are unary, which means that each resource may perform no more than one activity at a time. This limitation is often referred to as a *resource constraint.* An important feature in the FlowOpt model is that an activity may require a number of resources to be executed. A resource that can participate in performing an activity is called a *resource dependency* of the activity. Resource dependencies of an activity are divided into disjoint subsets called *resource dependency groups* of the activity. To make a schedule feasible, exactly one resource dependency from each resource dependency group of an activity must be *selected*.

Nevertheless, the fact that the activities may require more resources does not make the problem harder. It may be modelled through disintegrating the activities. More formally, suppose activity A contains k resource dependency groups $\{G_1, ..., G_k\}$. Then, we can replace activity A by k activities such that A_1 has resource dependency group G_1, and so on up to A_k having resource dependency group G_k. Further, set the durations of activities $A_1, ..., A_k$ to equal the duration of activity A, and bound all these activities with $(k-1)$ synchronization constraints to enforce that they start at the same time. As far as existence of a feasible schedule is concerned, these models are obviously equivalent.

Consequently, in order to simplify the description and explanation of the suggested algorithms, it is assumed throughout that each activity has only one resource dependency group from which the activity requires one resource dependency to be selected[1].

4.2 Formal Definition

4.2.1 Scheduling Problem

Scheduling problem P is a triplet of three sets: *Activities*, *Constraints*, and *Resources*.

- $\mathit{Activities} = \{\text{all activities in } P\}$
- $\mathit{Constraints} = \{\text{all temporal constraints in } P\}$
- $\mathit{Resources} = \{\text{all available resources in } P\}$

Each activity A is specified by its start time $\mathit{Start}(A)$ and end time $\mathit{End}(A)$, which we will look for, and fixed duration $\mathit{Duration}(A)$, which is part of the problem specification.

Since we do not allow preemptions (interruptibility of activities), $\mathit{Start}(A) + \mathit{Duration}(A) = \mathit{End}(A)$ holds.

Temporal constraints

Constraints determine mutual position in time of two distinct activities. Constraint $C \in \mathit{Constraints}$ is a triplet (A_i, A_j, w), where $A_i, A_j \in \mathit{Activities}$, $w \in \mathbb{Z}$, and the semantics is following.

$$\mathit{Start}(A_j) - \mathit{Start}(A_i) \leq w \tag{4.1}$$

[1]Although the algorithms on the attached CD are implemented and tested allowing multiple resource dependency groups.

Now, some terminology from the graph theory deserves to be clarified in terms of the scheduling model. Activities A_i and A_j are called *adjacent* if there exists a constraint (A_i, A_j, w) or (A_j, A_i, w) for any $w \in \mathbb{Z}$.

Two activities A_i and A_j are *connected* if there exists a sequence of activities $A_i, A_{i+1}, ..., A_{j-1}, A_j$ such that A_i and A_{i+1} are adjacent, A_{i+1} and A_{i+2} are adjacent, ..., A_{j-1} and A_j are adjacent.

A *connected component* is a maximal (in terms of inclusion) subset of activities such that all activities from the subset are connected. Each activity as well as each constraint belongs to exactly one connected component.

Resource Constraints

Let $A \in \mathit{Activities}$, then the set of resources that may process activity A is denoted $\mathit{Resources}(A)$. The set $\mathit{Resources}(A)$ is often referred to as a resource group.

Each activity needs to be allocated to exactly one resource from its resource group. Let $A \in \mathit{Activities}$, then a resource $R \in \mathit{Resources}(A)$ is *selected* if resource R is scheduled to process activity A, which we denote $\mathit{SelectedResource}(A) = R$.

Each activity must have a selected resource to make a schedule feasible. Formally:

$$\forall A \in \mathit{Activities} : \mathit{SelectedResource}(A) \neq \mathit{null}$$

All resources in a schedule are unary, which means that they cannot execute more activities simultaneously. Therefore, in a feasible schedule for all activities $A_i \neq A_j$ the following holds.

$$\begin{aligned} \mathit{SelectedResource}(A_i) &= \mathit{SelectedResource}(A_j) \\ &\Rightarrow \mathit{End}(A_i) \leq \mathit{Start}(A_j) \vee \mathit{End}(A_j) \leq \mathit{Start}(A_i) \end{aligned} \tag{4.2}$$

Special cases

Real-life scheduling problems are usually designed in such a way that there are subsets of resources that share certain capabilities and which then constitute resource groups of activities. For example, having resources divided into two groups: workers and pots, then boiling requires arbitrary pot, moulding requires arbitrary worker and so on. It is worth defining at least two special cases according to what conditions are fulfilled by all resource groups.

Nested The resource groups of a scheduling problem are *nested* [1] if one and only one of the following conditions holds for any two resource groups $\mathit{Resources}(A_1)$ and $\mathit{Resources}(A_2)$ of two distinct activities A_1 and A_2.

- $\mathit{Resources}(A_1)$ is equal to $\mathit{Resources}(A_2)$
  $(\mathit{Resources}(A_1) = \mathit{Resources}(A_2))$
- $\mathit{Resources}(A_1)$ and $\mathit{Resources}(A_2)$ do not overlap
  $(\mathit{Resources}(A_1) \cap \mathit{Resources}(A_2) = \emptyset)$

- $Resources(A_1)$ is a subset of $Resources(A_2)$
  $(Resources(A_1) \subset Resources(A_2))$
- $Resources(A_2)$ is a subset of $Resources(A_1)$
  $(Resources(A_2) \subset Resources(A_1))$

Equivalent The resource groups of a scheduling problem are *equivalent* if one and only one of the following conditions holds for any two resource groups $Resources(A_1)$ and $Resources(A_2)$ of two distinct activities A_1 and A_2.

- $Resources(A_1)$ is equal to $Resources(A_2)$
  $(Resources(A_1) = Resources(A_2))$
- $Resources(A_1)$ and $Resources(A_2)$ do not overlap
  $(Resources(A_1) \cap Resources(A_2) = \emptyset)$

If the resource groups are neither equivalent, nor nested, they are called *arbitrary.*

4.2.2 Schedule

A schedule S (sometimes referred to as a resulting schedule or a solution) is acquired by allocating activities in time and on resources. Allocation of activities in time means assigning particular values to the variables $Start(A)$ for each $A \in Activities$. Allocation of activities on resources means selecting a particular resource ($SelectedResource(A)$) from the resource group ($Resources(A)$) of each activity $A \in Activities$.

To make a schedule *feasible*, the allocation must be conducted in such a way that all the temporal constraints 4.1 as well as all the resource constraints 4.2 in the model are satisfied.

4.3 Rescheduling Problem

The problem we actually deal with in the rest of this thesis is that we are given a particular instance of the scheduling problem along with a feasible schedule, and also with a change in the problem specification. The aim is to find another schedule that is feasible in terms of the new problem definition. The feasible schedule we are given is referred to as an original schedule or an ongoing schedule.

Formally, let $R = (P_0, S_0, \delta^+, \delta^-)$ be a rescheduling problem, which is given by the original scheduling problem P_0, the original feasible schedule S_0, elements δ^+ to be added to the problem P_0, and elements δ^- to be removed from the problem P_0. New scheduling problem P_1 is then $P_0 \cup \delta^+ \setminus \delta^-$. The task of the rescheduling problem R is then to find a schedule S_1 for problem P_1, of which the quality is measured with respect to the original schedule S_0.

The way the scheduling problem can be modified depends on the disturbance. In case of a machine breakdown (chapter 5), we are given a resource that cannot be used (from a certain time point) while the set of activities remains unchanged, therefore $\delta^+ = \emptyset$ and δ^- is the broken down resource.

As to other disturbances, a hot order arrival (chapter 6.1) adds a set of activities, which is referred to as an order, into the model, and the aim is to allocate them the earlier the better. Thus δ^+ is the set of activities in the new order and $\delta^- = \emptyset$.

Oppositely, in an order cancellation (chapter 6.2), there is a set of activities that are removed from the problem. In this case $\delta^+ = \emptyset$ and δ^- is the set of activities in the cancelled order. This is easy to solve because after the activities are simply deallocated, the schedule is still feasible. However, it might be of interest to squeeze the schedule to decrease total completeness or lateness.

Finally, a machine repair (chapter 6.3) adds a resource to the problem, which may be used, again, to decrease total completeness or lateness. Hence δ^+ is the repaired resource and $\delta^- = \emptyset$.

As explained in previous chapters, regardless of what the optimization objective of the original schedule is, it seems to be wise to modify the schedule in such a way that the new schedule is as similar to the original one as possible. For this purpose we need to evaluate the modification distance.

Let us denote $Activities(P)$ the activities in the scheduling problem P, and $Start_S(A)$ denote the start time of activity A in schedule S. Then $Activities(R) = Activities(P_0) \cap Activities(P_1)$. In the rest of this thesis, we distinguish the following distance functions.

$$f_1 = \sum_{A \in Activities(R)} |Start_{S_1}(A) - Start_{S_0}(A)|$$

$$f_2 = |\{A \in Activities(R) \mid Start_{S_1}(A) \neq Start_{S_0}(A)\}|$$

$$f_3 = \max_{A \in Activities(R)} |Start_{S_1}(A) - Start_{S_0}(A)|$$

5. Machine Breakdown

Given a formal definition of the model, we can now describe the most feared disturbance: a machine breakdown. This disturbance, which is also referred to as a machine or resource failure, may happen in the manufacturing system at any point in time, say t_f, and means that a particular resource cannot be used anymore, i.e. for all $t \geq t_f$. This makes further questions arise, e.g., whether the activities that were being processed at time t_f are devastated and thus must be performed from the beginning, whether their predecessors must be also re-executed if there are only solutions violating temporal constraints, and scores more.

For the sake of simplicity, let us assume that a resource fails at the beginning of the time horizon (at time point $t = 0$), i.e. right before the schedule execution begins. The resource that fails is in what follows also referred to as a forbidden resource. Formally, let S_0 be the schedule to be executed and R_f be the failed resource; the aim is to find a feasible schedule S_1, such that R_f is not used at any point in time $t \geq 0$. As mentioned in previous chapters, the intention is to find S_1 as fast as possible and, regardless of the initial objectives, the more similar to S_0, the better. S_1 is referred to as a recovered schedule. The decision variant of the described problem, which means determining whether or not there exists a feasible recovered schedule regardless of any distance and objective function, is what we call a resource failure recovery problem.

This chapter firstly discusses the complexity of this problem, and then we suggest two different algorithms.

5.1 Complexity

The problem of existence of a feasible schedule is known to be NP-complete (which also follows from the forthcoming theorem). However, one might think that when we already have a feasible schedule solving the problem, then reallocating activities from the forbidden resource requires only little (easy) correction of the schedule. Nonetheless, we will show that the resource failure recovery problem is still NP-complete. The proof exploits the idea from [28].

Theorem. *Given a feasible schedule and one resource to which any activity cannot be allocated anymore, deciding existence of any feasible recovered schedule is NP-complete.*

Proof. The problem is in NP, because testing feasibility of a recovered schedule may be done in linear time in the number of constraints and resources. As to NP-hardness, we shall show that an independent set problem (known to be NP-complete) can be (in a polynomial time) reduced to a resource failure recovery problem. In the independent set problem, there is an undirected graph $G = (V, E)$ on n vertices $(v_1, ..., v_n)$ with m edges, and integer k. The question is whether or not there exists a subset of vertices $V_I \subseteq V$ such that $|V_I| = k$ and for each $v_i, v_j \in V_I : (v_i, v_j) \notin E$. We construct the resource failure recovery problem (see figure 5.1) as follows:

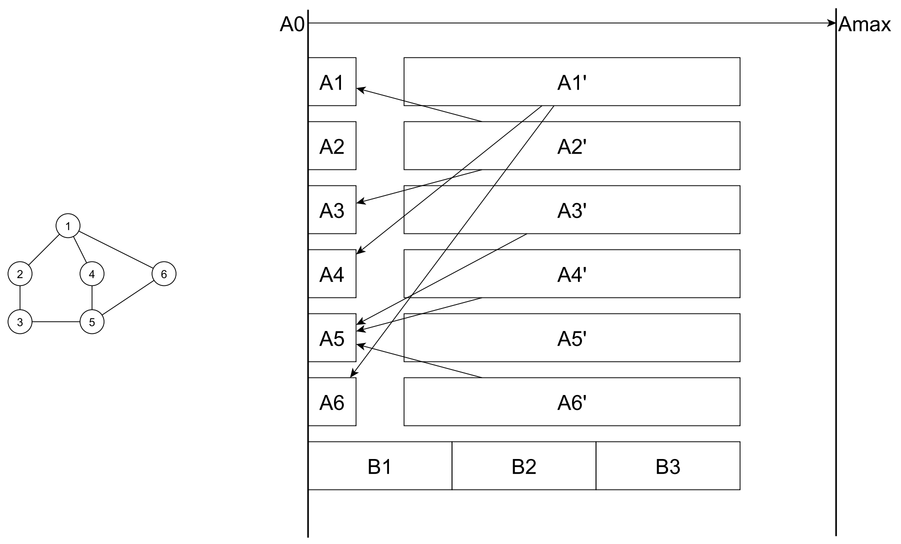

Figure 5.1: Schedule to recover, built from the independent set problem.

- Let us have $n + 1$ resources $R_1, ..., R_n$; and R_f, which is to fail
- For $i = 1$ to n: create activities A_i and A'_i such that $Duration(A_i) = 1$, $Duration(A'_i) = 3k - 2$, $Resources(A_i) = Resources(A'_i) = \{R_i\}$, $SelectedResource(A_i) = SelectedResource(A'_i) = R_i$, $Start(A_i) = 0$, and $Start(A'_i) = 2$
- For $i = 1$ to k: create activity B_i such that $Duration(B_i) = 3$, $Resources(B_i) = \{R_1, ..., R_n, R_f\}$, $SelectedResource(B_i) = R_f$, and $Start(B_i) = 3(i - 1)$
- Create dummy activities A_0 and A_{max}
- For $i = 1$ to n: add precedence constraint between A_0 and A_i, i.e. $(A_i, A_0, 0)$
- For $i = 1$ to k: add precedence constraint between A_0 and B_i, i.e. $(B_i, A_0, 0)$
- For $i = 1$ to n: add precedence constraint between A'_i and A_{max}, i.e. $(A_{max}, A'_i, -(3k - 2))$
- For $i = 1$ to k: add precedence constraint between B_i and A_{max}, i.e. $(A_{max}, B_i, -3)$
- For $i = 1$ to n: add minimal distance constraint of value 1 between A_i and A'_i, i.e. $(A'_i, A_i, -2)$
- For each edge (v_i, v_j) from E arbitrarily oriented: add maximal distance constraint of value 2 between A_i and A'_j, i.e. $(A_i, A'_j, 3)$
- Add maximal distance constraint of value $3k + 2$ between A_0 and A_{max}, i.e. $(A_0, A_{max}, 3k + 2)$

The schedule is constructed in such a way that activities $B_1, ..., B_k$, allocated to a resource that has broken down, must be reallocated to distinct resources between activities A_i and A'_i (figure 5.2). It is easy to see that (indices of) resources

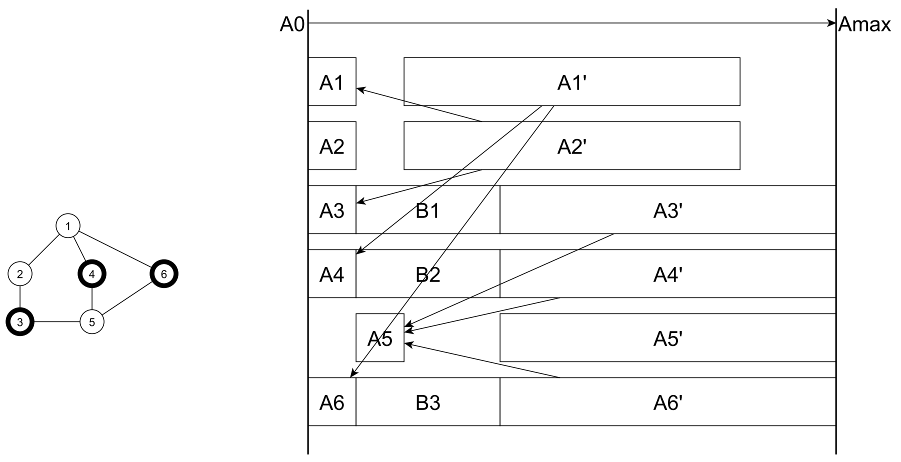

Figure 5.2: Recovered schedule and corresponding independent set.

selected to process activities $B_1, ..., B_k$ correspond to (indices of) vertices in V making the independent set of size k. On the other hand, if activities $B_1, ..., B_k$ cannot be reallocated without violating any temporal distance constraint, there is not an independent set of size k in the given graph.

□

Notice that the allocation of activities in the original schedule in the reduction did not actually play any role. It demonstrates that the original schedule may be absolutely unhelpful.

Furthermore, the resource groups in the proof are nested. Motivated by the nature of real-life scheduling problems and their need for speed, the following algorithms assume that the resource groups are equivalent (see definition in 4.2.1).

5.2 Right Shift Affected

In this chapter we describe a greedy algorithm to tackle the machine breakdown disruption. For each $A \in \mathit{Activities}$, it is assumed throughout that the forbidden resource is deleted from the resource group of activity A, i.e. $\mathit{Resources}(A) = \mathit{Resources}(A) \setminus \mathit{ForbiddenResource}$.

The algorithm is aimed at moving as few activities as possible, i.e. optimizing in distance function f_2. The idea is to reallocate activities from the forbidden resource and then keep reallocating activities that violate some constraint until the schedule is feasible.

We will now describe how to move (reallocate) the activities, how to repair the constraints, and in what order to pick the activities to repair the constraints.

5.2.1 Reallocating Activities

Firstly, we discuss how to reallocate activities. Suppose the algorithm wants to repair a constraint in such a way that an activity A should be reallocated to a time point t. The natural idea was to reallocate the activity A exactly to the time point t even if there is no resources available for the required $\mathit{Duration}(A)$. Then, when a repair function checks constraints satisfaction, it would have to check the resource constraints too and then repair according to the resource constraint violation. Unfortunately, there always turned out to be a model for which this method gets stuck in an infinite loop, regardless of implementations of both the constraint repairing and the sequence of activities to be repaired.

Consequently, the algorithm always allocates activity A in such a way that it does not violate any resource constraint. This is achieved through seeking a time point t^* (which is greater or equal to time point t) where activity A can be allocated without violating the resource constraints. Formally, when the algorithm desires to allocate activity A to time point t, then activity A is allocated to time point t^*, such that $t^* \geq t$ and $\forall t' : t' \geq t \wedge t' < t^*$ activity A cannot be allocated in t' without overlapping some other activity on any resource from $\mathit{Resources}(A)$.

Checking resource availability

In order to express whether or not a resource is free at a specified time interval, let us first define $\mathit{Impedimentary}(A, R, t)$ as a set of activities that preclude activity A from being allocated on resource R at time t.

$$\begin{aligned}\mathit{Impedimentary}(A, R, t) = \{A' \mid A' \in \mathit{Activities} \wedge R = \mathit{SelectedResource}(A')\\ \wedge\, (t < \mathit{End}(A') \leq t + \mathit{Duration}(A) \vee t \leq \mathit{Start}(A') < t + \mathit{Duration}(A))\}\end{aligned}$$

Now we can define a set of resources where activity A can be allocated at time t as such:

$$\mathit{AvailableResources}(A, t) = \{R \mid R \in \mathit{Resources}(A) \wedge \mathit{Impedimentary}(A, R, t) = \emptyset\}$$

Another question is which resource the algorithm should select if there are more resources available. Since the resource groups in the model are assumed to be equivalent, it seems useful to pick the resource on which the activity best fits in terms of surrounding gaps. Therefore, the following heuristic is used.

Earliest Succeeding Start Latest Previous End (ESSLPE) Rule

Suppose activity A is about to be allocated at time t (see figure 5.3). The algorithm picks the resource with the earliest (closest) occupied time after the time point $t + \mathit{Duration}(A)$ (= earliest succeeding start), which holds for the resources number 3 and 4 in the figure 5.3. Like in this case, when there are more resources with the same earliest succeeding start, then the algorithm picks the resource with the latest (closest) occupied time before the time point t (= latest previous end), which is met by the resource number 4 in the figure 5.3. A resource that has at least some activity to process is always preferred to an empty resource.

To describe the rule formally, let us first define the earliest succeeding start time as follows.

$$\begin{aligned}\mathit{SuccStart}(A, t) = \min_{A' \in \mathit{Activities}} \{\mathit{Start}(A') \mid t + \mathit{Duration}(A) \leq \mathit{Start}(A')\\ \wedge\, \mathit{SelectedResource}(A') \in \mathit{AvailableResources}(A, t)\}\end{aligned}$$

Then, let $CandidateResources1(A,t)$ be the subset of $AvailableResources(A,t)$ with the earliest succeeding activities defined as follows.

$$\begin{aligned} Candidate1(A,t) = \{&R \in AvailableResources(A,t) \\ &\mid \exists A' \in Activities : Start(A') = SuccStart(A,t) \\ &\qquad \wedge R = SelectedResource(A')\} \end{aligned}$$

It may happen that $Candidate1(A,t)$ is empty, because there may be no succeeding activity (so that $SuccStart(A,t) = \infty$). For this reason, assign:

$$CandidateResources1(A,t) = \begin{cases} Candidate1(A,t) & \text{if } Candidate1(A,t) \neq \emptyset \\ \\ AvailableResources(A,t) & \text{otherwise} \end{cases}$$

As to the example in the picture 5.3, the set $CandidateResources1(A,t) = \{3,4\}$.

Further, let us define the latest previous end as such.

$$\begin{aligned} PrecEnd(A,t) = \max_{A' \in Activities} \{&End(A') \mid t \geq End(A') \\ &\wedge SelectedResource(A') \in CandidateResources1(A,t)\} \end{aligned}$$

Similarly, let $CandidateResources(A,t)$ be the subset of $CandidateResources1(A,t)$ with the latest preceding activities defined as follows.

$$\begin{aligned} Candidate2(A,t) = \{&R \in CandidateResources1(A,t) \\ &\mid \exists A' \in Activities : End(A') = PrecEnd(A,t) \\ &\qquad \wedge R = SelectedResource(A')\} \end{aligned}$$

Analogously, it may happen that $CandidateResources(A,t)$ is empty, because there may be no preceding activity. For this reason, assign:

$$CandidateResources(A,t) = \begin{cases} Candidate2(A,t) & \text{if } Candidate2(A,t) \neq \emptyset \\ \\ CandidateResources1(A,t) & \text{otherwise} \end{cases}$$

In the picture 5.3, the set $CandidateResources(A,t) = \{4\}$.

Finally, the algorithm picks arbitrary resource from $CandidateResources(A,t)$.

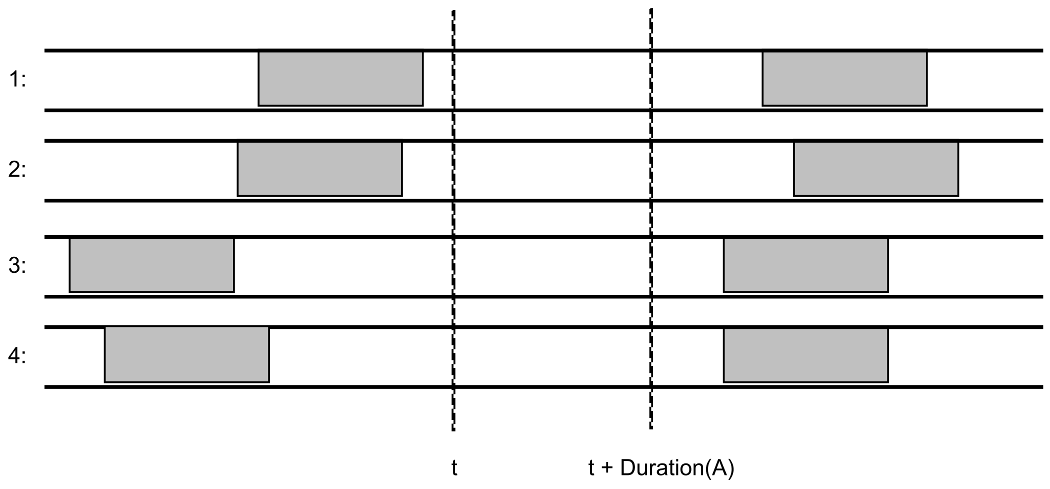

Figure 5.3: Illustration for ESSLPE rule

Reallocation

The procedure `ReallocateActivity` (see algorithm 3) obtains two parameters: an activity to allocate (A) and a time point where it is desired to allocate the activity (t). The seeking for an available resource starts at time t, but the activity is ultimately allocated to the time point t^*, where an available resource is found.

Algorithm 3 Reallocating an activity

```
function ReallocateActivity(Activity $A$, TimePoint $t$)
    $SelectedResource(A) \leftarrow$ null
    $Start(A) \leftarrow$ null
    $t^* \leftarrow \min_{t' \geq t}\{t' \mid CandidateResources(A, t') \neq \emptyset\}$
    $Start(A) \leftarrow t^*$
    $SelectedResource(A) \leftarrow$ arbitrary from $CandidateResources(A, t^*)$
end function
```

5.2.2 Constraint Repair

Now we describe how the violated constraints are repaired. When a temporal constraint between activities A_1 and A_2 of weight w is violated, it means that the distance between $Start(A_1)$ and $Start(A_2)$ is greater than allowed. Then the algorithm seeks for possible allocation of A_1 from the minimal time point that satisfies the constraint rightwards.

Here is where the title of the algorithm comes from. It repairs temporal constraints via moving activities to the right, which, of course, may cause violation of other temporal constraints. An important property is that when the algorithm picks an activity to be repaired, then it iterates over all temporal constraints associated with the activity being repaired until the activity does not violate any associated constraint.

Moving activities to the left, when trying to repair violated constraints, is intuitively unsubstantiated, because if it was beneficial to allocate some activity sooner in time, then the activity would have been allocated sooner (in the original schedule[1]). However, as the rescheduling process proceeds, it may suitably make some free space for moving activities leftwards. We tried several methods involving moving to the left, but, again, the algorithm always came across a model that made it never end. Moreover, for the models the algorithm finished, the improvement was negligible. Right shifting approach gives good tradeoff between time-consumption and schedule similarity.

Regardless of the order, in which the activities are selected to be repaired, the entire `RightShiftAffected` algorithm works as follows (see algorithm 4). First, it goes through all activities in the model and checks whether the activity uses the forbidden resource. In the positive case, the activity is reallocated through the `ReallocateActivity` procedure (the seeking for available resources starts at the original start time of the activity), and the activity is added to the set $affected$. Now, none of the activities uses the forbidden resource and the set $affected$ contains activities that have been reallocated and therefore must be checked for temporal constraint violation.

Next, the algorithm takes an activity from the set $affected$ and proceeds to repair all violated temporal constraints associated with the activity in question. It repairs the constraints, as described, through moving activities to the right, so that if another activity is moved, it is added into the set $affected$ because it must be then checked for constraint violation. Recall that `ReallocateActivity` procedure always allocates an activity such that it does not violate any resource constraint, so that only temporal constraints are checked here. If the activity has been successfully healed, which means that the activity does not violate any constraint, the algorithm proceeds to another one from $affected$. The order of taking activities from $affected$ will be discussed further.

5.2.3 Order of Activities

As shown in the code, the activities that are moved in time during the healing process are added to the structure $affected$. Finally, we should discuss how to pick activities from $affected$. We examined five strategies that found a feasible schedule for any solvable model:

Left – Pick the leftmost activity, i.e. the activity with the minimum $Start(A)$.

Right – Opposite, pick the rightmost activity, i.e. the activity with the maximum $Start(A)$.

Queue – FIFO: pick the activity that has been in the structure for the longest time (the least recently added).

Stack – LIFO: pick the activity that has been in the structure for the shortest time (the most recently added).

Random – Pick activities uniformly at random.

[1]But it was not allocated there probably because of unavailability of resources or due to some earliness objective – either reason is likely to persist in case of a machine breakdown

Algorithm 4 Right Shift Affected

```
function RightShiftAffected
    $affected \leftarrow \emptyset$
    for all $A \in Activities$ do
        if $SelectedResource(A) = ForbiddenResource$ then
            ReallocateActivity($A$, $Start(A)$)
            $affected \leftarrow affected \cup \{A\}$
        end if
    end for
    while $affected \neq \emptyset$ do
        $A \leftarrow PopFrom(affected)$
        while $(A_1, A_2, w) \in ViolatedConstraints(A)$ do
            ReallocateActivity($A_1$, $Start(A_2) - w$)
            if $A_1 \neq A$ then
                $affected \leftarrow affected \cup \{A_1\}$
            end if
        end while
    end while
end function
```

Comparison of these strategies is depicted in figure 5.4. The x-axis corresponds to the size of the schedule in the number of activities. (For details how the schedules are generated see appendix A). The y-axis corresponds to the value of distance function f_1. Other performance measures – i.e. distance functions f_2 and f_3, number of activities that swapped the resource to process them, deterioration on makespan and total callings of `ReallocateActivity` – also show that the best strategy is picking the rightmost activity. The explanation is that shifting the rightmost activities rightwards makes consecutively free space for shifting the activities allocated more on the left, which would otherwise have to creep over one another. (For all results refer to appendix B.1.)

One might ask how the algorithm behaves with respect to the density of constraints. This is depicted in figure 5.5. All scheduling models are of the same size (2400 activities). The x-axis corresponds to the number of constraints within one connected component (each consisting of at most 8 activities), the y-axis corresponds to the run-time of the algorithm in milliseconds. (For information about the testing machine refer to appendix A.3.) Again, other performance measures also show that picking the rightmost activity is the best strategy.

5.2.4 Correctness

The soundness of the algorithm clearly follows from the fact that it keeps healing the constraints until all constraints are satisfied and therefore the schedule is feasible. Unfortunately, the question whether the algorithm always ends and finds the solution, provided the schedule is recoverable, is still open.

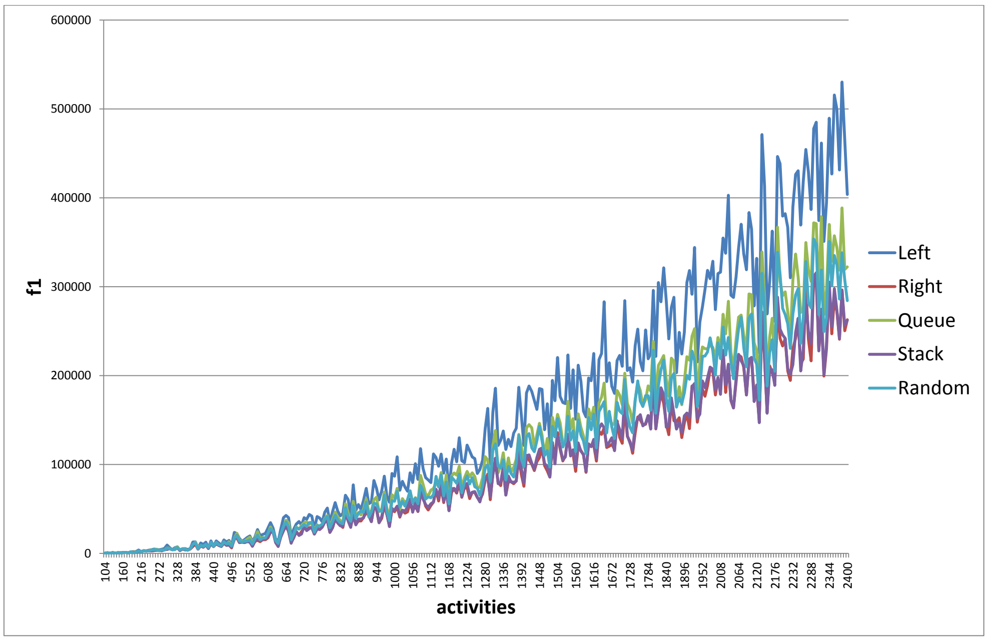

Figure 5.4: Comparison of schedule quality (measured using function f1) for different activity selection heuristics

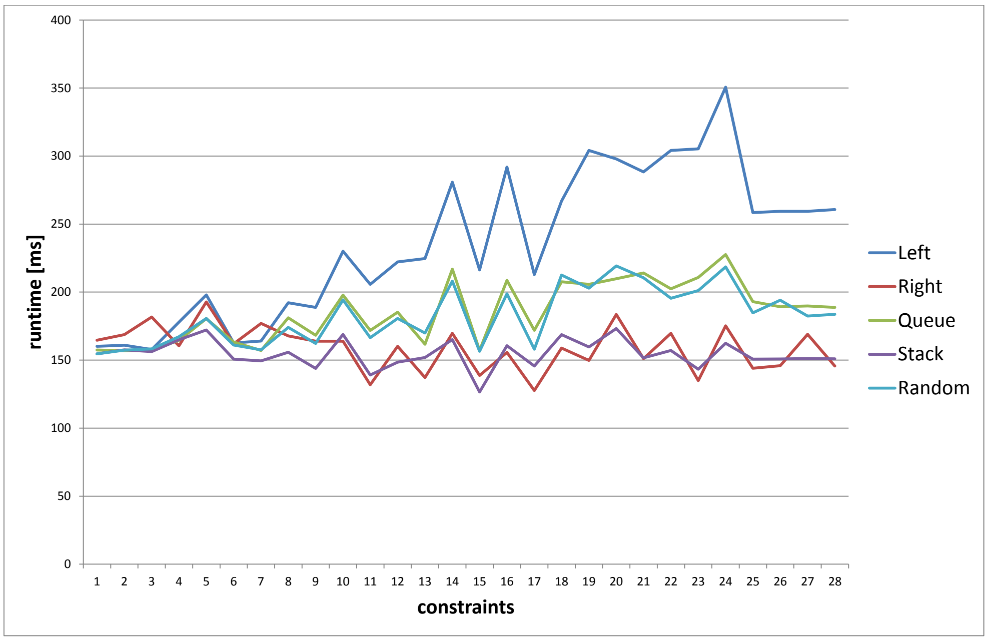

Figure 5.5: Comparison of schedule quality (measured in run-time) for different activity selection heuristics

5.2.5 Discussion

If there is no feasible schedule recovery, the algorithm gets stuck in an infinite loop. This is obviously the main shortcoming of the algorithm. One possible way to detect unrecoverability of the schedule is by passing and checking a time limit. Another way is to check where an activity is being allocated, and if the activity is allocated at a time point exceeding a certain threshold, it may be considered as an unsuccessful finding of a schedule.

Another drawback is that if the alternative resources for the broken-down resource make a bottleneck, the affected activities (and subsequently all connected components with them) are moved to the end of the schedule horizon. This might lead to very poor performance in function f_3, which is unacceptable when the original schedule objective is related to lateness or tardiness. This is the motivation for a different approach described in the following section.

5.3 STN-Recovery

In this chapter, a more sophisticated algorithm to tackle the machine breakdown is given. This algorithm anticipates that moving a large number of activities by small time is preferable to moving activities a lot in time. Since iterative correcting tends to shift the activities to the end of the schedule horizon, it is necessary to deallocate some set of already scheduled activities and then allocate them back again. This is what is now meant by reallocation.

The point of the algorithm is to allocate connected components one after another through conflict-directed backjumping. The allocation of an activity is carried out such that the start time of an activity is continuously incremented until an available resource at that time is found, or until the maximal value of the start time (based on the start times of already allocated activities) is exceeded. In the former case the algorithm proceeds to allocate the next activity, in the latter case the algorithm goes back to reallocate some previous activity. Since this allocation process might involve excessive computational burden, it is useful to prune the search space based on the fact that a resource failure leads only to deterioration of the schedule in the original optimization objective. Moreover, the group of resources where the broken down resource belongs is now likely to make a bottleneck. This assumption is used in such a way that the activities are reallocated from the broken down resource to available resources and then the activities are shifted so as they do not overlap each another – thus the minimal potential start times for allocation are obtained – and then the reallocation process can begin.

Firstly, the skeleton of the algorithm is given, and next, its particular steps are described in more details.

5.3.1 Skeleton of STN-Recovery

The STN-recovery algorithm uses the structures and techniques described in chapter 3. The STN (including the global predecessor) with the APSP property is assumed to have already been computed from the temporal constraints in the model; the resource constraints are not involved in the STN. Recall that the

APSP property of the STN provides us the two-dimensional array w, of which the values say that $Start(A_j) - Start(A_i) \leq w[i, j]$, where $A_i, A_j \in Activities$.

A sketch of the STN-recovery algorithm decomposed into 6 steps follows.

1. Find activities allocated to the forbidden resource and change their resource selection from the forbidden resource to an available resource, picking the resource with the lowest usage. Now some activities allocated on the same resource may overlap.

2. In order to find out which activities should be reallocated, do the following. For each resource (to which some activity has been added in step 1) shift the activities that overlap (to the right) so as they do not overlap, and add them into the set $affected$. Include in $affected$ also activities that were not actually shifted but are allocated on the right of those shifted.

3. For the sake of pruning the search space of the forthcoming reallocation, add STN constraints between the global predecessor and each activity in $affected$ so as to enforce that they can only start at the time they are currently allocated or later.

4. For each activity A in $affected$, acquire the connected component the activity A belongs to, and for all activities in all acquired connected components compute the values from which the allocation of the activity in the last step will begin ($= MinStart$), which is the maximum of (i) its current start time and (ii) its minimal distance from the global predecessor resulting from the STN.

5. Deallocate all activities in all connected components acquired in step 4.

6. Take the leftmost (according to the $MinStart$ values) non-allocated component C and allocate all activities in C starting with its leftmost activity using conflict-directed backjumping with backmarking. The activities within a connected component are allocated in the increasing order of their $MinStart$ values. Repeat this step until all connected components are allocated.

The skeleton of the algorithm is depicted in algorithm 5.

5.3.2 Swapping Resource Selections

In the first step, the algorithm goes through all activities in the model and checks whether the activity is scheduled to be processed on the forbidden resource. In the positive case, the function `SwapForbiddenSelection(Activity A)` changes resource selection of activity A to some allowed resource.

It is not important which resource is selected because the activity is most likely going to be reallocated in the later steps. Nevertheless, the algorithm picks the resource with the lowest usage, which is the sum of the durations of the activities that are allocated to the resource in question.

Formally, let us first denote the set of activities that use resource R as such.

Algorithm 5 STN-Recovery

```
Require: The STN with the APSP property
  function STN-RECOVERY
      for all $A \in Activities$ do
          if $SelectedResource(A) = ForbiddenResource$ then
              SWAPFORBIDDENSELECTION($A$)
          end if
      end for

      $affected \leftarrow$ SHIFTONRESOURCES

      for all $A_i \in affected$ do
          IFPC($i$, 0, $-Start(A_i)$)
      end for

      $components \leftarrow$ ACQUIRECOMPONENTS($affected$)

      DEALLOCATECOMPONENTS($components$)

      while $components \neq \emptyset$ do
          $C \leftarrow$ GETLEFTMOSTCOMPONENT($components$)
          ALLOCATECOMPONENT($C$)
          $components \leftarrow components \setminus \{C\}$
      end while
  end function
```

$$\mathit{ResourceActivities}(R) = \{A \in \mathit{Activities} \mid \mathit{SelectedResource}(A) = R\}$$

The usage of resource R can be written as follows.

$$Usage(R) = \sum_{A \in \mathit{ResourceActivities}(R)} \mathit{Duration}(A)$$

Then picking the resource with the lowest usage means this:

$$\mathit{SelectedResource}(A) = \arg \min_{R \in \mathit{Resources}(A)} (Usage(R))$$

At this time being, some activities may violate resource constraints.

5.3.3 Shifting Activities

In the second step, the algorithm repairs the violated resource constraints. It visits the resources one after another and shifts activities that overlap to the right. Since the original schedule is assumed to have been feasible, only the resources where some activities were added should be revised.

Procedure `ShiftOnResources` sweeps over the activities and conducts the shifting as follows. If activity A_0 overlaps activity A_1 on a resource, the activity with the later start time, say A_1, is set its start time to the end time of A_0. This shift may cause activity A_1 to overlap next activity, which is then set to start at the end of activity A_1 and so forth. The order of activities on the resource is preserved. All activities from the first activity that has been shifted up to the last activity (in terms of start times), even if some have not been shifted, are added to the set $affected$.

Formally, let $\mathit{begin}(R)$ be the start time of the first (earliest) activity that overlaps with another activity on resource R.

$$\begin{aligned} \mathit{begin}(R) \leftarrow & \min_{A \in \mathit{ResourceActivities}(R)} \{\mathit{Start}(A) \\ & \mid \exists B \in \mathit{ResourceActivities}(R), B \neq A, \mathit{Start}(A) \leq \mathit{Start}(B) < \mathit{End}(A)\} \end{aligned}$$

Further, let us denote R^i the i-th earliest activity allocated on resource R, which means that the following holds.

$$1 \leq i < j \leq |\mathit{ResourceActivities}(R)| \Rightarrow \mathit{Start}(R^i) \leq \mathit{Start}(R^j)$$

The activities on resource R are consecutively (from the leftmost activity) shifted such that:

$$\mathit{Start}(R^i) \leftarrow \max\{\mathit{Start}(R^i), \mathit{End}(R^{i-1})\}$$

Finally, the activities are added to the set $affected$ as follows.

$$affected \leftarrow \{A \in \mathit{Activities} \mid \mathit{Start}(A) > \mathit{begin}(\mathit{SelectedResource}(A))\}$$

Algorithm 6 Shifting activities on resources

```
function ShiftOnResources
    for all $R \in Resources$ do
        $begin(R) \leftarrow \min_{A \in ResourceActivities(R)}\{Start(A)$
  $\mid \exists B \in ResourceActivities(R), B \neq A, Start(A) \leq Start(B) < End(A)\}$
        for $i \leftarrow 2$ to $|ResourceActivities(R)|$ do
            $Start(R^i) \leftarrow \max\{Start(R^i), End(R^{i-1})\}$
        end for
    end for
    Return $\{A \in Activities(A) \mid Start(A) > begin(SelectedResource(A))\}$
end function
```

The entire routine is depicted in algorithm 6.

This shifting may violate a large number of temporal constraints. The activities in the set $affected$ are going to be reallocated in the forthcoming steps. The reason why the set $affected$ includes the activities that have not been shifted, but are allocated on the right of the shifted activities, is, that they would otherwise preclude other activities from allocation.

5.3.4 Updating STN

In this step, the constraints determining the minimal distance of an activity from the global predecessor are added to the STN so as to modify the $MinStart$ values of activities to be reallocated, according to the start time values set in the previous shifting step. The IFPC algorithm is used because modifying the minimal start time of an activity affects the minimal start times of other activities from the same connected component.

Precisely, for each $A_i \in affected$, add to the STN via IFPC algorithm the constraint $(A_i, A_0, -Start(A_i))$, where A_0 denotes the global predecessor.

The point of adding this constraints is to reasonably maintain similarity to the original schedule, along with adequate pruning of the search space of the upcoming reallocation process.

5.3.5 Components Acquirement

There is still a question which and in what order the activities should be reallocated. Because shifting one activity is likely to violate temporal constraints emanating from or to the activity, it is necessary to reallocate the entire connected component. Therefore, procedure `AcquireComponents(affected)` acquires the connected component that each activity $A \in affected$ belongs to, and the acquired connected component is added to the set *components*. After this step, $components = \{C_1, ...C_k\}$, where C_z for $z = 1, ..., k$ is a connected component.

In addition, for each activity, the $MinStart$ value, which is the maximum of the current start time and of the minimal potential start time following from the STN (computed via IFPC in the previous step), is computed. Precisely, for each $C_z \in components$ and for each $A_i \in C_z$, assign:

$$MinStart(A_i) = Max\{Start(A_i), -w[i, 0]\}$$

As to the order for upcoming allocation, it is suitable to allocate activities in the increasing order of the $MinStart$ values. The activity in a connected component with the lowest $MinStart$ value is referred to as the leftmost activity. The leftmost connected component is the connected component of which the leftmost activity has the lowest $MinStart$ value among all connected components. The algorithm always selects for allocation the leftmost component that has not yet been allocated.

5.3.6 Deallocation

Since the best way for allocating activities turned out to be the way without violating resource constraints, it is necessary to deallocate all activities in the connected components acquired in the previous step. Otherwise they would preclude other activities from allocation. Procedure `DeallocateComponent(components)` deallocates activities from each connected component $C \in components$, which means that for each $A \in C$: $Start(A) = null$ and $SelectedResource(A) = null$. After this (fifth) step, all activities from *components* are deallocated.

5.3.7 Allocation

Allocating an activity again means searching for the time point when there is an available resource for the required duration. The resources are selected according to the ESSLPE rule described in 5.2.1.

The skeleton of the code for allocating a connected component (see algorithm 7) is constituted from the algorithm 1 which is explained in section 3.1.3.

In order to allocate a connected component, conflict-directed backjumping with backmarking is used. When an activity cannot be successfully allocated, it is necessary to jump back to the activity that is causing the conflict. This is the reason why conflict-directed backjumping is used. For keeping the information which activity is conflicting with the activity being allocated, the conflict set for each activity is remembered. For this purpose, $cs[i]$ is a set of activities conflicting with A_i.

The activities are going to be allocated in the increasing order of their indexes that are determined according to their $MinStart$ values. Thus we can anticipate that the connected component to be allocated, which is passed as a parameter, consists of activities $A_1, ..., A_n$. When two activities are compared, i.e. $A_j < A_i$, it means that their indexes are compared ($j < i$).

There are two possible causes why an activity cannot be allocated: a temporal conflict and a resource conflict.

Temporal conflicts

Temporal conflicts are handled in procedure `UpdateBounds(Activity A)` (see algorithm 8), which is called at line 6, i.e. before activity A_i is going to be allocated. In this procedure, the bounds of possible time allocation for activity A_i are computed according to the STN and start times of already allocated activities.

The lower bound of an activity is initially set to the $MinStart$ value acquired in the previous steps. Then the procedure goes through the already allocated activities within the connected component in the same order as they have been

Algorithm 7 Allocating an entire connected component

```
 1: function AllocateComponent(Activities $A_1, ..., A_n$)
 2:     $i \leftarrow 1$
 3:     while $i \leq n$ do
 4:         $newVal \leftarrow newVals[i]$                                  ▷ initially 0
 5:         if $newVal = 0$ then                        ▷ tracking forward to new activity
 6:             UpdateBounds($A_i$)
 7:             $newVal \leftarrow LowerBound(A_i)$
 8:         end if
 9:         while $SelectedResource(A_i) = null$ & $newVal \leq UpperBound(A_i)$ do
10:             if $newVal \in Keys(Mark[i])$ & $Max(Mark[i][newVal]) < BackTo[i][newVal]$ then
11:                 $cs[i] \leftarrow cs[i] \cup Mark[i][newVal]$
12:                 $newVal \leftarrow newVal + 1$
13:                 continue
14:             end if
15:             $BackTo[i][newVal] \leftarrow A_i$
16:             $newConflicts \leftarrow \emptyset$
17:             for all $R \in Resources(A_i)$ do
18:                 $newConflicts \leftarrow newConflicts \cup Min^*(Impedimentary(A_i, R, newVal))$
19:             end for
20:             if $CandidateResources(A_i, newVal) \neq \emptyset$ then
21:                 $SelectedResource(A_i) \leftarrow$ arbitrary from $CandidateResources(A_i, newVal)$
22:                 $Start(A_i) \leftarrow newVal$
23:                                                     ▷ $newVal$ can be tried again
24:                 $Keys(Mark[i]) \leftarrow Keys(Mark[i]) \setminus \{newVal\}$
25:             else
26:                 $Keys(Mark[i]) \leftarrow Keys(Mark[i]) \cup \{newVal\}$
27:                 $Mark[i][newVal] \leftarrow newConflicts$
28:             end if
29:             $cs[i] \leftarrow cs[i] \cup newConflicts$
30:             $newVal \leftarrow newVal + 1$
31:         end while
```

```
        if $SelectedResource(A_i) = null$ then
            $A_j \leftarrow Max(cs[i])$            ▷ latest activity in $cs[i]$ is where to jump
            $cs[j] \leftarrow cs[j] \cup cs[i] \setminus \{A_j\}$
            for $k \leftarrow j + 1$ to $n$ do
                for all $key \in Keys(BackTo[k])$ do
                    $BackTo[k][key] \leftarrow Min(BackTo[k][key], A_j)$
                end for
            end for
            while $i > j$ do                                    ▷ jump back to $j$
                $newVals[i] \leftarrow 0$
                $i \leftarrow i - 1$
                $SelectedResource(A_i) \leftarrow null$
                $Start(A_i) \leftarrow null$
            end while
        else
            $newVals[i] \leftarrow newVal$
            $i \leftarrow i + 1$
        end if
    end while
end function
```

Algorithm 8 Updating lower and upper bounds

```
function UpdateBounds(Activitiy $A_i$)
    $cs[i] \leftarrow \emptyset$                                        ▷ clear conflict set
    $LowerBound(A_i) \leftarrow MinStart(A_i)$
    $UpperBound(A_i) \leftarrow \infty$
    for $k \leftarrow 1$ to $i - 1$ do
        $newValue \leftarrow Start(A_k) - w[i, k]$
        if $LowerBound(A_i) < newValue$ then
            $LowerBound(A_i) \leftarrow newValue$
            $cs[i] \leftarrow cs[i] \cup \{A_k\}$
        end if
        $newValue \leftarrow Start(A_k) + w[k, i]$
        if $UpperBound(A_i) > newValue$ then
            $UpperBound(A_i) \leftarrow newValue$
            $cs[i] \leftarrow cs[i] \cup \{A_k\}$
        end if
    end for
end function
```

allocated and updates bounds of A_i. Precisely, for each $k < i$, if $Start(A_k)$ + "minimal distance from $Start(A_k)$ to $Start(A_i)$" is greater than the current lower bound, then increase the lower bound, and add A_k to the conflict set of A_i. Similarly, if $Start(A_k)$ + "maximal distance from $Start(A_k)$ to $Start(A_i)$" is smaller than the current upper bound, then decrease the upper bound, and add A_k to the conflict set of A_i. The reason why activity A_k is added to the conflict set is that changing the start time of A_k creates (straight away or after a number of steps) some new possible start time for A_i.

Resource conflicts

As far as resource conflicts are concerned, recall that $Impedimentary(A_i, R, t)$, which has been formally introduced in section 5.2.1, is a set of activities that preclude activity A_i from selecting resource R at time t. To make it possible to allocate activity A_i on resource R at time t, all activities from the set $Impedimentary(A_i, R, t)$ would have to be reallocated. The question is, which activity from $Impedimentary(A_i, R, t)$ should be added to the conflict set of activity A_i. The answer is, the activity that has been the least recently allocated (from the connected component being allocated), but if there is an activity in $Impedimentary(A_i, R, t)$ from another connected component, which means it cannot be deallocated, then no activity is added to the conflict set.

This is exactly what Min^* does (at line 18). Formally, let C be the connected component being allocated. If $Impedimentary(A_i, R, t) \subseteq C$, then:

$$Min^*(Impedimentary(A_i, R, t)) = \arg \min_{A_k \in Impedimentary(A_i,R,t)} \{k\}$$

Otherwise $Min^*(Impedimentary(A_i, R, t)) = \emptyset$.

For illustration, when the algorithm is allocating activity A_7 and there are activities A_2, A_4, and A_6 inhibiting on a resource, then activity A_2 is added to the conflict set. If there is an activity from different, already allocated component, then no activity is added to the conflict set.

Further, recall $CandidateResources(A_i, t)$ is a subset of available resources given by ESSLPE rule, and from which activity A_i may arbitrarily select one. Regardless of the result of the search for an available resource, the conflicting activities are merged into the conflict set of the activity being allocated, i.e. $cs[i]$ (line 29).

Backjump

When the algorithm is about to conduct a backjump (starting at line 32), which happens when all possible start times of A_i have been tried, the most recently allocated activity from the conflict set of A_i is found (line 33). Let us denote this activity as A_j. Next, before deallocating activities that are jumped over, the activities from the conflict set of A_i except activity A_j are added to the conflict set of A_j.

Backmarking

The backmarking technique is implemented as follows. Firstly, the time horizon is infinite so that the structures *BackTo* and *Mark* cannot be simple two-dimensional arrays as described in 3.1.3 but arrays of dictionaries. Precisely, $BackTo$ is an array of size n, $BackTo[i]$ is a dictionary, where keys are the (attempted) start times of the activity, and values are activities, i.e., $BackTo[i][newVal]$ is the lowest-indexed activity whose instantiation has changed since activity A_i was last tried to be allocated at time $newVal$.

As to the structure $Mark$, there is one difference. Notice that when the algorithm cannot find an available resource for activity A_i at time $newVal$, not only one, but a number of activities may be added to the conflict set of A_i. Consequently $Mark[i][newVal]$ is a set of activities, of which at least one must be reallocated in order to make activity A_i allocatable at time $newVal$. Therefore, when values $BackTo$ and $Mark$ are to be compared, it is firstly checked, whether there is $newVal$ among the keys of $Mark[i]$, and in the positive case, $Max(Mark[i][newVal])$ and $BackTo[i][newVal]$ are compared (see line 10).

If $Max(Mark[i][newVal]) < BackTo[i][newVal]$, it means that none of the conflicting activities has been re-instantiated and thus it makes no sense to look for an available resource. However, before proceeding to the next value of $newVal$, it is necessary to merge the conflicting activities to the current conflict set (line 11) as if the search for an available resource was conducted – this is the reason why $Mark[i][newVal]$ must store the set of activities (and not just the most recent activity).

Oppositely, if $newVal$ is not presented among the keys of $Mark[i]$ or $Max(Mark[i][newVal]) \geq BackTo[i][newVal]$, the algorithm does look for an available resource. If activity A_i is successfully allocated, the key $newVal$ is removed from $Mark[i]$ (line 24), otherwise $Mark[i][newVal]$ stores the conflicting activities (line 27). Using this concept of allocation, we have not found an efficient way to achieve type B saving.

Termination

Notice that the algorithm does not check for the recoverability of the disrupted ongoing schedule, which means that if there is no feasible solution, the procedure `AllocateComponent(Component C)` never terminates. This can be solved by giving it a limited time (cut-off limit), or by detecting that the method got stuck in a loop, which may be proven for example when it tries to allocate an activity in time greater than the maximal estimate of makespan (which may be the sum of the durations of all activities and of all minimal distances in the model).

5.3.8 Correctness

If a resource selection change at the first step causes a resource constraint violation, at least one of the activities involved in the conflict is shifted in time. All activities that are shifted at the second step are then reallocated along with the entire connected component they belong to. During the allocation process, the activity is always allocated to such a time point that neither resource constraints, nor temporal constraints are violated. Therefore, if the algorithm finds a schedule, it is feasible and hence the algorithm is sound.

As to completeness, the only nontrivial thing to realize is that, provided a schedule is recoverable, the process of allocating a component, i.e.

`AllocateComponent(Component C)`, terminates and finds the solution. Here comes an informal argumentation.

To the conflict set of an activity A_i, i.e. $cs[i]$, there is always added such an activity A_j, that A_j, having set its start time and having selected resource, participates in the fact that A_i cannot be allocated. It follows that reallocating activity A_j modifies (immediately or later) the domain of possible start times (in case of temporal conflicts) or available resources (in case of resource conflicts) of activity A_i.

When seeking for an available resource is avoided thanks to the backmarking technique, that is because it is known the activity in question could not be really allocated, and the conflict set is updated as if the search for an available resource was conducted.

Further, when an activity A_i cannot be allocated, i.e. all possible start times have turned out not to lead to solution, the backjump is conducted to the most recently allocated activity that is contained in the conflict set of A_i and the conflict sets are merged, so that no information about conflicts is lost and therefore all jumps are safe, which means that no solution is overleaped. Hence, if there exists a feasible solution, the algorithm will find it.

Unfortunately, the formal proof is yet to be done.

5.3.9 Discussion and Experimental Evaluation

Owing to the nature of the algorithm, it moves a lot of activities by a small amount of time, which means that it should not be used when optimising in distance function f_2 (which is the number of shifted activities) as depicted in figure 5.6. On the other hand, it performs very well in function f_3 (which is the biggest shift of an activity) as shown in picture 5.7, and also, when compared to the Right Shift Affected algorithm, slightly better in function f_1. For more results refer to appendix B.2.

The Right Shift Affected algorithm is somewhat faster than STN-recovery (figure 5.8), however, STN-recovery has the following advantage. The algorithm always allocates the leftmost connected component that has not been allocated yet, therefore, when the algorithm is allocating the connected component with the leftmost activity that has the $MinStart$ value t, the schedule is not going to be modified before time point t. This allows the system to keep executing an ongoing schedule even if it has not been completely recovered yet.

The dependencies on the density of constraints showed no tendency. However, one might wonder how the algorithm performs as the size of connected components increases. As depicted in figure 5.9, there are significantly longer run-times for some sizes, however, exponential growth is not apparent.

As far as the backmarking technique is concerned, it brought some saving of time as expected, because determining availability of a resource is carried out in logarithmic time in the number of activities on the resource. On the other hand, as the number of resources in the model decreases below a certain number, one might expect backmarking to become counterproductive owing to the overhead costs. However, according to the figure 5.10, backmarking pays regardless of the number of resources in the model.

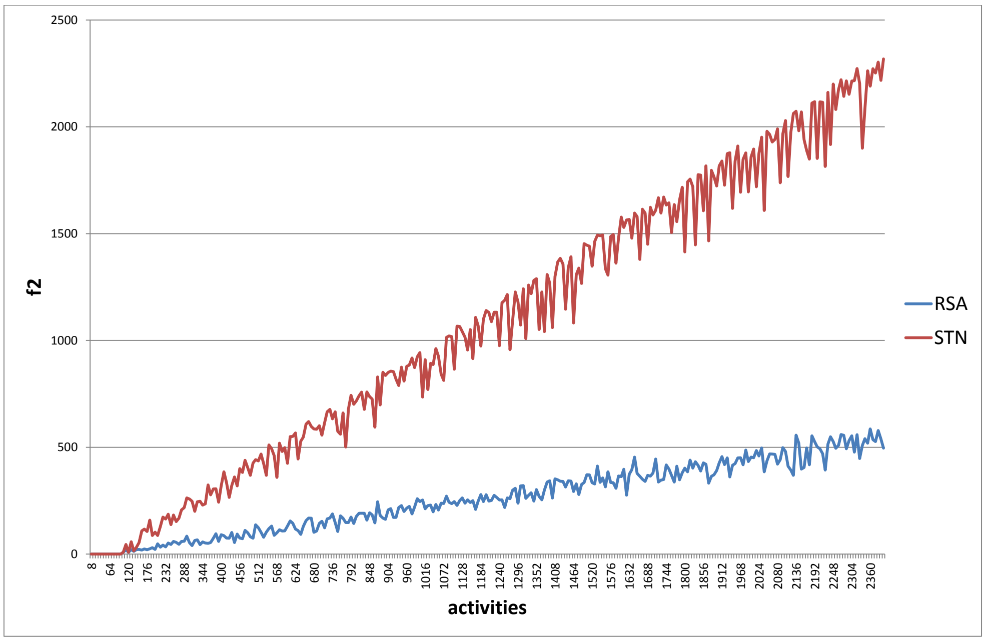

Figure 5.6: Comparison of schedule quality (measured using function f2) for Right Shift Affected and STN-recovery

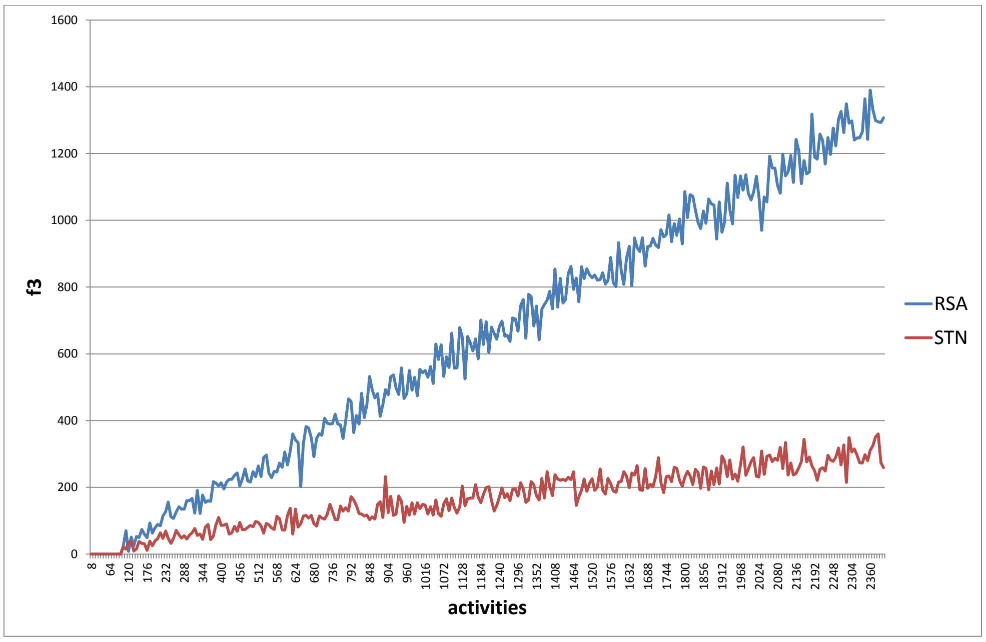

Figure 5.7: Comparison of schedule quality (measured using function f3) for Right Shift Affected and STN-recovery

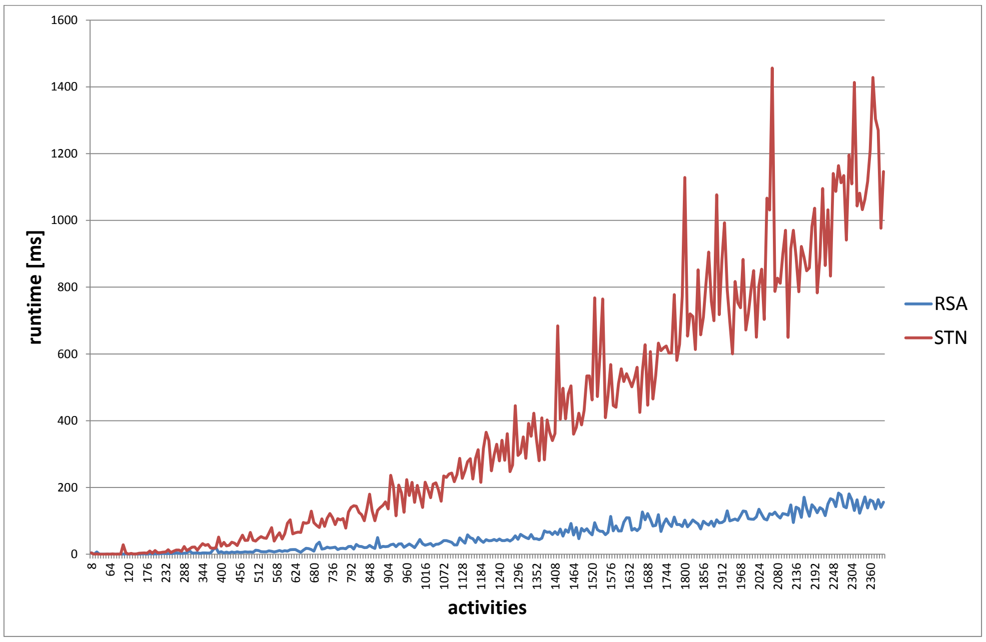

Figure 5.8: Comparison of run times for Right Shift Affected and STN-recovery

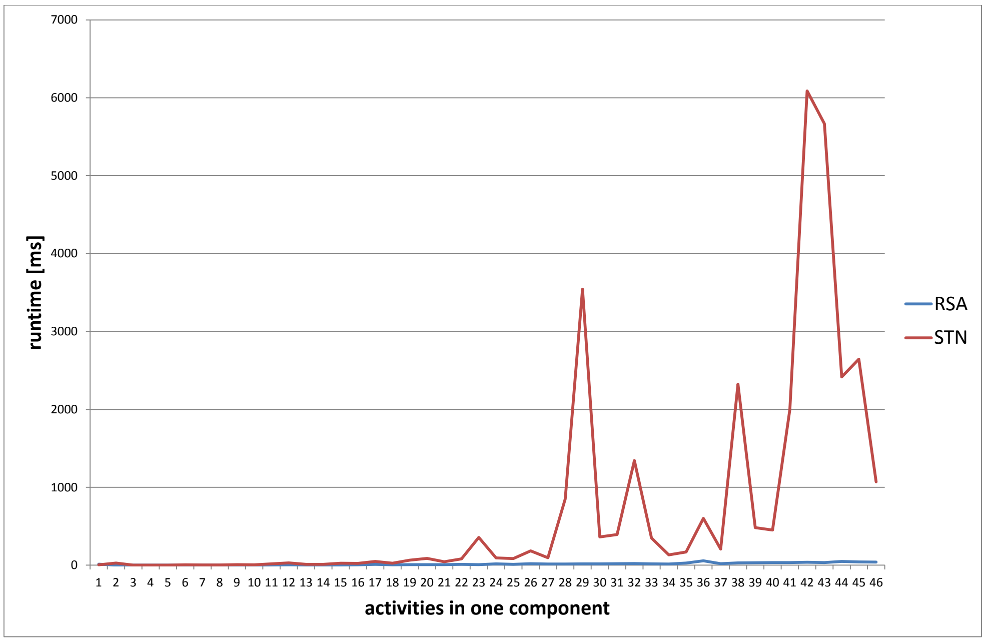

Figure 5.9: Comparison of run times for Right Shift Affected and STN-recovery, dependent on the number of activities in one component

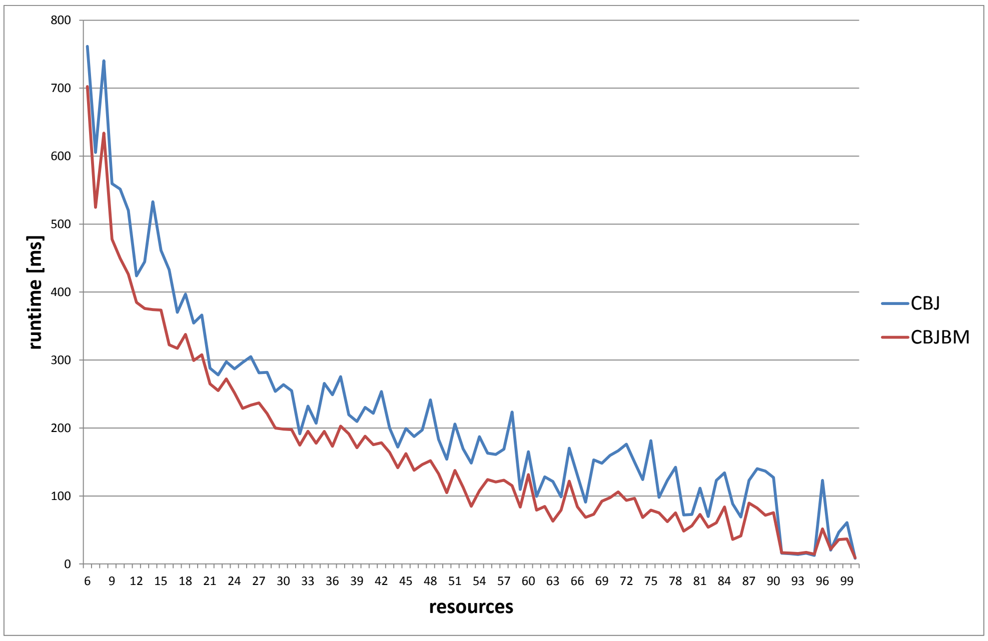

Figure 5.10: Comparison of run times for conflict-directed backjumping and conflict-directed backjumping with backmarking

5.4 Breakdown with Arbitrary Resource Groups

So far, the algorithms anticipated that the resource groups are equivalent. In that case, the algorithms use the ESSLPE rule described in 5.2.1. If the resource groups are not equivalent, the ESSLPE rule cannot be used because selecting one resource may not lead to finding a feasible solution, while selecting some other resource may lead to recovery. Here comes an example.

Consider the following model with two available resources, $Resources = \{R_1, R_2\}$, and three activities, $Activities = \{A, B, C\}$. The attributes of the activities are given in the following table.

	A	B	C
Resources	$\{R_1\}$	$\{R_1, R_2\}$	$\{R_1\}$
Duration	1	2	1

There is the end-to-start synchronization between A and B, and end-to-end synchronization between B and C, which involves the following constraints.

- $(A, B, 1)$
- $(B, A, -1)$
- $(B, C, 1)$

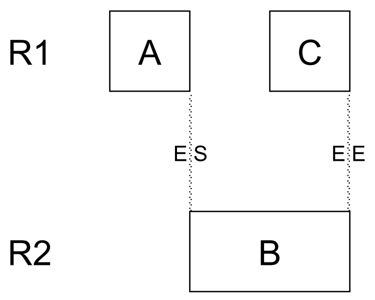

Figure 5.11: Feasible schedule for the model

- $(C, B, -1)$

It is easy to see that there exists the feasible schedule given by the following table and depicted in figure 5.11.

	A	B	C
Start	0	1	2
SelectedResource	R_1	R_2	R_1

However, suppose STN-recovery is about to allocate this component, i.e. `AllocateComponent(A, B, C)` is called. From the STN we get the following $MinStart$ values:

	A	B	C
MinStart	0	1	2

Hence the order of activities for allocation is indeed $[A, B, C]$. When the algorithm is allocating activity B, the ESSLPE rule always picks resource R_1, because it is non-empty, and then activity C cannot be allocated, therefore the algorithm sticks in a loop, while selecting R_2 would yield the feasible schedule at a blow.

Consequently, to accommodate with arbitrary resource groups, the suggested algorithms require some changes.

Right Shift Affected

As to the Right Shift Affected algorithm, the only necessary modification is, when allocating an activity, select resource from the group uniformly at random instead of according to the ESSLPE rule.

STN-Recovery

As far as STN-recovery is concerned, the required changes are somewhat more involved. When the algorithm backtracks from activity A_i to activity A_j, then before trying to reallocate A_j to another time point, it may be necessary to try other resource selections first. It is handful to pass the reason for tracking back from A_i to A_j. That is why the following surgery is required.

So far, a conflict set was simply a set of conflicting activities. Now, the conflict set will be a set of conflicts, where *conflict* is, instead of being just an activity in conflict, a pair of (i) the conflicting activity and (ii) a flag indicating the reason we are jumping back: temporal or resource. More precisely, $Conflict$ is a structure containing two fields: *Activity*, and *Type*. If the conflict comes from a temporal constraint, *Type* is set to 0. When the conflict comes from a resource constraint, *Type* is set to 1.

When the algorithm determines where to jump, it searched for the most recently allocated activity in the conflict set by comparing their indexes. Therefore, it is now handful to redefine comparison operators. A resource constraint violation has the priority to temporal constraint violation, which means that a resource conflict between activities A_j and A_i is understood as more recent when compared to a temporal conflict between activities A_j and A_i. Formally, when two conflicts, say C_1 and C_2, are compared, then the result is defined as follows.

$<$

```
if $C_1.Activity = C_2.Activity$ then
    return $C_1.Type < C_2.Type$
else
    return $C_1.Activity < C_2.Activity$
end if
```

$\leq$

```
if $C_1.Activity = C_2.Activity$ then
    return $C_1.Type \leq C_2.Type$
else
    return $C_1.Activity \leq C_2.Activity$
end if
```

Operators $>$ and $\geq$ are defined analogously.

Finally, when we backtrack from activity A_i to A_j owing to a temporal conflict, A_j is deallocated, the value for time allocation (= $newVal$) is increased and A_j is tried to allocate at (increased) $newVal$. When we backtrack owing to a resource conflict, $newVal$ remains unchanged and another resource from $AvailableResources(A_j, newVal)$ is selected. If all selections have been tried, the algorithm proceeds as if we have just backtracked to A_j due to a temporal conflict, $newVal$ is increased, and so on.

5.5 Breakdown at time $t_f > 0$

Up to now, we assumed that a machine breakdown occurs at the beginning of the schedule horizon. This section describes one possible way to cope with the problem when resource R_f fails at time point $t_f > 0$. As mentioned at the very beginning of this chapter, this brings further questions such as what happens to the activity that is being processed at time t_f on resource R_f, and so on.

Let us define that activity A is *completed* if $End(A) \leq t_f$, A is *in progress* if $Start(A) < t_f$ and $End(A) > t_f$, and A is *untouched* if $Start(A) \geq t_f$.

We have decided to implement the algorithms to tackle this problem as follows. If activity A allocated to resource R_f is in progress, it must be processed again from the beginning as if it has not been processed at all, i.e. for the time span that equals $Duration(A)$. Activities belonging to connected component C that are completed or in progress (and not using R_f) do not have to be re-executed even if there is no reallocation of other activities belonging to C such that all constraints are satisfied. This means that violation of constraints, determining the maximal time distance between an activity that is completed or in progress and an activity that is untouched, is tolerated.

STN-Recovery

This solution required to modify the implementation of STN-recovery as follows.

- The forbidden resource selections are changed only for activities in progress and untouched activities, i.e. for those of which $End(A) > t_f$.
- Activities that are completed and activities in progress that are not allocated to R_f are set to be *pinned*, i.e. $Pinned(A) \leftarrow true$, which means that these activities are not going to be deallocated and violations of temporal constraints emanating from them are going to be tolerated.
- Before the shifting procedure, activities in progress that were allocated to R_f are shifted to start at the time t_f, i.e. $Start(A) = t_f$.
- In the deallocation process, activities that are pinned are not deallocated.
- In the `AllocateComponent` procedure, when variable i points to a pinned activity, it proceeds to the next one (i.e. $i + 1$).
- In the `UpdateBounds` procedure, the pinned activities are not allowed to modify the upper bound value, and thus are not added to the conflict set.
- Finally, if the set $Impedimentary(A, R, t)$ contains an activity that is pinned, none is added to the conflict set.

Right Shift Affected

The Right Shift Affected algorithm required the following changes.

- At the beginning, only activities in progress and untouched activities, i.e. those of which $End(A) > t_f$, are checked for using R_f, and reallocated in the positive case.
- Activities that are completed and activities in progress that are not allocated to R_f are set to be pinned, i.e. $Pinned(A) = true$.
- Constraint $(A_1, A_2, w) \in Constraints$ is repaired (i.e. one of the activities is reallocated), only if $Pinned(A_1) = Pinned(A_2)$.

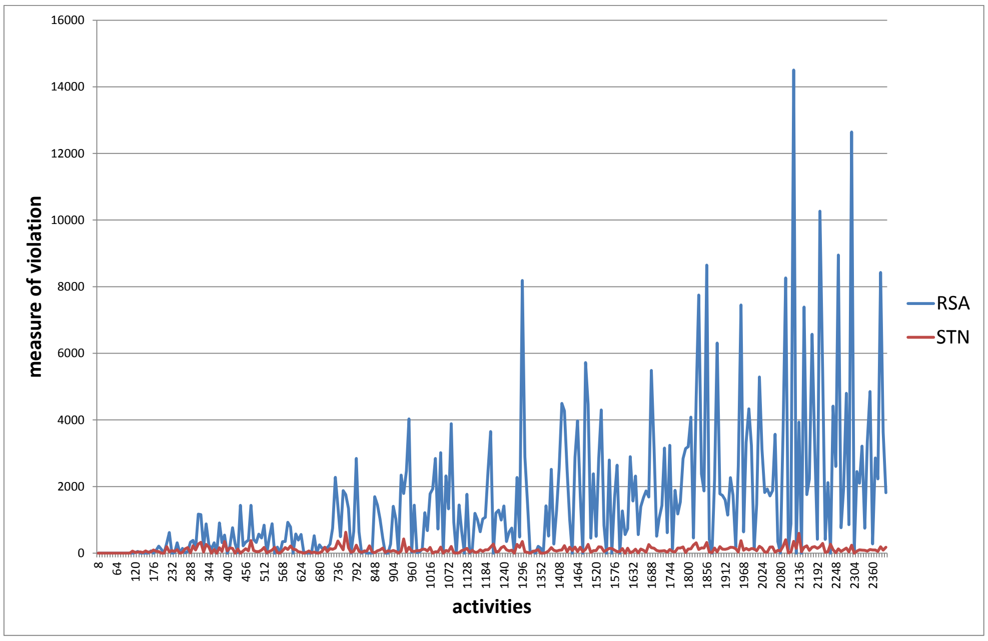

Figure 5.12: Comparison of schedule quality (measured in constraints violation) for Right Shift Affected and STN-recovery

Constraint Violation

In spite of the described modifications, the results regarding a machine breakdown at time $t_f = 0$ hold also for a machine breakdown occurring for example at time $t_f = 30$. However, since we allowed violating some constraints, it is suitable to compare the algorithms in terms of *measure of violation*, which is computed this way. For each temporal constraint $(A_1, A_2, w) \in Constraints$ do the following:

- If $Start(A_2) - Start(A_1) = b$, and $b > w$, then increase the *measure of violation* by $b - w$.

The results, as depicted in figure 5.12, show that the Right Shift Affected algorithm is rather useless, whereas the measure of violation in STN-Recovery can be, despite non-negligible fluctuation, regarded as constant for models large enough, which is expected because the violated constraints occur only nearby time t_f.

6. Other Disturbances

6.1 Hot Order Arrival

Next to the machine breakdown, one of the most common disturbances bothering manufacturing processes is an urgent order arrival, also called a hot order arrival. When an urgent order comes during the schedule execution, say at time t_h, the task is to correct the ongoing schedule in such a way that the activities of the urgent order are scheduled after time t_h the earlier the better, to the detriment of the other activities, which consequently have to be postponed.

To tackle this problem, it is suitable to exploit the STN-recovery algorithm described in the previous chapter. Initially, among a few technical adjustments, it is important to extend the STN to accommodate the activities of the urgent order newly added to the scheduling model, and introduce the temporal constraints among the activities to the STN through the IFPC algorithm.

The only necessary modification of STN-recovery is the first step. Instead of changing resource selections from forbidden resource, do for each activity A_i in the newly added order the following: From $Resources(A_i)$ select the resource with the lowest usage and set $Start(A_i)$ to the current time (when the order arrives) plus minimal time distance of activity A_i from the global predecessor obtained from the STN, i.e. $Start(A_i) = t_h - w[i, 0]$. Now the activities of the urgent order overlap some activities of the original schedule on chosen resources. Afterwards, STN-recovery continues without any change.

Note that the way of tackling a resource failure at a non-zero time as described in 5.5 leads to some constraint violation, which holds in the case of hot order arrival too. It may be beneficial, at least for less urgent order arrivals, to avoid the violation completely. This may be simply achieved by pinning the entire connected components that are in progress.

The Right Shift Affected algorithm could be used too, with the only one modification. At the beginning, instead of reallocating activities using forbidden resource, the activities in the urgent order are allocated. Then the algorithm proceeds the same way. Unfortunately, the higher the usage of resources, the later in time the urgent order is likely to be accomplished, making thus the algorithm useless if the order is hot indeed.

6.2 Order Cancellation

An order cancellation, which is actually the counterpart of the hot order arrival, is another event that often occurs in manufacturing environments. The order cancellation is usually ranked among so called minor disturbances, because simple invalidating the cancelled activities along with letting empty holes on resources at the places where the cancelled activities were allocated preserves feasibility of the schedule, so that no modification is needed. However, when the objective function is for example (total) lateness or completeness, it may be desired to somehow exploit newly enabled spaces on resources.

When the order cancellation occurs, the intention is to improve the schedule efficiency without significant schedule modification. The idea is to shift some activities in time to the left without changing any resource selection and without changing the processing order of activities on resources.

This may be achieved very simply through using the STN. As in the previous chapters, the STN based on the temporal constraints is assumed to have been computed. But now, we need to add the constraints into the STN so as to enforce that the order of activities remains unchanged. For each two consecutive activities on a resource, say A_i and A_j, add a constraint into the STN ensuring that A_j cannot start sooner than A_i finishes, i.e. $Start(A_j) - Start(A_i) \geq Duration(A_i)$, and propagate it via the IFPC algorithm. Then set the start times of all activities to their minimal possible values.

Formally, let us define the set *Consecutives* containing pairs of activities that are in a row on a resource as follows.

$$\begin{aligned} Consecutives = \{(A_i, A_j) \mid A_i, A_j \in Activities \\ \wedge\ SelectedResource(A_i) = SelectedResource(A_j) \wedge End(A_i) \leq Start(A_j) \\ \wedge\ (\nexists A_k \in Activities : SelectedResource(A_i) = SelectedResource(A_k) \\ \wedge\ End(A_i) \leq Start(A_k) \leq Start(A_j))\} \end{aligned} \quad (6.1)$$

What the algorithm exactly does is the following. For each $(A_i, A_j) \in Consecutives$ add to the STN the constraint $(A_j, A_i, -Duration(A_i))$ via IFPC, and, when done, for each activity A_i set its start time to the minimal possible distance from the global predecessor obtained from the STN. The entire squeezing algorithm is depicted in algorithm 9.

Algorithm 9 The Squeezing Algorithm

```
function SQUEEZE
    for all $(A_i, A_j) \in Consecutives$ do
        IFPC($j$, $i$, $-Duration(A_i)$)
    end for
    for all $A_i \in Activities$ do
        $Start(A_i) \leftarrow -w[i, 0]$
    end for
end function
```

6.3 Machine Repair

Another event that may be of interest in dynamic environments is introduction of a new resource. Such a significant change in the model definition usually requires special attention and is conducted when no schedule is executed or during a maintenance-break. However, it often happens that a machine breaks down temporarily, so that when it is fixed, it is desired to exploit the machine in order to reduce the objective function without substantial schedule modification. For this purpose, we suggest a very simple method that reallocates some activities to the repaired machine and then runs the squeezing algorithm described in the previous section.

Let us denote the resource newly added to the model as R_{new}. Further, let us assume that, for each activity $A \in \mathit{Activities}$, if A can be processed on the resource R_{new}, then R_{new} is added to the resource group of A, i.e. $\mathit{Resources}(A) = \mathit{Resources}(A) \cup R_{new}$.

Then the reallocation of activities is carried out as follows. It seeks for the earliest activity of which a selected resource can be swapped to R_{new}. Formally:

$$A_{r_1} = \arg \min_{A' \in \mathit{Activities}} \{\mathit{Start}(A') \mid R_{new} \in \mathit{Resources}(A')\}$$

Activity A_{r_1} is then reallocated to R_{new}, i.e. $\mathit{Selected}(A_{r_1}) = R_{new}$. The next activity to be reallocated to R_{new} must start at $\mathit{End}(A_{r_1})$ or later so that it does not overlap activity A_{r_1}. Formally, let us define $NextAllowedStart$ as the minimal allowed start time of the next activity to be reallocated. Suppose that i activities has been reallocated to R_{new}. Then $NextAllowedStart$ may be assigned as such:

$$NextAllowedStart = \mathit{End}(A_{r_i})$$

Next activity to be reallocated to R_{new} is then:

$$A_{r_{i+1}} = \arg \min_{A' \in \mathit{Activities}} \{\mathit{Start}(A') \mid R_{new} \in \mathit{Resources}(A') \wedge \mathit{Start}(A') \geq NextAllowedStart\}$$

The question is how to suitably assign $NextAllowedStart$. If $NextAllowedStart$ is $\mathit{End}(A_{r_i})$ as just defined, then it probably does not lead to any schedule improvement. Consider a schedule with only one resource, say R_1, and that the usage of R_1 is 100 %, i.e. there are no gaps between any two consecutive activities. Now, the resource R_{new} is added, and, for each activity A, $\mathit{Resources}(A) = \{R_1, R_{new}\}$. In this case, setting $NextAllowedStart$ to $\mathit{End}(A_{r_i})$ causes that all activities are reallocated to R_{new} while R_1 becomes idle. Learnt from that extreme case, some free time before the next activity to be reallocated should be enforced.

FS One possible time point for $NextAllowedStart$ is the start time of the activity that follows activity A_{r_i} on the resource from which A_{r_i} is retracted (plus one time unit). Formally:

$$NextAllowedStart = \mathit{Start}(Following(A_{r_i})) + 1,$$

where $Following(A_{r_i})$ is the activity allocated in time just after A_{r_i} on the same resource, i.e. $(A_{r_i}, Following(A_{r_i})) \in \mathit{Consecutives}$. (See definition 6.1, in the previous section.) Thus it is ensured that the activity succeeding A_{r_i} on the resource is not reallocated. Let us refer to this heuristic as FS.

DR Another conceivable value for the time slack is the duration of activity A_{r_i} divided by the number of resources from its resource group. Formally:

$$NextAllowedStart = \mathit{End}(A_{r_i}) + \lfloor \mathit{Duration}(A_{r_i}) / \mathit{Resources}(A_{r_i}) \rfloor$$

This is what we refer to as DR.

Finally, after the activities have been reallocated to R_{new}, the squeeze method from section 6.2 is called.

Suppose all activities in the scheduling model are sorted in the increasing order of their start times. Then algorithm 10 depicts the heuristic DR.

These two heuristics may be compared using distance function f_1. Since the activities are moved only to the left, the function f_1 corresponds to the decrease in total completeness, which is now desired to be maximized. Therefore, the DR heuristic seems to outperform FS (see figure 6.1). However, this may be because of the random nature of the testing data, so that this result may no longer be valid for the real life schedules.

Algorithm 10 Repaired machine utilization

```
Require: Activities $A_1, ..., A_n$ sorted increasingly in their start times
  function UseRepairedResource(Resource $R_{new}$)
      $NextAllowedStart \leftarrow 0$
      for $i \leftarrow 1$ to $n$ do
          if $R_{new} \in Resources(A_i)$ & $Start(A_i) \geq NextAllowedStart$ then
              $NextAllowedStart \leftarrow End(A_i) + \lfloor Duration(A_i)/|Resources(A_i)| \rfloor$
              $SelectedResource(A_i) \leftarrow R_{new}$
          end if
      end for
      Squeeze
  end function
```

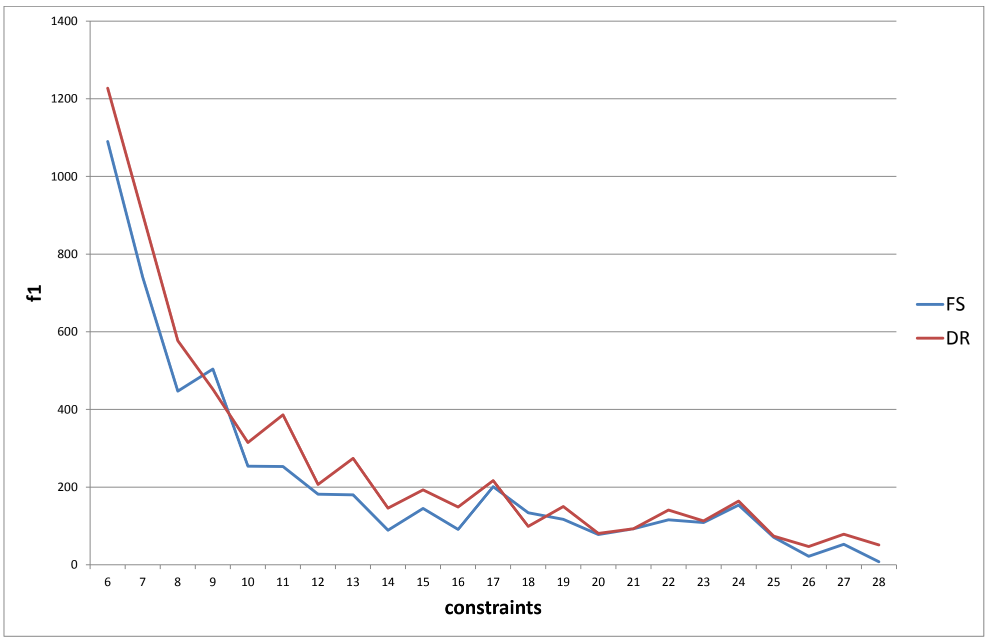

Figure 6.1: Comparison of schedule quality (measured using distance function f1) for reallocation heuristics

Conclusion

The aim of this thesis was to handle certain unexpected disturbances that may occur during a manufacturing schedule execution, which means finding quickly a feasible schedule as similar to the ongoing one as possible. The main focus has been taken on a machine breakdown, i.e. a disruption when a resource suddenly cannot be used anymore by any activity. We proposed two wholly different methods.

The first method takes the activities that were to be processed on a broken machine, reallocates them, and then it keeps repairing violated constraints until it gets a feasible schedule. This approach is suitable when it is desired to move as few activities as possible; however, the question whether the algorithm always ends is still open. The second method takes a number of activities, deallocates them, and then it allocates one activity after another in suitable order in such a way that no constraints are violated. This approach is useful when the intention is to shift activities by a short time distance, regardless of the number of moved activities.

The main shortcoming is that if there is no feasible recovery of the ongoing schedule, neither of the algorithms is able to securely report it. The reason is that deciding whether or not a schedule is recoverable, is NP-complete, therefore it would take prohibitive amount of time. Justification to omit this step is that in real-life environments the schedule recoverability after the breakdown of any particular machine is obvious or can be computed before the schedule execution begins.

Further, we described how to exploit these two suggested algorithms to tackle another disturbance – hot order arrival. Other unexpected events are order cancellation and machine repair. We described the algorithm that simply squeezes a schedule after these minor disturbances by shifting activities in time to the left.

Future work

In order to ease of the enormous computational burden, we omitted possibility of selecting alternative branches, which are involved in the model in the FlowOpt project. This might be of interest for future work. Further investigation is needed also for determining the conditions under which a schedule is recoverable.

Next, it may be of interest to generalize the algorithms for models that for example involve:

- Interruptibility of activities
- Various speeds of resources
- Setup times of resources
- Calendars of availabilities of resources

Oppositely, one might be interested in more efficient algorithms for models somewhat simplified, for example models without time distance constraints.

Bibliography

[1] M. Pinedo, *Scheduling. Theory, algorithms, and systems*, New York: Prentice-Hall, 2002.

[2] M. Pinedo, *Planning and Scheduling in Manufacturing and Services*, New York: Springer, 2005.

[3] R. Barták, M. Jaška, L. Novák, V. Rovenský, T. Skalický, M. Cully, C. Sheahan, D. Thanh-Tung, *FlowOpt: Bridging the Gap Between Optimization Technology and Manufacturing Planners*, In Luc De Raedt et al. (Eds.): Proceedings of 20th European Conference on Artificial Intelligence (ECAI 2012), pp. 1003-1004, IOS Press, 2012 (ISBN 978-1-61499-097-0, DOI:10.3233/978-1-61499-098-7-1003).

[4] J.T. Black, *Manufacturing systems*, in Encyclopedia of Production and Manufacturing Management, Paul M. Swamidass, editor, Kluwer, Norwell, Massachusetts, 2000.

[5] G.Vieira, J. Herrmann, E.Lin, *Rescheduling manufacturing systems: a framework of strategies, policies, and methods*, Journal of Scheduling 6: 39-62, Kluwer Academic Publishers, 2003.

[6] D. Ouelhadj, S. Petrovic, *A survey of dynamic scheduling in manufacturing systems*, Journal of Scheduling, v.12 n.4, p.417-431, 2009.

[7] A. S. Raheja , V. Subramaniam, *Reactive recovery of job shop schedules – a review*, International Journal of Advanced Manufacturing Technology, 19, 756-763, 2002.

[8] R. Barták, T. Skalický, *A local approach to automated correction of violated precedence and resource constraints in manually altered schedules*, Proceedings of the 6th International Workshop on Planning and Scheduling for Space (IWPSS-09), Pasadena, USA, 2009.

[9] A. Cesta, A. Oddi, S.F. Smith, *Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems*, AAAI/IAAI, 17th National Conference on Artificial Intelligence, pp. 742–747, 2000.

[10] A. Oddi, N. Policella, A. Cesta, S.F. Smith, *Boosting the Performance of Iterative Flattening Search*, AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, LNCS 4733, pp. 447–458, Springer Verlag, 2007.

[11] P. Cunningham, B. Smyth, *Case-Based Reasoning in Scheduling: Reusing Solution Components*, International Journal of Production Research, 35: 2947-2961, 1997.

[12] R. Ramkumar, Dr. A. Tamilarasi, Dr. T. Devi, *Multi Criteria Job Shop Schedule Using Fuzzy Logic Control for Multiple Machines Multiple Jobs*, International Journal of Computer Theory and Engineering, Vol. 3, No. 2, ISSN: 1793-8201, April 2011.

[13] A. S. Jain, S. Meeran, *Job-Shop Scheduling Using Neural Networks*, International Journal of Production Research, 36(5), 1249-1272, May 1998.

[14] K. Kouiss, H. Pierreval, N. Mebarki, *Using multi-agent architecture in FMS for dynamic scheduling*, Journal of Intelligent Manufacturing 8, 41-47, 1997.

[15] L. Zhang, T.N. Wong, S. Zhang, S.Y. Wan, *A multi-agent system architecture for integrated process planning and scheduling with meta-heuristics*, Proceedings of the 41st International Conference on Computers & Industrial Engineering, 2011.

[16] R. Barták, T. Müller, H. Rudová, *Minimal Perturbation Problem – A Formal View*, Neural Network World, Volume 13(5), pp. 501-511, 2003.

[17] W. Kocjan, *Dynamic Scheduling – State of the Art Report*, SICS Technical Report T2002:28, SICS.

[18] T. Skalický, *Interactive Scheduling and Visualisation*, Prague, 95 p. Master's thesis, Charles University in Prague, 2011.

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

[20] L.R. Planken, *New Algorithms for the Simple Temporal Problem*, Delft, the Netherlands, 75 p. Master's thesis, Delft University of Technology, 2008.

[21] S.C. Brailsford, Ch.N. Potts, B.M. Smith, *Constraint satisfaction problems: Algorithms and applications*, European Journal of Operational Research 119, 557-581, 1999.

[22] J. Gaschnig, *Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems*, In Proceedings of the 2nd Biennial Conference of the Canadian Society for Computational Studies of Intelligence, pages 268-277, 1978.

[23] R. Dechter, *Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition*, Artificial Intelligence, 41(3):273-312, 1990.

[24] X. Chen, P. van Beek: *Conflict-Directed Backjumping Revisited*, Journal of Artificial Intelligence Research, Volume 14, pages 53-81, 2001.

[25] P. Prosser. *Hybrid Algorithms for the Constraint Satisfaction Problem*, Computational Intelligence, 9(3), 268-299, 1993.

[26] G. Kondrak, P. van Beek. *A Theoretical Evaluation of Selected Backtracking Algorithms*, Artificial Intelligence 89, 365-387, 1997.

[27] R. Barták, O. Čepek. *Nested Temporal Networks with Alternatives*, Hans W. Guesgen, Gerard Ligozat, Jochen Renz, Rita V. Rodriguez (Eds.): Papers from the 2007 AAAI Workshop on Spatial and Temporal Reasoning, Technical Report WS-07-12, AAAI Press, pp. 1-8 (ISBN: 978-1-57735-339-3), 2007.

[28] Ch. Artigues, C. Briand, *The Resource-Constrained Activity Insertion Problem with Minimum and Maximum Time Lags*, Journal of Scheduling, v.12 n.5, p.447-460, 2009.

A. Input Data Acquisition

A.1 Random Model Generation

This section describes the generation of a scheduling problem, suitable for testing the proposed algorithms.

Firstly, the routine introduces resources to the model. It creates 4 resource dependency groups and 9 resources within each resource dependency group, i.e. 36 resources in total. Secondly, the orders are generated, each one containing 8 activities, initially without constraints. The durations of activities are chosen uniformly at random from 1 to 15 time units. Each activity is assigned the resource dependency groups as follows. First resource dependency group is chosen uniformly at random. Next, the routine adds another resource dependency group with probability $\frac{1}{2}$. In the positive case, the routine adds yet another resource dependency group again with probability $\frac{1}{2}$. And finally, in the positive case, the routine adds also the remaining resource dependency group with probability $\frac{1}{2}$. Hence the probability that an activity has all 4 resource dependency groups equals $\frac{1}{8}$, and each activity has at least one resource dependency group.

Thirdly, the routine goes through all the orders in the model and tries to add a given number of constraints into each order. Adding a constraint is conducted as follows. Firstly, activity A_1 and activity A_2 are picked from the order uniformly at random. Then the algorithm introduces a constraint of a random type between activities A_1 and A_2. The constraint is precedence constraint with probability $\frac{1}{2}$, temporal distance constraint with probability $\frac{1}{4}$, and synchronization constraint with probability $\frac{1}{4}$. In case of temporal distance constraint, the type is either minimal distance constraint or maximal distance constraint, either one with probability $\frac{1}{2}$; the distance is chosen uniformly at random from 1 to 30 time units. In case of synchronization constraints, the type is start-to-start, end-to-end, or start-to-end, any one with probability $\frac{1}{3}$. The end-to-start type is omitted because it is symmetrical case of start-to-end, i.e. achievable by interchanging A_1 and A_2. Before the constraint is actually added to the model, it is checked through the IFPC algorithm, whether the addition of the constraint is feasible, and not redundant. The constraint is not added to the model if it would make the STN infeasible as well as if the constraint is redundant. Note that adding the constraint may make some other constraints redundant. If the algorithm exceeds the limited number of attempts (100) before the desired number of constraints is added to the order, it proceeds to another order.

There are 8 activities in one order, which means that in extreme case it may happen that 8 activities requiring a resource from the same resource dependency group must be in process at the same time point. For this reason, there are 9 resource dependencies within each resource dependency group, making it sure that the ongoing schedule is always recoverable when a machine fails.

A.2 Scheduling Model Solver

Solving scheduling problems that are generated as described above is carried out pretty much the same way as in STN-recovery. After the STN is computed, the connected components are acquired, and all single connected components are allocated using the procedure `AllocateComponent` (described in section 5.3.7), allocating the activities in the increasing order of the *MinStart* values. Note that there are no original start times of the activities, unlike in STN-recovery, so that the *MinStart* value of the leftmost activity of each component is 0. Ties are broken arbitrarily.

In such schedules, when attempting to introduce as many constraints as possible, the usage of resources fluctuates around 73 %. To obtain schedules with fully loaded resources, it is inevitable to generate firstly a schedule, i.e. activities allocated to resources, and then add constraints among the activities in such a way that all constraints are satisfied. However, in real life scheduling environments it is impossible to get 100% used resources for such big models.

A.3 Testing Machine

Processor Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz, 3701 Mhz, kernels: 4, logical processors: 8

RAM 8,00 GB

OS Microsoft Windows 7 Professional 64-bit Version 6.1.7601 Service Pack 1 Assembly 7601

IDE Microsoft Visual Studio Ultimate 2012 Version 11.0.61030.00 Update 4

.Net Microsoft .NET Framework Version 4.5.50938

B. Evaluation Graphs

B.1 Right Shift Affected Heuristics

B.1.1 Dependence on Model Size

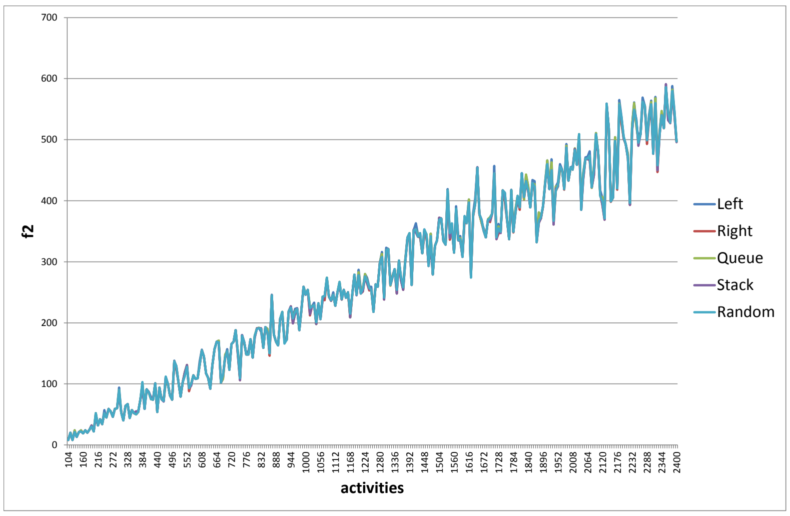

Figure B.1: Comparison of schedule quality (measured using function f2) for different activity selection heuristics

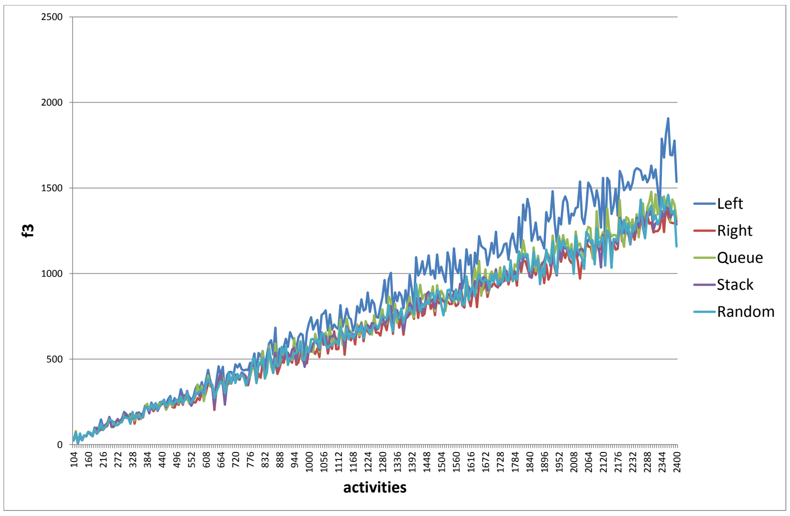

Figure B.2: Comparison of schedule quality (measured using function f3) for different activity selection heuristics

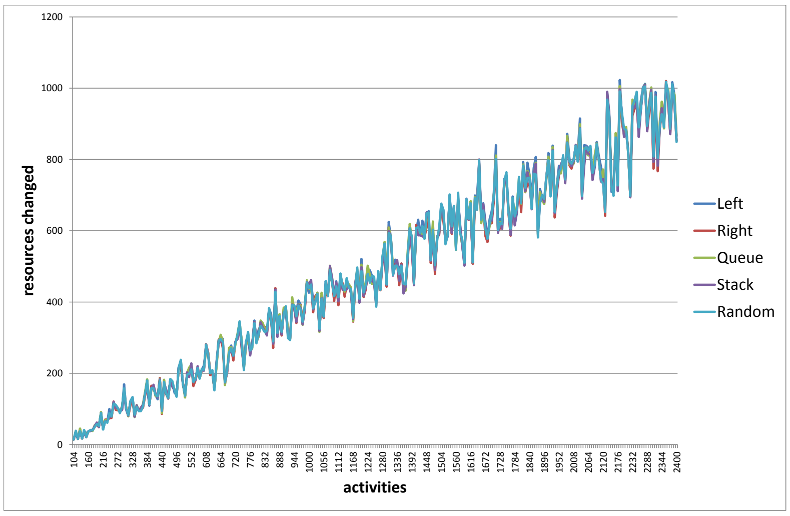

Figure B.3: Comparison of schedule quality (measured by number of changed resource selections) for different activity selection heuristics

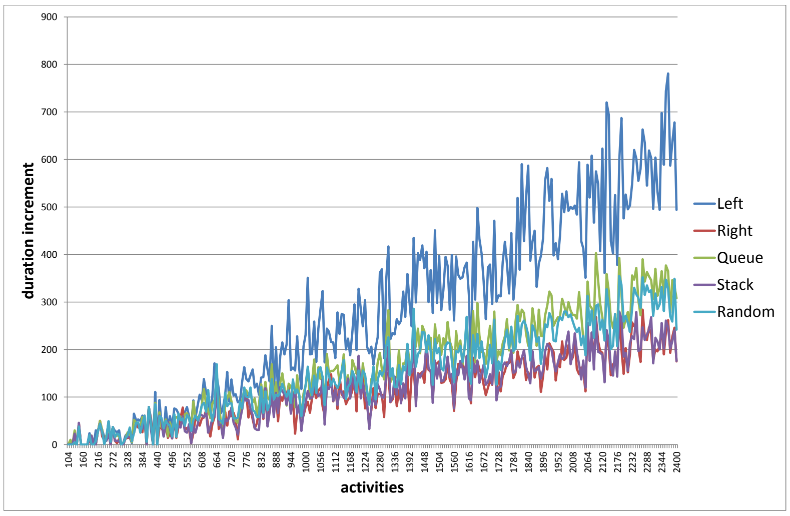

Figure B.4: Comparison of schedule quality (measured using makespan increment) for different activity selection heuristics

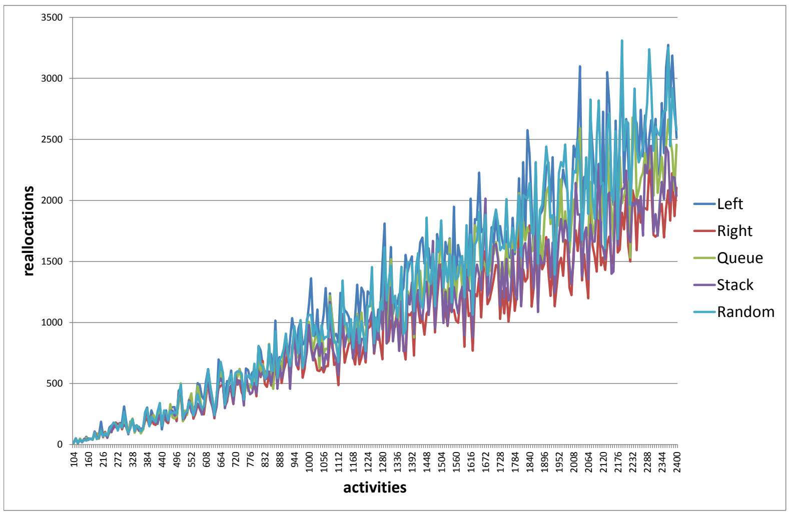

Figure B.5: Comparison of schedule quality (measured by number of reallocations) for different activity selection heuristics

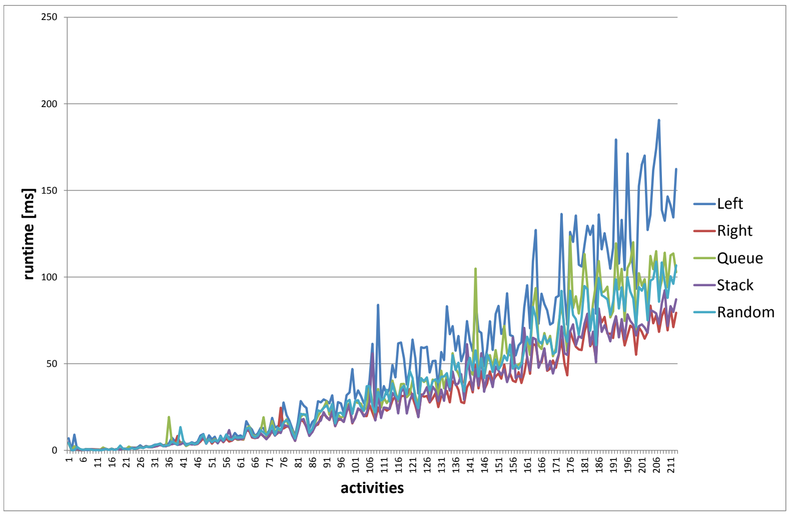

Figure B.6: Comparison of schedule quality (measured in run-time) for different activity selection heuristics

B.1.2 Dependence on Constraint Density

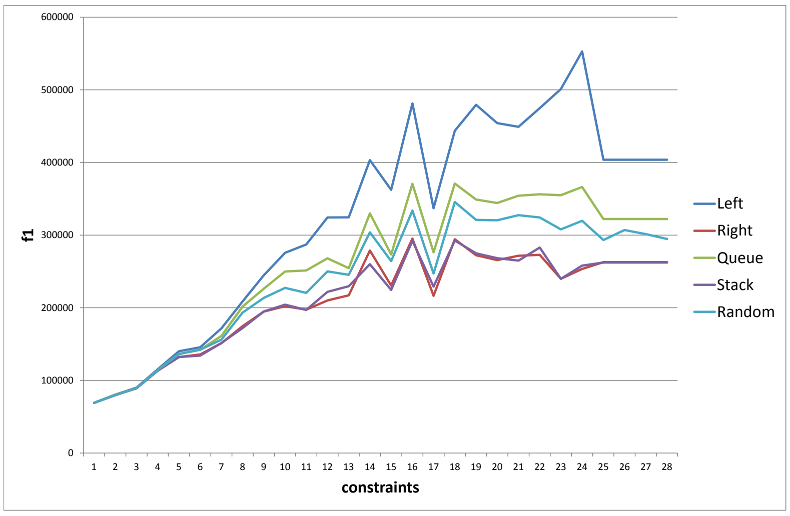

Figure B.7: Comparison of schedule quality (measured using function f1) for different activity selection heuristics

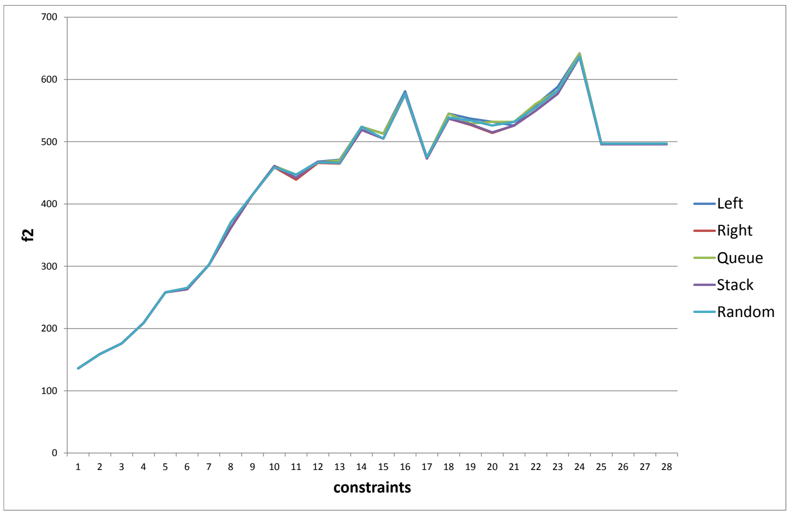

Figure B.8: Comparison of schedule quality (measured using function f2) for different activity selection heuristics

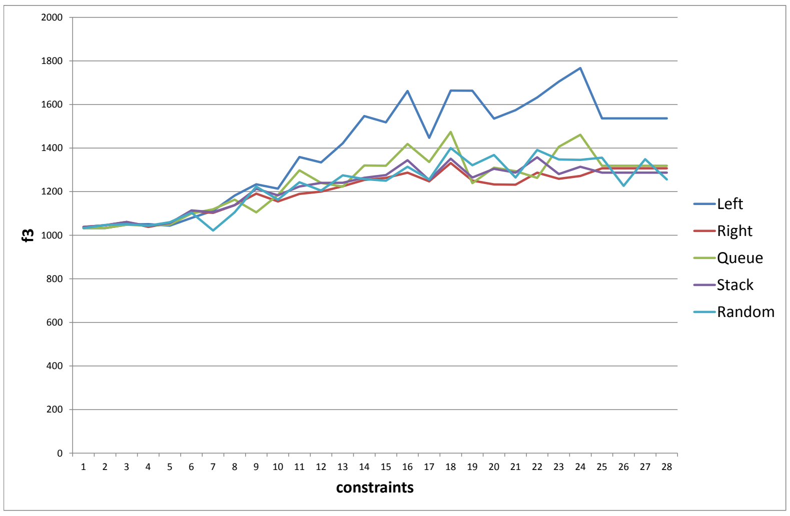

Figure B.9: Comparison of schedule quality (measured using function f3) for different activity selection heuristics

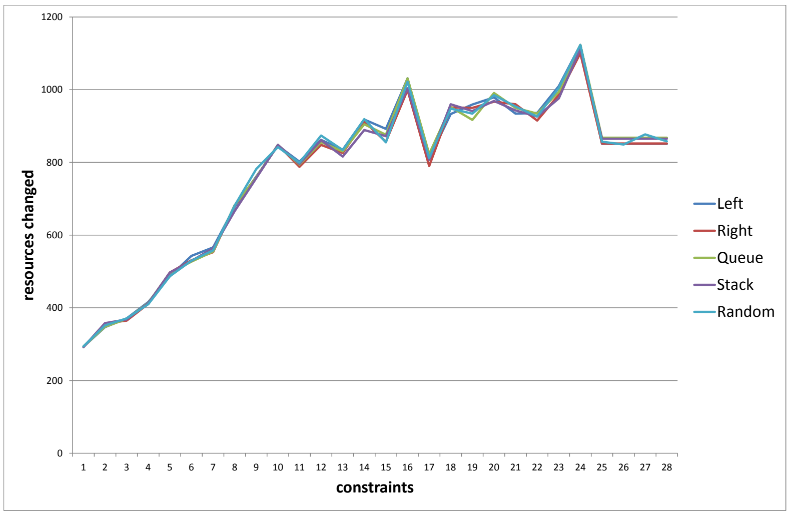

Figure B.10: Comparison of schedule quality (measured by number of changed resource selections) for different activity selection heuristics

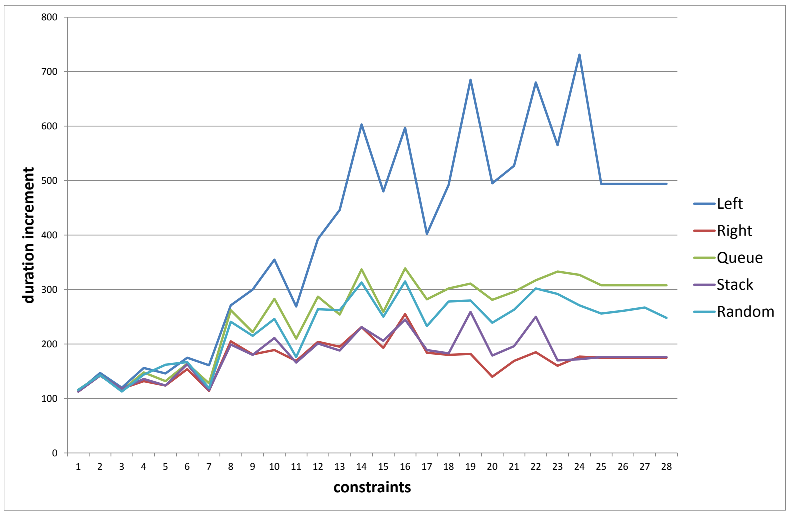

Figure B.11: Comparison of schedule quality (measured using makespan increment) for different activity selection heuristics

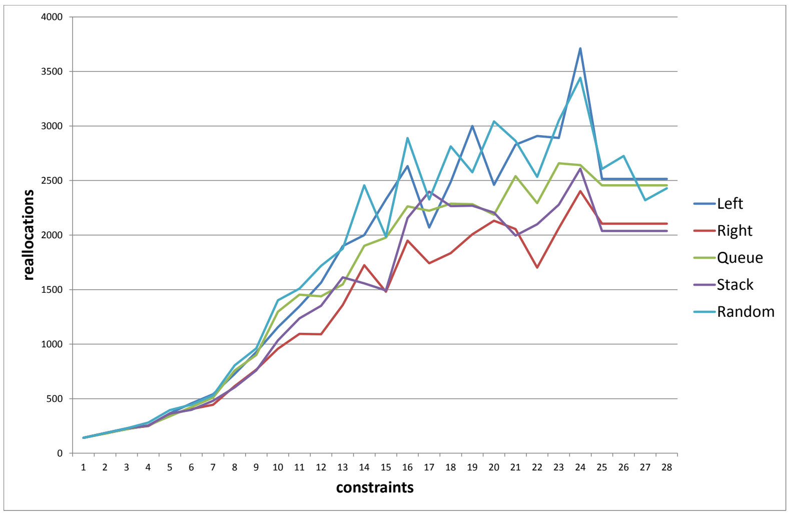

Figure B.12: Comparison of schedule quality (measured by number of reallocations) for different activity selection heuristics

B.2 STN-Recovery

B.2.1 Dependence on Model Size

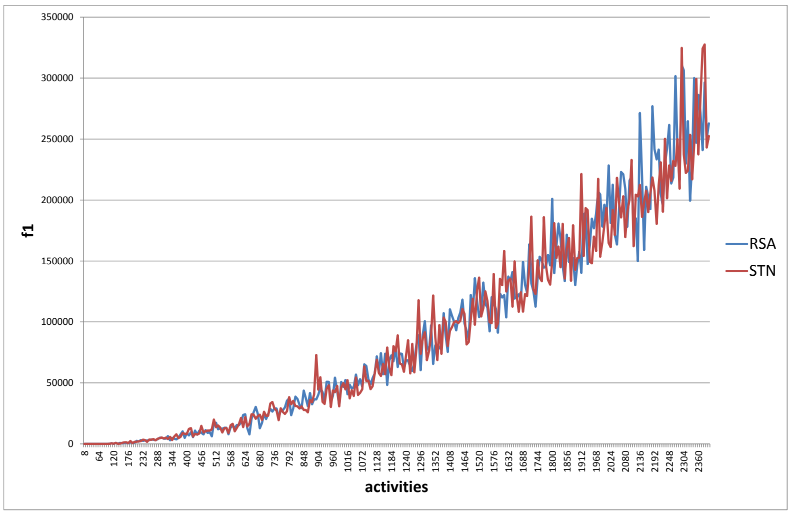

Figure B.13: Comparison of schedule quality (measured using function f1) for Right Shift Affected and STN-recovery

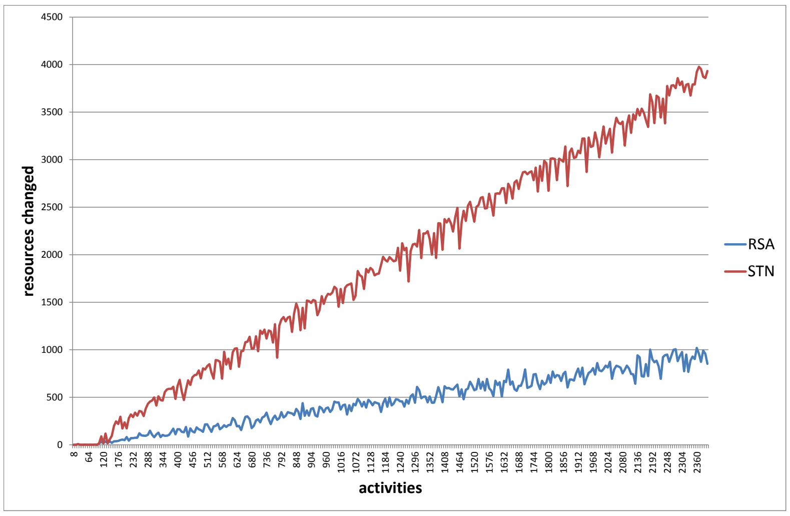

Figure B.14: Comparison of schedule quality (measured by number of changed resource selections) for Right Shift Affected and STN-recovery

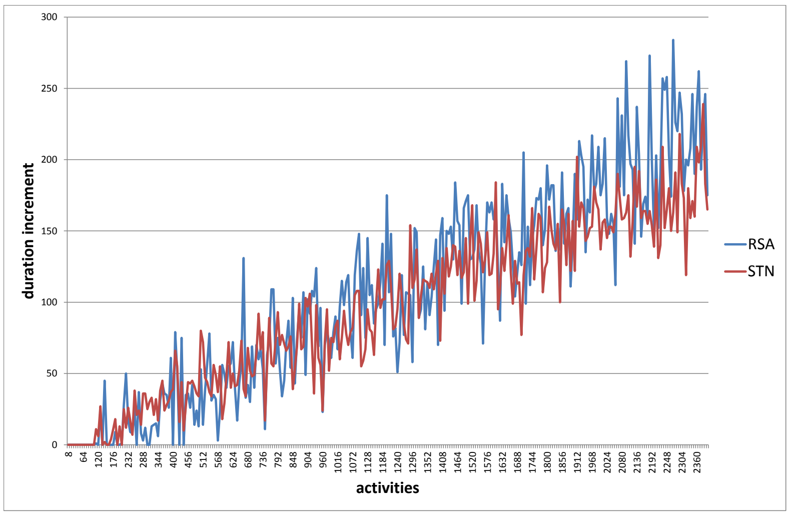

Figure B.15: Comparison of schedule quality (measured using makespan increment) for Right Shift Affected and STN-recovery

B.2.2 Dependence on Constraint Density

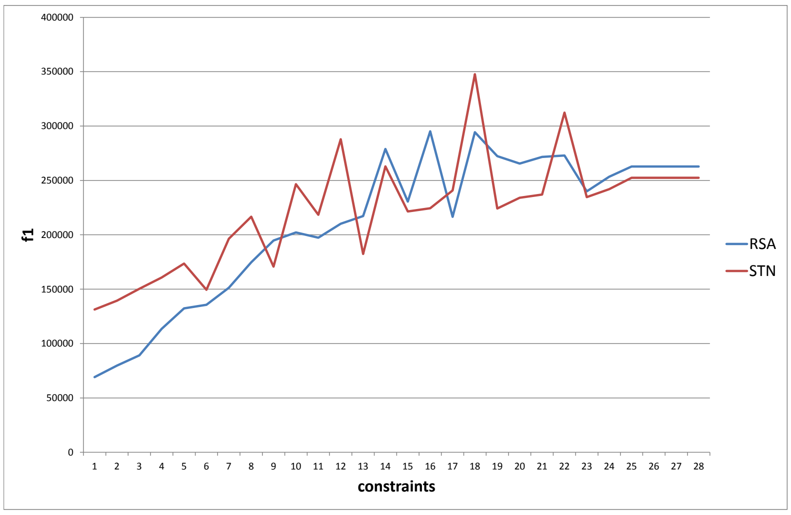

Figure B.16: Comparison of schedule quality (measured using function f1) for Right Shift Affected and STN-recovery

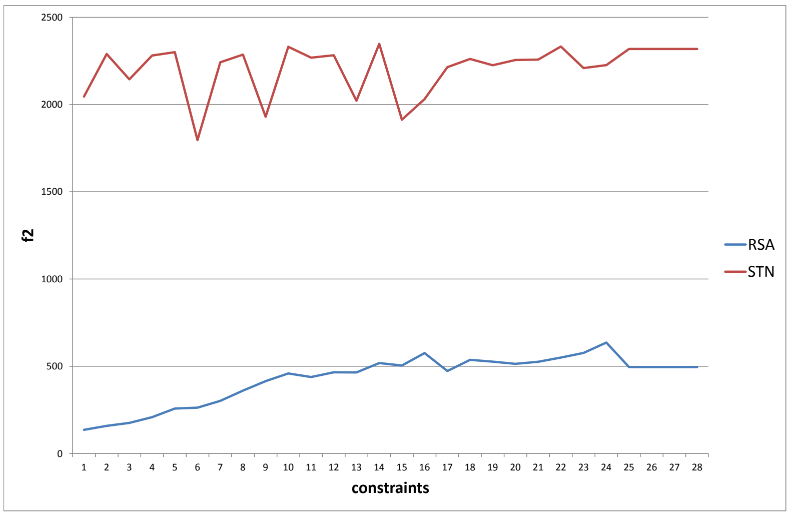

Figure B.17: Comparison of schedule quality (measured using function f2) for Right Shift Affected and STN-recovery

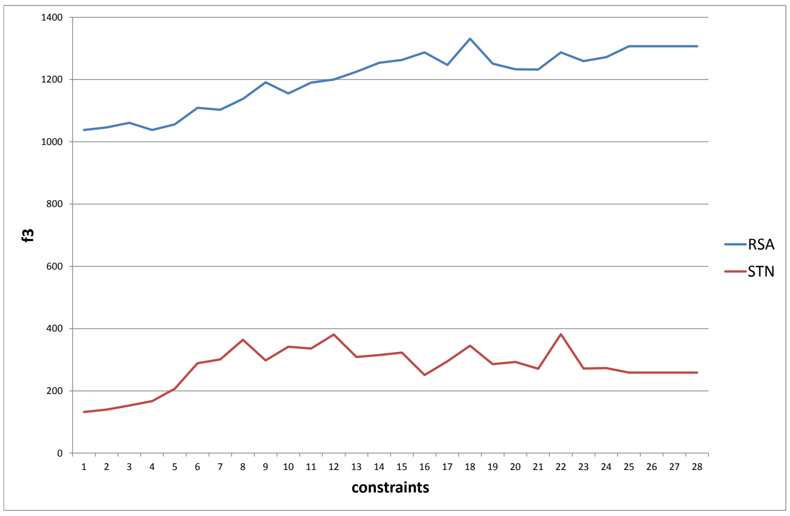

Figure B.18: Comparison of schedule quality (measured using function f3) for Right Shift Affected and STN-recovery

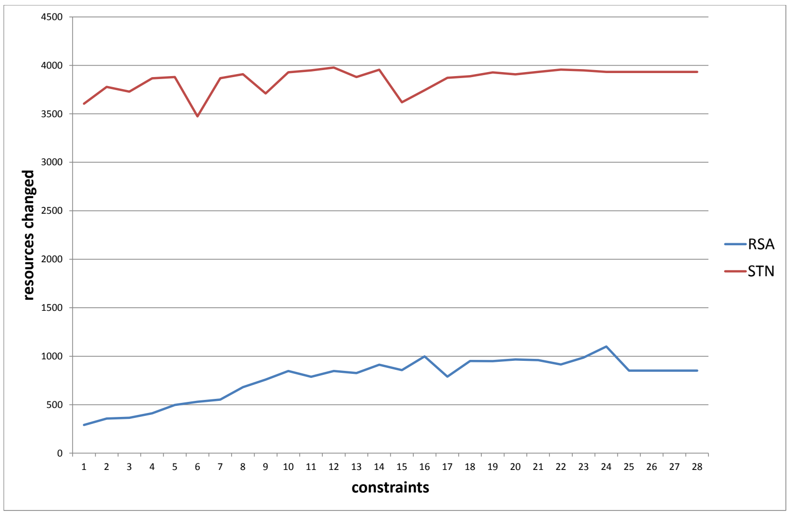

Figure B.19: Comparison of schedule quality (measured by number of changed resource selections) for Right Shift Affected and STN-recovery

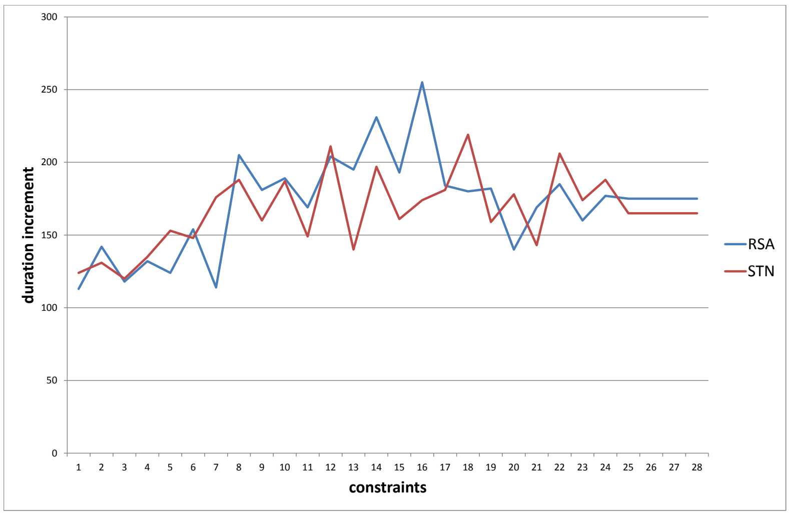

Figure B.20: Comparison of schedule quality (measured using makespan increment) for Right Shift Affected and STN-recovery

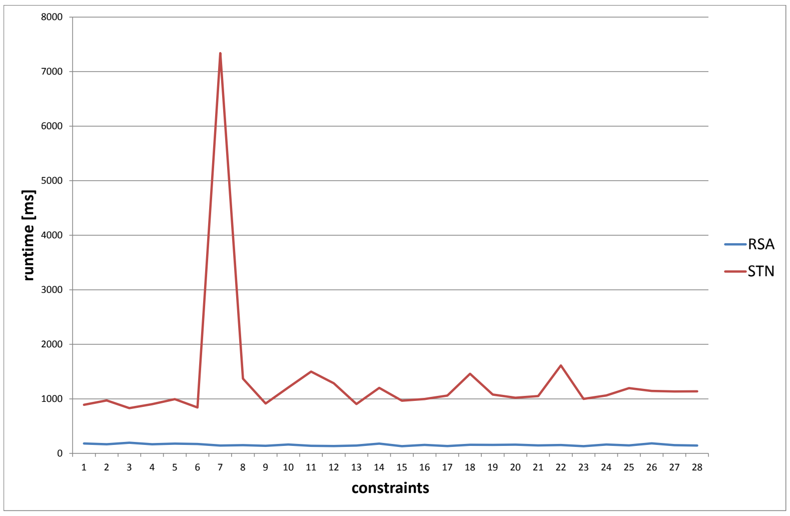

Figure B.21: Comparison of schedule quality (measured in run-time) for Right Shift Affected and STN-recovery

B.2.3 Dependence on Connected Component Size

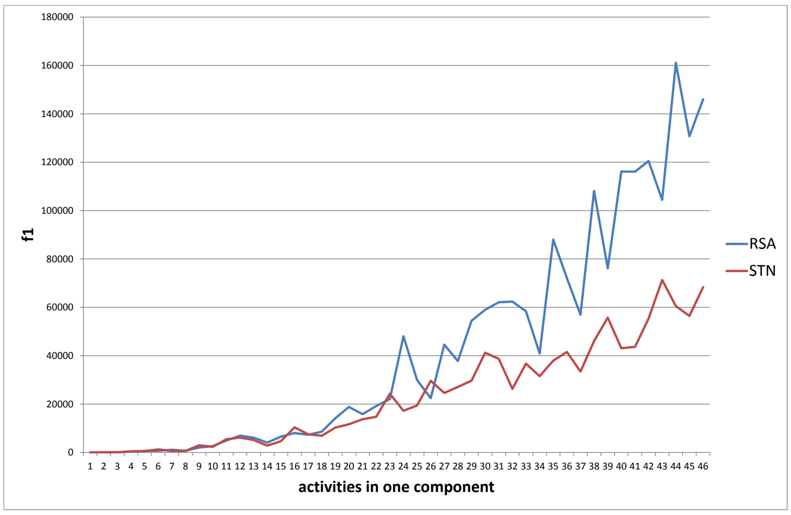

Figure B.22: Comparison of schedule quality (measured using function f1) for Right Shift Affected and STN-recovery

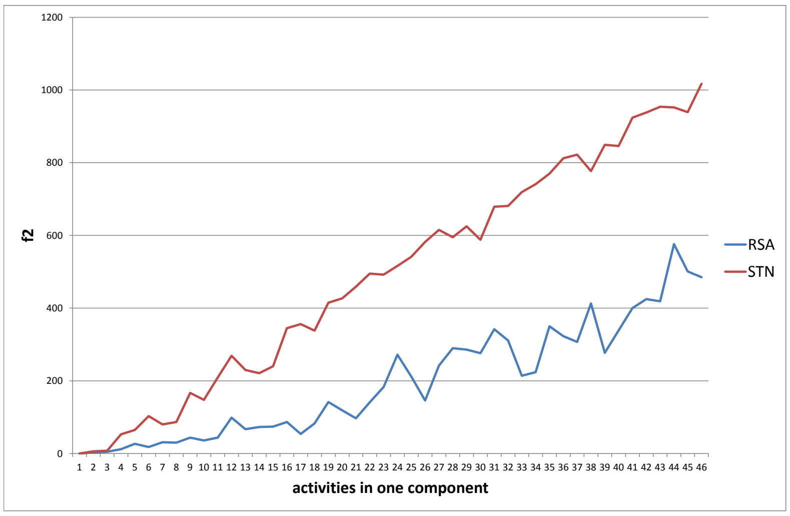

Figure B.23: Comparison of schedule quality (measured using function f2) for Right Shift Affected and STN-recovery

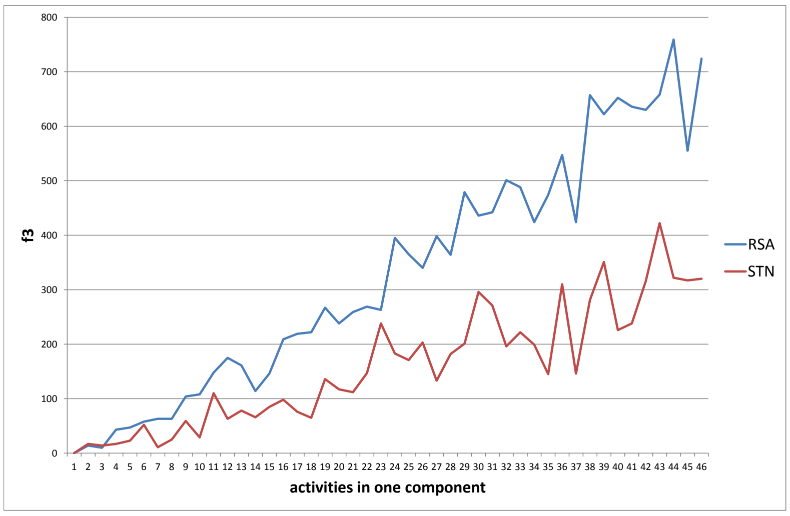

Figure B.24: Comparison of schedule quality (measured using function f3) for Right Shift Affected and STN-recovery

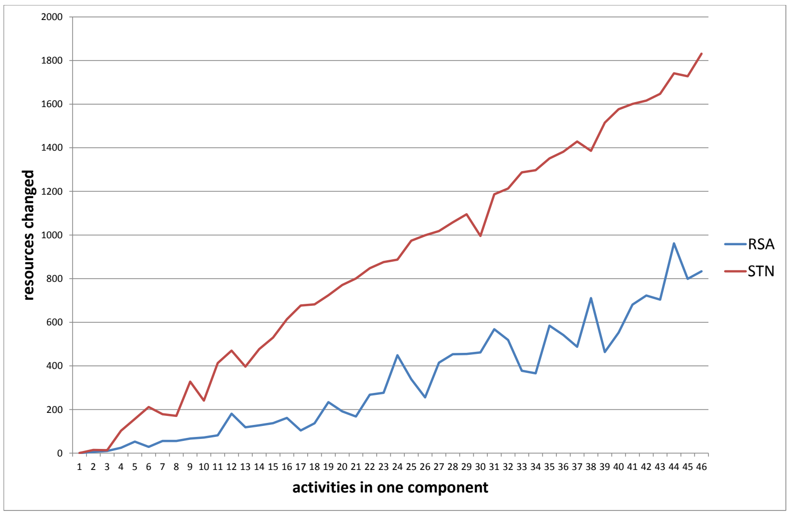

Figure B.25: Comparison of schedule quality (measured by number of changed resource selections) for Right Shift Affected and STN-recovery

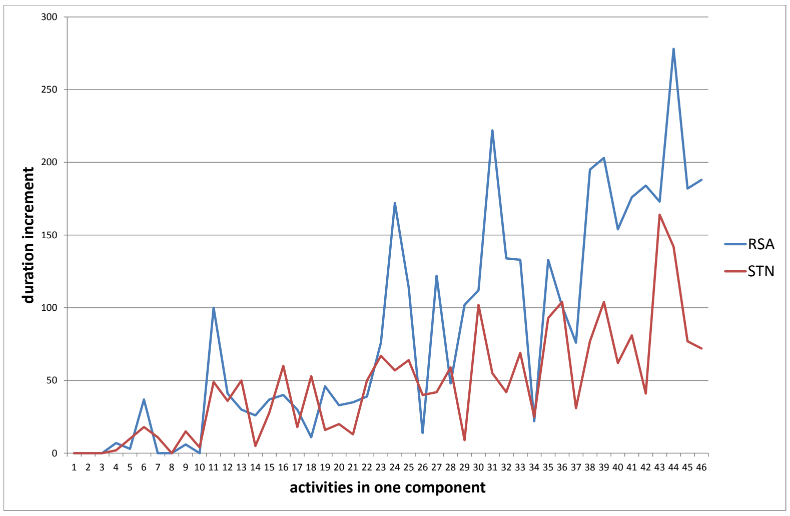

Figure B.26: Comparison of schedule quality (measured using makespan increment) for Right Shift Affected and STN-recovery