Dynamic scheduling
State of the art report

by

Waldemar Kocjan
waldemar@sics.se

October 2002

Abstract

Most of the research literature concerning scheduling concentrates on the static problems, i.e. problems where all input data is known and does not vary over the time. However, the real world scheduling problems are very seldom static. Events like machine failures or work overloads are, in some situations, impossible to predict.
Dynamic scheduling is a research field which take into consideration uncertainty and dynamic changes in the scheduling problem. This paper gives an overview of the state of the art in the field of dynamic scheduling.

SICS Technical Report T2002:28
ISSN 1100-3154
ISRN:SICS-T–2002/28-SE

Keywords Dynamic scheduling, Uncertainty in scheduling, Dynamic Constraint Satisfaction, Constraint programming

Contents

Chapter 1

Introduction

Scheduling basically concerns assigning start times to a set of activities. In recent years, automated computer based scheduling has drawn a lot of attention from industry and commerce. In cases of scheduling for industry, scheduled activities are often subject to resource constraints. For example, when scheduling activities for the assembly line, the number and the capacity of the used machines must be taken into consideration, as well as, if necessary, human resources which service them.

In the research literature scheduling is often seen as a function of perfect, known inputs. The set of activities and their characteristics as well as resources used, their capacity etc. are assumed to be known and unvarying over time. In real life executed schedules are often subject to unexpected events like a machine failure or a temporary work overload, which causes their inconsistency.

This has led some researchers to redefining the concept of optimality of the schedule. E.g. Hildum states in [Hil94] that a schedule which is determined to be optimal prior to its execution is optimal only to the degree that the real world behaves during the execution of the schedule exactly as predicted. The natural consequence of this statement is that since the true optimality of the schedule can be only verified during its execution and since its practically impossible to predict all the factors causing inconsistency of an executing schedule, it is not practical to devote a lot of effort to ensure theoretical optimality of the schedule before its execution in the real life.

Similar conclusions can be drawn with respect to the robustness of a schedule. It can be stated that, like in case of optimality, robustness of given schedule can be defined only to a degree in which it is possible to predict the behavior of a scheduling environment.

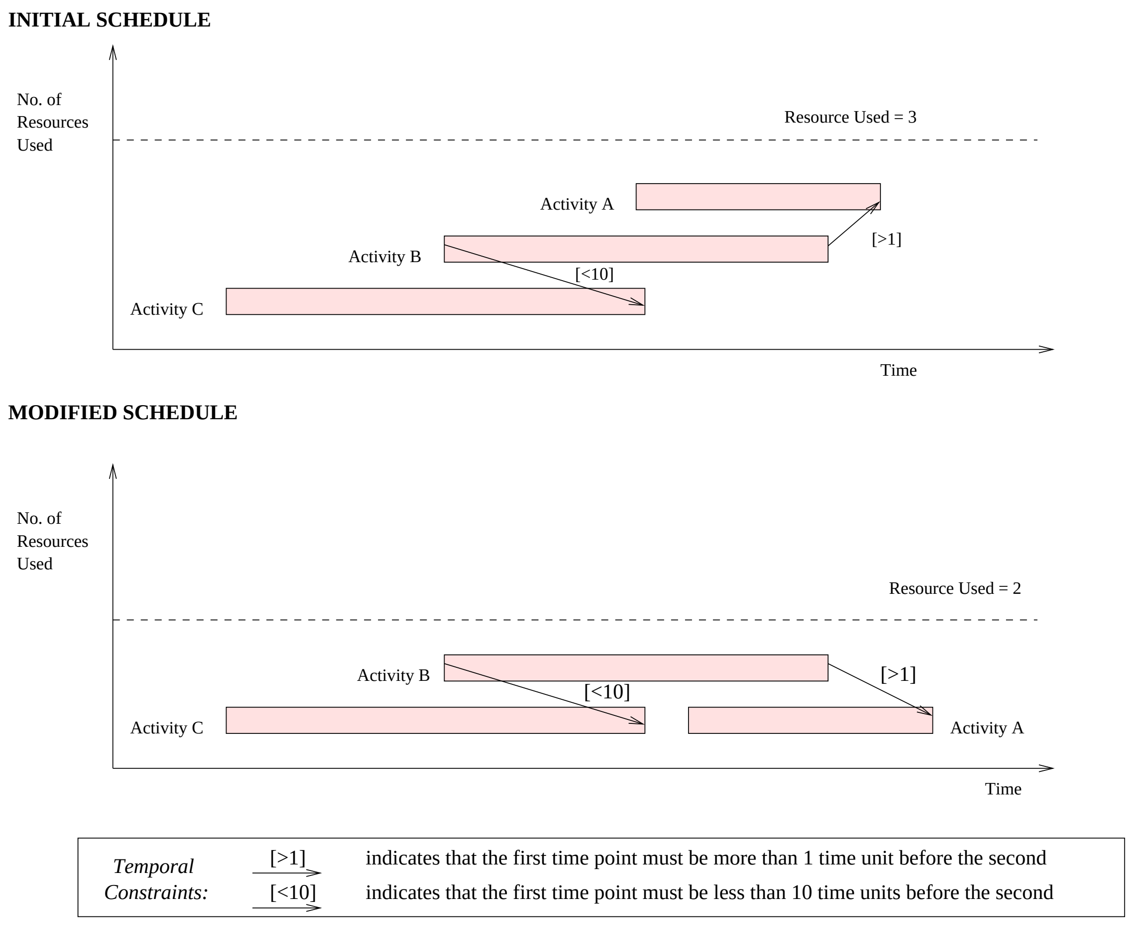
This report gives an overview of the state of the art in on-line scheduling in dynamic environments.

1.1 Motivation

1.1.1 A Toy Example

The essence of dynamic scheduling can be illustrated by the toy example in Figure 1.1 (after [SW00]). In the figure tree activities A,B and C are scheduled. They utilize three identical resources. The activities in the schedule are subject to temporal constraints relating their start and end times. Those constraints are satisfied in the current schedule.

The second part of the figure shows a schedule which is rearranged as a response to a reduction of the number of available resources. The modified schedule is consistent with respect to all temporal constraints.



indicates that the first time point must be more than 1 time unit before the second

indicates that the first time point must be less than 10 time units before the second

Figure 1.1: Example of a schedule modified to reduce resources

1.1.2 Commercial Applications

There is a strong commercial need for techniques that can manage the process of schedule development better. Usually, executed schedules are subjects to some unexpected events that cause their infeasibility. Such disruptive events must be taken into consideration if schedules are going to be executed efficiently.

One way of handling any disruption is to consider all possible sources of uncertainty before an actual schedule is created. Such a pro–active approach is directed toward creating robust, fault–tolerant schedules which do not need to be corrected during the actual execution. Examples of such approaches are redundancy–based techniques such as fault tolerant real time scheduling ([GMM95, Gho96]), slack–based protection ([LW94]), temporal protection ([CF90, Gao95]), just–in–case scheduling ([DBS94]) and others (For classification and examples of scheduling under uncertainty see [DB00]).

The robustness of a schedule can be guaranteed only in case of small changes. Introducing e.g. temporal redundancy on a resource can make a tradeoff between robustness of a schedule and degree of resource utilization too expensive. In many other situations predicting all sources of uncertainty may be impossible. In both situations major changes will still cause an inconsistency of an executed schedule.

Algorithms for dynamic scheduling should be able to manage any disruption of a schedule caused by changes in scheduling environment.

Such changes can be classified in tree major groups:

Activity Changes

Request for new or extended activities can result in resource contention and inconsistency of a schedule. In the long term scheduling introducing new activities can aim at improving the schedule efficiency and degree of resource utilization (e.g. leasing out some resources leads). In the short term scheduling activities are introduced as they arise (e.g. emergency service). Changes in activity duration and increased level of resource usage can occur.

Resource Changes

Primary reduction of resources (e.g. machine failure) can disrupt a schedule. Resource changes may be also requested to reduce the cost of a schedule (e.g. machine utilization problems). Shorter term resource changes are usually connected with resource failure.

Temporal Changes

The most frequent form of temporal change is a contraction of schedule horizon. Long term temporal changes (e.g. changing a timetable in public transport for regularity) and short time changes (e.g downstream effect of delayed aircraft or train) may also cause schedule inconsistency.

1.2 Related Work

Scheduling under uncertainty was surveyed by Davenport and Beck in [DB00]. The paper gives a short classification and a brief description of different off–line and on–line scheduling techniques. Many of the methods described in the survey were used in scheduling real–life problems.

An introductory survey on on-line scheduling was also given in [Sga97]. This survey concentrates on theoretical problems of on-line job–shop scheduling and does not cover large scale, complex, optimization problems.

1.3 Structure of this paper

In Chapter 2 a short introduction to constraint satisfaction and scheduling models originating from it is given. The chapter briefly presents methods for solving those models.

Chapter 3 describes some frequent dynamic scheduling problems, which often arise in industry. The way of formalizing those problems, using models described in Chapter 2, is presented.

The idea of cooperative solvers is explained in Chapter 4. The chapter exploits mainly so called probing methods originating in operative aircraft planning.

Other interesting methods are summarized in Chapter 5.

Finally, Chapter 6 concludes this report and gives directions for future work.

Acknowledgment

This research is part of the HEMCO project of Swedish Institute of Computer Science and is supported by Institute For Research And Competence Holding AB (IRECO).

Author wish to thank Markus Bohlin of Swedish Institute of Computer Science for valuable comments, Per Kreuger of Swedish Institute of Computer Science and Björn Lisper of Mälardalen University in Västerås for supervising this work.

Chapter 2

Constraint Satisfaction Models

A model frequently used to represent classical, "static" scheduling problem is the constraint satisfaction problem (CSP). The dynamic variant of scheduling is often an extension of CSP. This chapter presents the constraint satisfaction problem and dynamic scheduling models originating from it. The first section describes basic concepts of CSP and presents methods to solve it. The following sections describe how the idea of CSP is extended to handle scheduling problems involving permanent (Dynamic CSP) and temporary changes (Recurrent Dynamic CSP) in scheduling environment.

2.1 Constraint Satisfaction Problem

The constraint satisfaction problems are problems of assigning certain values to the variables restricted by some conditions. For example, a graph coloring problem can be represented as a CSP. In that case the nodes of a graph are variables of the CSP and the colors which can be assigned to a node represent values in the domain of every variable. An arc between two nodes represents the constraint that those two nodes may not take the same color value.

2.1.1 Basic definitions

Before we give a formal definition of CSP we need to introduce some basic concepts. The definitions below follow those given in [Tsa93] if not stated otherwise.

Definition 2.1 *The **domain of a variable** is a set of all possible values that can be assigned to the variable.*

The domain of variable x is denoted D_x.

Definition 2.2 *A **label** is a variable–value pair that represents the assignment of the value to the variable.*

In this paper we use $< x, v >$ to denote the label assigning value v to the variable x, where $v \in D_x$.

Definition 2.3 *A* ***compound label*** *is the simultaneous assignment of values to the set of variables.*

A compound label is denoted as $(< x_1, v_1 >, < x_2, v_2 >, \dots, < x_n, v_n >)$ and represents assigning $v_1, v_2, \dots, v_n$ to $x_1, x_2, \dots, x_n$ respectively.

Definition 2.4 *A* ***constraint*** *on the set of variables is a set of compound labels on the variables.*

A constraint on the set of variables is generally a restriction on the values which they can take simultaneously. Conceptually, a constraint can be seen as as a set that contains all the legal compound labels for the variables of a constraint. A constraint on the set of variables S is denoted C_S.

Definition 2.5 *If the variables of the compound label $\mathcal{X}$ are the same as those of the compound labels in a constraint $\mathcal{C}$, then $\mathcal{X}$* ***satisfies*** *$\mathcal{C}$ if and only if $\mathcal{X}$ is an element of C.*

Given these basic definitions we can give here formal definition of Constraint Satisfaction Problem.

Definition 2.6 *A* ***Constraint Satisfaction Problem*** *is a triple (Z, D, C) where Z is a finite set of variables $\{x_1, x_2, \dots, x_n\}$, D is a set of functions which maps every variable in Z to a finite set of objects of arbitrary type and C is a finite set of constraints of variables in Z.*

As mentioned above the task in CSP is to assign a value to each variable such that all the constraints are simultaneously satisfied.

Definition 2.7 *A* ***solution tuple*** *of a CSP(Z, D, C) is a compound label $\{< x_1, v_1 >, \dots, < x_n, v_n >\}$ for all variables $\{v_1 \in D, \dots, v_n \in D_{x_n}\}$ which satisfies all the constraints $c \in C$:*

2.1.2 Representation

Any given CSP may be represented as a *constraint hypergraph.*

Definition 2.8 *A* ***hypergraph*** *is a tuple $(\mathcal{V}, \mathcal{E})$,where $\mathcal{V}$ is a set of nodes and $\mathcal{E}$ is a set of hyperedges, each of them connecting a subset of nodes.*

Definition 2.9 *The* ***constraint hypergraph*** *of a CSP $\mathcal{P} = (Z, D, C)$ is a hypergraph in which each node represents a variable in Z and each hyperedge represents a constraint in C.*

The hypergraph of CSP $\mathcal{P}$ is denoted $\mathcal{H}(P)$.

Since every CSP with k–ary constraint can be transformed into a binary problem, i.e., a CSP with only unary and binary constraints [Bes92, BvB98, SW99], then every constraint hypergraph can be reduced to *constraint graph.*

Definition 2.10 *The* ***constraint graph*** *of a general CSP(Z, D, C) is an undirected graph in which each node represents a variable in Z, and for every pair of distinct nodes whose corresponding variables are involved in any k–constraint in C there is an edge between them. The constraint graph of a general CSP $\mathcal{P}$ is also called a* ***primal graph*** *of $\mathcal{P}$.*

In this paper $G(\mathcal{P})$ will be used to denote the constraint graph of CSP$(\mathcal{P})$.

In some literature originating in Truth Maintenance Systems, e.g. [DD88], a Constraint Satisfaction Problem is referred to as a Constraint Network. It is said that a binary Constraint Network may be associated with a constraint graph.

2.1.3 Solving methods

Solving constraint satisfaction problem is often carried out by a combination of problem reduction and search. In the literature, problem reduction is often referred to as *achieving consistency.*

Consistency

Consistency algorithms were first introduced for binary constraints problems. Binary CSPs are associated with graphs, where the nodes represent variables and arcs binary constraints. The concept of consistency is related to this representation.

Definition 2.11 *A CSP is* ***node consistent*** *if and only if for all variables, all values in its domain satisfy the unary constraints on that variable.*

Achieving node consistency is trivial. All that is needed is to determine if each element in each domain satisfies a unary constraint [Mon74a, Mon74b, Mac77].

Definition 2.12 *An arc (x, y) in the constraint graph of a CSP(Z, D, C) is* ***arc–consistent*** *(AC) if and only if for every value a in the domain of x which satisfies the constraint on x, there exists a value in the domain of y which is compatible with $< x, a >$.*

There exists several methods for achieving arc–consistency. The most naive approaches, *AC–1* and *AC–2*, initiate a queue of all edges and checks for each element in the queue if for each element in domain of variable x there exists a value in domain of variable y, where x and y are subject to the binary constraint C_{xy}, such that C_{xy} holds [Mac77, Wal75].

More effective algorithms *AC-3*[Mac77], *AC-4* [MH86] up to *AC-7* [BFR95] have been developed. All of them check for consistency for all values of $x, y \in C_{x,y}$. They differ on the level of specialization and implementation.

Since arc consistency algorithms referred above check for consistency for every value in domain of one variable with the values of another variable in the constraint, they are usually very costly with respect to run time. A more effective method of propagation, so called *arc B–consistency*, was developed for numerical CSP.

A numerical CSP is a subset of CSPs which has numerical variables connected with domains, which are sets of numerical values and which are subject to numerical constraints. In case of arc B–consistency the consistency check is applied only on the bounds of variable domain. The formal definition of the arc B–consistency was first given in [Lho93]. However, the method was used in practise long before its formal definition was given.

Definition 2.13 *Let $\mathcal{P}$ be a numerical CSP, X a variable of $\mathcal{P}$, $D_X = [a, b]$. D_X is* ***arc B–consistent*** *if and only if for all constraints $C(X_1, \ldots, X_k)$ over X holds that:*
1. $\exists v_1, \ldots, v_k \in D_1 \times \ldots D_k : C(a, v_1, \ldots, v_k)$ is satisfied
2. $\exists v_1, \ldots, v_k \in D_1 \times \ldots D_k : C(b, v_1, \ldots, v_k)$ is satisfied
An NCSP is arc B–consistent if all the domains are arc B–consistent.

More informally arc B–consistency is a form of arc consistency restricted to the bounds of the domain.

Backtrack Search

Solving CSP propagation methods are usually combined with search. The basic algorithm of searching for a solution tuple is *simple backtracking*, the method widely used in problem solving. In CSP context, the basic operation is to pick one variable at the time, and consider one value for it at the time, making sure that the newly picked label is compatible with all the labels picked so far. Assigning a value to a variable is called *labeling*. If the current variable with the picked value violates some constraints (consistency check), then the alternative value, if available, is chosen. If all the variables are labeled, then the problem is solved. At any stage, if for some variable there is no value which can be assigned to it without violating any constraints, the label which was last picked is revised and an alternative value, if available, is picked. This search is carried until a solution is found or all the combination of labels have been tested and have failed, which indicates that the problem has no solution.

For more detailed description of backtrack search see e.g. [Tsa93].

Global constraints

One weakness of consistency based methods is that all the primitive constraints are examined in isolation from each other. In many cases knowledge about other constraints can dramatically improve domain pruning. Consider the `alldifferent`$(\{V_1, \ldots, V_n\})$ constraint [Rég94], which holds whenever each of the variables V_1 to V_n in its argument takes a different value. Consider X, Y, Z with domains $D_X = D_Y = D_Z = \{1, 2\}$ to be arguments to `alldifferent`. It is obvious, that there is no such assignment to X, Y and Z, which satisfies such conditions, so the consistency check should return *false*.

Nevertheless, if the constraint is propagated using arc–consistency techniques, Figure 2.1, then, for every pair of variables $(X, Y), (X, Z), (Y, Z)$, for every value in the domain of one variable there exists some value in domain of the other such that inequality holds. To discover inconsistency for constraints of `alldifferent` type it is necessary to take into consideration information about all variables which are subject to such a constraint and their domains. For this purpose a maximal bipartite matching algorithm which matches variables

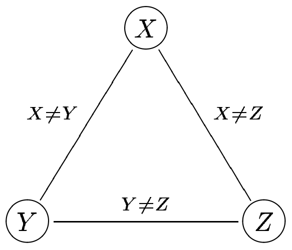


Figure 2.1: The `alldifferent` constraint

against all values was developed [LP86, Rég94]. E.g. the maximal bipartite matching algorithm for the `alldifferent` example will determine that the most 2 variables can be assigned legal values, so the constraint is unsatisfiable, Figure 2.2.

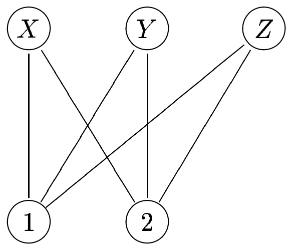


Figure 2.2: Matching variables against values

A specialized group of constraints, so called global constraints, which uses various algorithms to achieve higher level of consistency, was developed. Many of global constraints, e.g. `serialize, cumulative, diffn` [CP89, CP94, AB92, BE94] etc., were developed to solve complex scheduling and geometrical problems [Bel00, KCSÅ01].

2.2 Dynamic CSP

The Constraint Satisfaction model presented in the previous section applies to the problems, i.e. problems which require a one time solution of a system representing all the available information.

An extension of CSP which deals with uncertainty and dynamic changes in scheduling environment was presented as Dynamic Constraint Satisfaction Problem (DCSP) by Dechter and Dechter in [DD88]. The concept of DCSP was introduced in context of Truth Maintenance Systems [Doy81], but can be easily adopted to Constraint Satisfaction Problem. On the relation between Truth Maintenance System and CSP see [McA90, dK89].

Definition 2.14 *A **Dynamic Constraint Satisfaction Problem** is a sequence of CSPs, where every CSP is a result of changes in the preceding one. A difference between two consecutive CSPs is expressed by the set of constraints added C_{add} and constraints deleted from the problem. C_{del}.*

As mentioned above a constraint satisfaction problem is a static problem. A dynamic constraint satisfaction problem is modelled as a sequence of static problems, which suggest that the changes in the scheduling environment are more or less permanent.

Most of the methods based on DCSP, including some described in this report, (see Sections 4.2 and 4.3) solve aech new scheduling problem separately as an ordinary, static CSP using the original problem to e.g. measure the discrepancy between the new and the original solution.

Nevertheless, solving every new CSP, created as a result of an alternation to the original problem, from scratch has two important drawbacks: inefficiency, and possible instability of the new solution in face of new changes to the problem. Several methods were proposed to remedy these drawbacks. The following three groups of algorithms can be distinguished:

- incremental search methods, [HP91];
- local repair methods, e.g. [MJPL92, VS94b];
- constraint recording methods, which records any kind of constraint which can be deduced in the framework of a CSP and reuse this information when solving any new CSP. [HP91, VS94b, VS94a].

This tree classes of algorithms for dynamic CSPs are described below.

2.2.1 Incremental search methods

The main idea of incremental search methods for a sequence of CSPs is to preserve execution context of a problem and use it while solving subsequent CSPs.

The incremental search for dynamic CSPs was first presented in [vEOT86] and developed later in [OTF86, Cha87, Cha89, MS89]. These approaches achieve incrementality using backtracking search. The incremental search based on backtracking works as follows. For any given DCSP $\mathcal{P} =< \Theta_0, \ldots, \Theta_n >$, first Θ_0 is solved as an ordinary, static CSP and the solution α_0 is obtained. Then, for any given Θ_i, if the solution α_{i-1} is violated by any added constraint, the program backtracks to the latest decision point. Another solution is then generated and tested until all of the constraints in Θ_i are satisfied or the search fails.

The method has some severe deficiencies pointed out in [HP91]. One is connected with the fact that the backtracking point selected when the constraints are not satisfied, has probably nothing to do with the cause of constraint violation. This results in a lot of redundant computation, which could be avoided by choosing more accurate backtracking point.

To remedy this problem the authors of [HP91] propose a method based on re–execution, where the search for a solution to a new problem is guided by informations gathered during solving previous problems. The gathered information consists of the computation path leading to a solution. The specific computation path used to guide the search is called an oracle and is denoted $O(\Theta_i)$. In the framework presented in [HP91] the oracle $O(\Theta_i)$ is associated with the set of constraints of CSP Θ_i. These oracles will be used during re–execution to achieve incrementality.

In a case of addition of constraints, the solution to the new CSP $\Theta_i \in \mathcal{P}$ can be found by following the oracle $O(\Theta_{i-1})$ until a failure, due to the added constraint, occurs or a solution is found. If a failure occurs, execution goes back to the recent choice point and proceeds from there in the standard way, i.e. without any oracle.

Since the node where backtracking occurs is associated with the first choice point where the constraints are not satisfiable, this point will be a best possible backtracking point (for a proof see [HP91]).

In the case of delating some constraints an oracle which achieves most pruning without losing any solution is chosen. The basic scheme for deletions proceeds with oracle O in the same way like for additions. Note that the addition of constraints must be now refined to pick up the best oracle. While the last oracle was the best when only additions were concerned it is no longer true when additions are allowed.

The re–execution approach presented in [HP91] outperforms incremental search based on the backtrack search. However, it should be noted that the same pruning can be achieved without using an oracle, simply by the replacing the re–execution scheme by a backjumping mechanism (for more information on backjumping see [Tsa93]).

2.2.2 Local repair methods

The second class of algorithms which deals with inconsistencies of a schedule caused by dynamic changes in a problem is based on the local repair methods. The basic idea behind this approach can be described as follows. Any solution which became infeasible because of addition or deletion of constraints, may be gradually impoved until the new solution is acceptable.

An example of a method using local repair methods was presented by Verfaillie and Schiex in [VS94b]. The main idea behind this algorithm can be intuitively described in following way. Activities which violate some constraint may be removed from the schedule. All removed activities can be then gradually re–entered into the schedule. Re–entering an activity t into a schedule is performed even if t can be placed in such way that all the activities incompatible with its location can be removed and gradually re–entered, without modifying location of the t.

More formally, given a CSP, let A be a consistent assignment of a subset Z of variables. Let v be a variable which does not belong to Z. The variable v can be assigned, i.e. a consistent assignment $Z \cup \{v\}$ can be obtained, if and only if there exists a value val such that val can be assigned to v, then all the assignments $< v', val' >$ which are incompatible with assignment $< v, val >$ can be unassigned and reassigned one after another without modifying the assignment of v.

An evaluation of the algorithm was conducted on randomly generated CSPs [VS94b]. The local changes procedure was used together with the simple heuristics for choosing variables and values for assignment. When choosing a variable for assignemt the variables with smallest domains are chosen first.

The evaluation shows that the algorithm performs best on the least and most constrained problems. Nevertheless, the algorithm is inefficient on intermediate strongly constrained problems.

Another heuristic repair method called **min–conflicts** was presented by Minton et al. in [MJPL92]. Given a DCSP $\mathcal{P} =< \Theta_1, \ldots, \Theta_n >$, let α_i be a solution to the Θ_i and Θ_{i+1} a CSP which differs by the set of constraints added or removed from the problem. To find a solution to Θ_{i+1} the method starts with the initial, complete but inconsistent assignment, which is this case is a solution to the proceding problem. It is said that two variables are in *conflict* if their values violate a constraint. At each choice point during the search, the heuristic chooses an arbitrary variable which is currently in conflict and, if possible, reassigns its value, until a solution is found. The system thus searches the space of possible assignments, favouring assignments with fewer total conflicts.

The described mathod is usally used with the hill climbing search, presented in Algorithm 2.1. The min–conflics heuristics is then applied at every search step of the hill climbing algorithm. The variable $conflicts$ is a measure of a number of conflicts.

Algorithm 2.1 Hill climbing

```
Require: a problem
  current ← Initial state (problem)
  loop
    next ← a highest value successor of current
    if conflicts[next] > conflicts [current] then
      return current
    else
      current←next
    end if
  end loop
```

The hill climbing procedure combined with minimum conflicts heuristics was successful in handling dynamical changes in scheduling observations of the Hubble telescope and another problems [MJPL92]. Nevertheless, several factors limit the applicability of the method. The most important factor is that all the conflicts are assumed to be independent. For highly structured problems minimum conflicts heuristics performs poorly. Moreover, since this method ignores all fine structure in the problem, there exists possibility of pathological configurations occuring during the search procedure. This manifests itself for example in “cycles” where the same variable is repaired again and again without any progress towards finding a solution.

2.2.3 Constraint recording methods

The third group of algorithms which deal with dynamic DCSPs includes the methods which record earlier solutions (information) and reuse them to solve subsequent problems. Some of the methods mentioned before, like incremental search presented in [HP91], depends on recorded information.

An example of an algorithm which uses information recorded during the search is *Dynamic Backtracking(ddbt) for DCSPs.* This method was introduced by Verfaille and Schiex in [VS94a]. The algorithm is extension of *dynamic backtracking(dbt)* presented by Ginsberg in [Gin93]. The dynamic backtracking algorithm (dbt) is presented first.

The dynamic backtracking algorithm, Algorithm 2.2, is based on the idea of recording information about earlier search phases. At each step of the search, for every variable v and each value val, which has been eliminated from the current domain of v, sets of previously assigned variables responsible for this elimination are recorded. These sets are called *eliminating explanations* and require $O(n^2d)$ space to be recorded, where n is a number of variables and d is the maximum domain size. The *conflict sets* of a variable is the union of eliminating explanations and all its eliminated values. If a domain of a variable v is empty and v' is a last variable recorded in the conflict set of variable v then the algorithm backtracks to v', unassigns it but does not assign any new variable. Then, all the eliminating explanations which contain variable v' must be removed and their values need to be returned to their corresponding current domains. Moreover, new eliminating explanations are created for the value val' previously assigned to v'.

Algorithm 2.2 Dynamic Bactracking

```
Procedure: dbt($\mathcal{P}$)
  Require: CSP $\mathcal{P} = (Z, D, C)$
  return dbt-variables($\emptyset, Z$)
.................................................................
Procedure: dbt-variables($Z_1, Z_2$)
  Require: $Z_1$ is a set of assigned variables
  Require: $Z_1$ is a set of unassigned variables
  if($Z_2 = \emptyset$)then
    return success
  else
    choose $v \in Z_2$
    $d = D(v)$
    if dbt-variable($Z_1, v, d$)= failure then
      return dbt-bt-variable($Z_1, Z_2, v$)
    else
      return dbt-variables($Z_1 \cup \{v\}, Z_2 - \{v\}$)
    endif
  endif
```

Procedure `backward-checking`(Z, v) checks the current value of v against the current values of the variables of Z and returns the first failing constraint, if any, or *success*, if none. Procedure `create-eliminating-explanations`(v, val, Z) records the set Z of variables as an explanation for the elimination of the value val from the current domain of the variable v. For all the variables in Z, `remove-eliminating-explanations`(v, Z) removes the eliminating explanations which contain v and returns the corresponding values in their corresponding current domains.

The Dynamic Backtrack algorithm can be easily extended to take into account constraints in eliminating explanations. This requires adjusting the `dbt-value` and `dbt-bt-values` procedures. These procedures are modified as in Algo-

Algorithm 2.3 Variable and value choosing procedures

```
**Procedure:** dbt-variable($Z_1, v, d$)
  if $d = \emptyset$ then
    return $failure$
  else
    $val = d_x \in d$
    if dbt-value($Z_1, v, val$)$= success$ then
      return $success$
    else
      return dbt-variable($Z_1, v, d - \{val\}$)
    endif
  endif
.................................................................
Procedure: dbt-value($Z_1, v, val$)
  assign-variable($v, val$)
  $c =$backward-checking($Z_1, v$)
  if $c = success$ then
    return $success$
  else
    $Z_3$ = set of variables of $c$
    unassign-variable($v$)
    create-eliminating-explanation($v, val, Z_3 - \{v\}$)
    return $failure$
  endif
.................................................................
Procedure: dbt-bt-variable($Z_1, Z_2, v$)
  $Z_3$ = conflict set of $v$
  if $Z_3 = \emptyset$ then
    return $failure$
  else
    $v'$ = last variable of $Z_3$ in $Z_1$
    $val'$ = current value of $v$
    $Z_4$ = set of variables following $v'$ in $Z_1$
    unsign-variable($v'$)
    create-eliminating-explanations($v', val', Z_3 - \{v'\}$)
    remove-eliminating-explanations($v', Z_4 \cup Z_2$)
    dbt-variables($Z_1 - \{v'\}, V_2 \cup \{v'\}$)
  endif
```

rithm 2.4.

The eliminating explanations in Algorithm 2.4 are composed of two parts: the set of previously assigned variables (the *variable eliminating explanation*) and the set of constraints (the *constraint eliminating explanation*). The procedure `create-eliminating-explanation`(v, val, Z, C) records the set Z of variables and the set C of constraints as an explanation for the elimination of the value val from the current domain of variable v.

Algorithm 2.4 Modified Dynamic Backtracking

```
Procedure: dbt-value($Z_1, v, val$)
  assign-variable($v, val$)
  $c$ =backward-checking($Z_1, v$)
  if $c = success$ then
    return $success$
  else
    $Z_3$ = set of variables of $c$
    unassign-variable($v$)
    create-eliminating-explanation($v, val, V_3 - \{v\}, \{c\}$)
    return $failure$
  endif
.....................................................................
Procedure: dbt-bt-variable($Z_1, Z_2 v$)
  $Z_3$ is a union of all the variable eliminating explanations of $v$
  $C$ is a union of all the constraint eliminating explanations of $v$
  if $Z_3 = \emptyset$ then
    return $C$
  else
    $v'$ is last variable of $Z_3$ in $Z_1$
    $val'$ is its current value
    $Z_4$ is the set of variables following $v'$ in $Z_1$
    unassign-variable($v'$)
    create-eliminating-explanation($v', val', Z_3 - \{v'\}, C$)
    remove-eliminating-explanations($v', Z_4 \cup Z_2$)
    dbt-variables($Z_1 - \{v'\}, Z_2 \cup \{v'\}$)
  endif
```

Solving CSPs using the Dynamic Backtracking algorithm produces the following:

- when the CSP is *consistent*:
  - a *solution*, i.e. a complete and consist assignment;
  - a set of *eliminating explanations* related to possible variable assignments.
- when the CSP is *inconsistent*:
  - a partial consistent assignment, which can be empty;
  - an *inconsistency explanation*

– a set of *eliminating explanations* related to some of the possible variable assignments.

These results can be recorded and reused when solving new CSP differs from the previous one by the sets of added and deleted constraints (see [VS94a]). It can often be expected that most of the recorded explanation remain valid for the new CSP. This explanation recording imported from a previous CSP should allow significant pruning of the search space of the new CSP. The new algorithm with necessary adjustments for Dynamic CSPs is presented as Algorithm 2.5.

Algorithm 2.5 Dynamic Backtracking for DCSPs

```
Require: $Z_1$ is a set of assigned variables, result of the previous search
Require: $Z_1$ is a set of assigned variables, result of the same search
Require: $C_{add}$ is a set of added constraints
Require: $C_{del}$ is a set of deleted constraints
  $Z = Z_1 \cup Z_2$
  $Z_3 =$ remove-assignments($Z_1, C_{add}$)
  remove-variable-eliminating-explanations($Z_3, Z$)
  remove-constraint-eliminating-explanations($C_{del}, Z$)
  return dbt-variables($Z_1 - Z_3, Z_2 \cup Z_3$)
```

The three procedures `remove-assignments`,`remove-variable-eliminating--explanations` and `remove-constraint-eliminating-explanations` aim at deriving a consistent point for the new search, the procedure `dbt-variables`.

For each constraint $c \in C_{add}$, let $Z(c)$ be a set of its variables. If the assignment of the variables of Z_1 violates c, the procedure `remove-assignments(`Z_1, C_{add}`)` unassigns the last variable of $Z(c) \in Z_1$ and creates the corresponding eliminating explanation. The procedure returns the set Z_3 of the variables which have been unassigned.

Experimental evaluations of the algorithm show that Dynamic Backtracking for DCSPs run on the randomly generated benchmarks for dynamic problems with small and intermediate changes clearly outperforms other algorithms based on recording previous assignments. The fact that the algorithm does not seem to perform well on problems including major problem configuration changes is not surprising. Since Dynamic Backtracking for DCSPs relies on the eliminating explanations for previous CSP, major changes can make them completely incompatible with the new problem.

2.3 Recurrent Dynamic Constraint Satisfaction Problem and Stable Solutions

As mentioned in Section 2.2 a dynamic CSP model can be used to capture changes in the scheduling environment but it assumes that those changes are permanent. However, there are situations where changes in the current scheduling problem are only temporary. Machine breakdown and employee absence are examples of such temporary changes.

The sequential nature of DCSP is a major drawback when using DCSP to model scheduling environments where temporary changes occur. Because DCSP is a sequence of CSPs then any change which causes infeasibility of an executed schedule demands solving a new CSP. Such a situation is highly undesirable. Ideally, one wouldn't like to lose the original solution in the first place, which would forestall all extra search and other undesirable aspects of solution failure.

To remedy this Freuder and Wallace introduce the notion of Recurrent Dynamic CSP (RDCSP). This model takes into consideration temporary, recurrent changes to the problem [WF98, WF99]. The aim of the model is to find solutions which are *stable*, i.e. which remain valid after the problem was altered.

The following criteria were used for evaluating the performance of the model:

1. efficency in finding a new solution,
2. solution similarity or consistency, i.e. a new solution should share as many values as possible with the old one, and
3. solution stability in face of problem alternation.

Since changes to the problem are unpredictable and can occur during a search, i.e. before a new solution, which replaces the lost one, is found, a heuristic repair method was chosen. This method is combined with min–conflicts heuristics extended by a random walk strategy. Min–Conflicts heuristic is defined as follows [MJPL92]:

Definition 2.15
Given: *A set of variables, a set of binary constraints, and an assignment specifying a value for each variable. Two variables* ***conflict*** *if their values violate a constraint.*
Procedure: *Select a variable that is in conflict, and assign it a value that minimizes number of conflicts. Break ties randomly.*

A random walk strategy makes an assignment to the variable in conflict regardless of whether the new assignment is better or worse than the previous one. The assignment is make according to the fixed probability. Randomization of assignment aims at getting min–conflicts out of local minima. This method is reported to be effective in finding global minima for random CSP [Wal98].

Using hill climbing, which is a generall iterative repairment technique, also makes it possible to incorporate the changes in the scheduling problem which appear during search for new solution.

The general hill climbing algorithm was presented in the Algorithm 2.1 in Section 2.2.2. The Algorithm 2.6 presents hill climbing technique for stable solutions.

The basic procedure of search must be extended by collecting information about changes in environment and incorporating those changes into the search procedure. Before we describe how it is done we need to present the basic properties of RDCSP. They are as follows:

- Values and/or variables may become insignificant or inconsistent and constraints may be added temporarily.
- There are differences in likelihood of changes which can occure,
- Those differences are assumed to be known *a priori.*

Algorithm 2.6 Basic hill climbing for stable solutions

```
{PREPROCESSING}
  For each successive variable
    choose a value that minimizes conflicts with variables already assigned

{HILL CLIMBING }
  repeat
     1. randomly select variable $x$ with $\geq 1$ conflicts
     2. with probability $p$
          choose value $v$ at random from the domain of $x$
    2'. with probability $1 - p$
          find all min–conflicts values in domain of $x$
          choose value $v$ at random from min–conflicts set
     3. assign $v$ to $x$
  until cutoff time reached or complete solution found
```

Collecting information about the changes will help to avoid choosing values for assignment that are likely to make the current solution invalid in the future. Information about changes is collected as they occur and is used to decide which value to assign to a given variable during subsequent search. The procedure of gathering information can be carried out in following ways:

1. changes may be traced directly by e.g. recording inconsistent values, or
2. solution loss resulting from the problem change may be traced and related to the values participating in solutions of varying stability.

The method described in [WF98] tracks changes by counting the number of times that an element changes. Those counts can be related to relative frequencies of changes, which in turn gives information about the probability that given change will occur in the future. Another essential feature of counting changes is that it provides a total ordering among the variables, whose assignments have become inconsistent.

Methods for incorporating gathered information into the search consider two cases.

The first one is that of value loss since the count of value loss, or "penalties" relate directly to min–conflict procedure. The method adds the penalty for each value to its current conflict at the point where conflicts of the values in domain are compared. This is called `penalty-adding`, Algorithm 2.7a.

Since using penalties in their original form can bias the hill climbing procedure heavily, they are rescaled relatively to the penalty of the variable being repaired.

The second method, `penalty selection` in Algorithm 2.7b, works as follows. After the set of min–conflicts is chosen in standard way, the value from this set is chosen at random, but the assignment is delayed unless its penalty is less than some criterion or cutoff value. This process is repeated until an acceptable value is found or all values have been tested.

In both cases, a selection or cutoff method is used with the random walk, i.e. if a value was chosen at random, then values are chosen successively and at random from the remaining, untested subdomain until one that matches the selection criterion is found. If no such value is found the random walk procedure is terminated.

Algorithm 2.7 Penalty ajustment for min–conflicts heuristics

```
a.PENALTY–ADDING
  2'. with probability $1-p$
      find all values in domain of $x$ with minimum (conflicts + scaled penalty)
        choose value $v$ at random from this biased min–conflicts set

b. PENALTY SELECTION
  2'. with probability $1-p$
        find all min conflicts values in domain of $x$
        **repeat**
          select and remove value $v$ at random from min–conflicts set
        **until** for $v <$ cutoff or no more values
        **if** penalty for $v <$ cutoff **then**
          choose $v$ for assignment
        **else**
          assign a value chosen at random from original min–conflicts set
        **endif**
```

To avoid values that violate potential constraints with high penalties the constraint penalties can be incorporated into hill climbing by checking them during value selection, Algorithm 2.8. In this case, a value chosen from the min–conflict set is tested against potential constraints which are not part of the current problem. If none of the potential constraints are violated or if the penalties of violated constraints do not reach the cutoff value, then the tested value is accepted. Otherwise, another value is chosen. If no acceptable value was found, a value from min–conflicts set is chosen at random.

Such a penalty–based selection may in certain situation lead to overconstraining the hill–climbing procedure. In the experimental work presented in [WF98], the selection criterion was relaxed each time the hill–climbing procedure failed to find a solution after k seconds.

Wallace and Freuder present the experimental evaluation of stable solution startegies in [WF98]. They were tested with simulated Recurrent DCSP on random 3–color problems with 100 variables. For those RDCSP probability of value loss was set to high probability $p_h = 0.3$ and low probability $p_l = 0.003$. The probability that a domain value was associated with the higher probability of loss, p_p was set to 0.025. In the runtime phase there was 250 occasions of change in which the entire set of relevant elements was scanned deciding for each element if any change should be made to original problem. Both changes to the problem when hill climbing finds a new solution as well as periodical changes at fixed intervals regardless search state was investigated.

In the case of value loss with the values penalized after each deletion both penalty startegies guided the hill climbing to a solution with appreciably fewer "bad" solutions. In addition, adding a penalty to the number of conflicts resulted in solutions of one magnitude better than without penalty. Nevertheless,

Algorithm 2.8 Adjustment to min–conflicts caused by constraint addition

```
SELECTION BASED ON CONSTRAINT PENALTIES
  2'. with probability $1 - p$
      find all min conflicts values in domain of $x$
      repeat
         select and remove value $v$ at random from min–conflicts set
         for each constraint not currently present
            if value is not compatible with other value(s) in constraint
            and constraint penalty > cutoff then
               reject value and exit for loop
            endif
      until $v$ not rejected or no more values
      if $v$ not rejected then
          choose $v$ for assignment
      else
          assign a value chosen at random from original min–conflicts set
      endif
```

adding penalties was showed to be more costly in terms of run time. In contrast, for the penalty selection strategy there were no consistent differences between penalty and non penalty conditions across problems. However, the large difference in run time favours the penalty condition as often as the control.

Compared to "restarting" the solver after assignment loss, solution reuse was very successful in avoiding bad values in solutions.

Experiments on using constraint addition procedure were carried out using the same problem setup. The average degree of improvement in this case was somewhat better than with value deletion, but the mean run time was 2–3 times greater under penalty conditions. Solution similarity was high in both penalty and non–penalty conditions, with the avarage difference by 0.01 in each tested case.

Experimental evaluation shows that the penalty function improves the general measure of solution quality. Nevertheless, there exists a tradeoff between solution quality and a run time. Reported experiments show, that the same problem configurations run under penalty condition has the mean run time 2–3 times greater. Solution similarity was high in both penalty and non–penalty conditions, differing at average by 0.01 in each case.

It was also concluded that tracking the value loss is an efficient way to gather information for finding stable solutions. Even in the situations where assumption about convergence of underlying probabilities are clearly violated penalties served as a bias for hill climbing procedure led in the direction of more stable solutions.

Finally, it can be concluded, that the stable solution framework is the only attempt within CSP framework to gather information about the frequence and the nature of the changes in the problem and use it later to solve it. The idea of guiding a search for solution to altered problem by using the information about probability of the changes may be used to effectively prune the search tree of the problem.

Chapter 3

Dynamic scheduling problems

In this chapter we will show how models presented in Chapter 2 can be used to model some of the scheduling problems. First, a generalized model for static scheduling problems, Kernel Resource Feasibility Problem, is presented. Then, the Minimal Perturbation Problem, used to create a minimally disruptive schedule in dynamic environment is described.

3.1 Kernel Resource Feasibility Problem

Different classes of scheduling problems can be generalized as a kernel resource feasibility problem (KRFP)[EKR96]. In the KRFP the goal is to fix start and end times of activities such that the quantities of available resources are not over–allocated.

3.1.1 Basic definitions

Definition 3.1 *A **Kernel Resource Feasibility Problem** is a 5-tuple* $(\mathcal{A}, \mathcal{R}, quantity, \mathcal{TC}, T_{max})$*, where* $\mathcal{A}$ *is a set of* n *activities,* $\mathcal{R}$ *is a set of* m *resource types* $r_1, \ldots, r_m$ *, quantity is a resource function* $quantity : \mathcal{R} \rightarrow \mathbb{N}$*,* $\mathcal{TC}$ *is a set of temporal constraints and* $T_{max} \in \mathbb{N}$ *is a latest point of scheduling horizon.*

The KRFP contains three major components: activities, resources and time constraints.

The activities of the KRFP are represented as a set of atomic activities $\mathcal{A} = \{A_1, \ldots, A_n\}$. The following assumptions characterize activities of KRFP:

1. they require only one type of resource;
2. they are non–interruptible but the duration may vary;
3. they consume a quantity of resource which does not change through the duration.

Each activity $A_i \in \mathcal{A}$ consists of five elements: r_i which is the name of the required resource, the resource area variable $area_i$, the variable $quantity_i$ representing used quantity of resource r_i, the discrete start point variable s_i and the discrete end point variable e_i.

Resources of KRFP are described by:

1. A set of $\mathcal{R}$ of resource types.
2. A function $quantity : \mathcal{R} \to \mathbb{N}$.

Temporal constraints ($\mathcal{TC}$) are of one of the following forms:

1. $u\ R\ c$ (bounding constraints)
2. $u\ R\ v \pm c$ (distance constraints)
   where: $R \in \{=, <, >, \leq, \geq\}, u, v \in \bigcup\limits_{a_i \in A} \{s_i, e_i\}$

A solution to KRFP is an assignment of values to the variables in $\mathcal{A}$ where the following constraints are satisfied:

- The activity constraints:

$$\forall A_i \in \mathcal{A} : area_i = quantity_i \times (e_i - s_i) \tag{3.1}$$

- The temporal constraint in $\mathcal{TC}$, including the constraints relating the start and end time of each activity, as well as constraints enforcing schedule horizon: $\{0 \leq s_i \leq T_{max}, 0 \leq e_i \leq T_{max} : A_i \in \mathcal{A}\}$.
- Let $\forall r \in \mathcal{R} : \mathcal{A}_r = \{A_i : A_i \in \mathcal{A} \wedge r_i = r\}$. The resource constraints:

$$\forall r \in \mathcal{R}, \forall t \in \{0, \ldots, T_{max}\} : \ quantity(r) \geq \sum_{A_i \in \mathcal{A}_r \wedge s_i \leq t < e_i} quantity_i \tag{3.2}$$

3.1.2 Representing KRFP as a Constraint Satisfaction Problem

The activity constraints of (3.1) and the temporal constraints $\mathcal{TC}$ can be directly represented as a CSP constraints according to the model presented by El–Kholy and Richards in [EKR96].

In this model, a new resource quantity variable Q_r is introduced for every $r \in \mathcal{R}$. This variable corresponds to the maximum quantity of this resource used over the schedule horizon. For each activity $A_i \in \mathcal{A}_r$, a variable Q_{rs_i} counts the quantity of resource r used at start time s_i. Variable Q_r denotes the maximum of these quantities $Q_r = \mathsf{MAX}\{Q_{rs_i} : A_i \in \mathcal{A}_r\}$. The variables of the Q_{rs_i} are defined in terms of Booleans. A Boolean $B_{s_i A_j}$ is introduced for each pair of activities $A_i, A_j \in \mathcal{A}_r$ and denotes a situation when activity A_j overlaps with s_i.

$$\forall r \in \mathcal{R}, \forall A_i, A_j \in \mathcal{A}_r : B_{s_i A_j} = \begin{Bmatrix} 1 & \mathsf{iff}\ s_j < s_i \wedge s_i < e_j \\ 0 & \mathsf{otherwise} \end{Bmatrix} \tag{3.3}$$

Linked with the corresponding Q_{rs_i}, these Boolean variables link temporal and resource reasoning and the following constraint can be applied:

$$\forall r \in \mathcal{R}, \forall A_i in \mathcal{A}_\nabla : Q_{rs_i} = \sum_{A_j \in \mathcal{A}_r} B_{s_i A_j} \cdot quantity_j \tag{3.4}$$

Even if each Q_{rs_i} is bounded directly by the maximum resource quantity $quantity(r)$ via the constraints $Q_{rs_i} \leq Q_r$ and $Q_r \leq quantity(r)$, introducing variables Q_r will allow us to modify the KRPF by adding constraints.

3.2 Minimal Perturbation Problem

A problem which often arises in a dynamic scheduling environment is creating a minimally disruptive schedule, where disruptiveness is measured as the distance between two solutions. In the case of scheduling under resource constraints this problem can be transformed into a Resource Utilization Problem (RUP). A hybrid method for solving this problem was presented by El Sakkout et al. in [SRW97, SW00, SRW98].

Definition 3.2 *A* ***minimal perturbation problem*** Π *is a 5-tuple (* Θ, α_Θ, C_{del}, C_{add},δ*) where* Θ *is a CSP,* α_Θ *is a solution to* Θ, C_{del} *and* C_{add} *are constraint removal and addition sets and* δ *is a distance function, which evaluates the distance between two solutions. This function is used to measure the degree of perturbation.*

A complete assignment is a solution to Π if and only if it is a solution to $\Theta' = (Z, D, C_{\Theta'})$, where $C_{\Theta'} = (C_\Theta \setminus C_{del}) \cup C_{add}$. A solution is optimal if and only if $\delta(\alpha_\Pi, \alpha_\Theta)$ is minimal.

In the context of Definition 3.2 a Resource Utilization Problem may be defined as follows:

Definition 3.3 *A* ***Resource Utilization Problem*** *is a minimal perturbation problem* $(\Theta_i, \alpha_i, C_{del}, C_{add}, \delta)$*where:*

- Θ *is any KRFP* $(\mathcal{A}, \mathcal{R}, \mathcal{TC}, T_{max})$ *modeled as a CSP, such that*
  - *there is only one resource type* $r(R = \{r\})$ *and* Q_r *is the resource quantity variable for that resource type*
  - $\forall A_i \in \mathcal{A} : quantity_i = 1$
- α_Θ *is a solution to* Θ*;*
- $C_{del} = \emptyset$, $C_{add} = \{Q_r \leq c\}$*, such that* $c < quantity(r)$*;*
- $\delta(\alpha_{\Theta'}, \alpha_\Theta) = \sum_{u \in \bigcup_{a_k \in A} \{s_k, e_k\}} |\alpha_{\Theta'}(u) - \alpha_\Theta(u)|$

The definitions above model the problems in terms of Dynamic Constraint Satisfaction Problems described in Section 2.2 and create the base of an array of methods described in Chapter 4. However, as mentioned in Section 2.3, the Dynamic CSP model is based on the assumption of permanent changes in the scheduling environment. Some of the instances of MPP, like a rescheduling caused by machine failure or by delays in a public transportation system, can contain only temporary changes. It seems more natural to model those cases using the Recurrent Dynamic CSP paradigm. However, as far as we know, there is no, investigation on modeling MPP in the context of recurrent dynamic CSPs.

Chapter 4

Probing methods

Most real world scheduling problems which has been addressed in recent years, including those coping with dynamic changes, have characteristics of large scale, complex, optimization problems. An attempt has been made to address those problems by decomposing them into multiple subproblems. Since the optimal solutions of different subproblems are invariably incompatible with each other ways of solving subproblems, which make such sub–solutions globally consistent, are being explored. This research topic belongs to an area known as "hybrid algorithms".

This chapter presents most of the work done on hybrid algorithms in the context of dynamic scheduling. The algorithms are complete tree search algorithms which, at every search node, implement a repair step. The repair is performed on *super–optimal* assignments, i.e. assignments which are optimal with respect to the objective function, but which are only partially consistent.

Since the probing method evolved from constraint backtrack we start with a presentation of this method.

4.1 Constraint Backtrack

The constraint backtrack algorithm presented in [SW00] is an extended version of the resource feasibility algorithm from [Pot97]. The algorithm contain an aspect of temporal optimization not present in [Pot97]. The outline of the Constraint Backtrack algorithm is presented here as Algorithm 4.1.

The backtracking procedure is initiated by parameter `MonitoredConstrs`, which is a set of violated constraints. The algorithm is based on simple depth–first search which calls `PushConstrStore` and `PopConstrStore`. The `PushConstrStore` pushes a decision represented by a constraint onto a constraint store stack and triggers the local consistency propagation. The `PopConstrStore` undoes the decision and the propagation based on it.

Constraints which are subject to contention are filtered by the `constraint_filter` procedure. Filtered out constraints have no impact on resource feasibility in this search branch and are omitted. If all the conflicts are removed the resource feasibility phase passes control over to the temporal optimization phase.

The procedure `select_constraint` selects from the filtered constraint set `ContentionConstraints` the constraint `Constr` with the greatest potential for

Algorithm 4.1 Constraint Backtrack

```
Require: Set of monitored constraints MonitoredConstrs
  ContentionConstrs = constraint_filter(MonitoredConstrs)
  if ContentionConstrs = $\emptyset$ then
    return TRUE
  else
    Constr = select_constraint(ContentionConstrs)
    Decision = select_decision(Constr)
    if PushConstrStore(Decision) then
      if constraint_backtrack(MonitoredConstrs) then
        return TRUE
      end if
    end if
    PopConstrStore
    if PushConstrStore not(Decision) then
      if constraint_backtrack(MonitoredConstrs) then
        return TRUE
      end if
    end if
    PopConstrStore
    return FALSE
  end if
```

conflicts. At this point the objective is to reduce conflicts in `Constr`. On backtracking the ordering constraint of `Decision` is revoked and replaced by its negation. If the negation also fails then it is revoked and the algorithm backtracks to an earlier decision.

The algorithm applies arc–B consistency propagation on arithmetic constraints, see Section 2.1. The decision is propagated by an interplay between resource and temporal constraints. The algorithm applies an additional look–ahead check: after some resource was not found to be a subject of contention at the latest search decision, a resource usage profile is built for the time horizon interval which given activity must span over. Other activities are assumed to take minimal span over the same interval. If the resource usage exceeds the capacity at any time point in this interval, a failure is signaled and the algorithm backtracks to the previous decision point. This procedure was found to be important for solving KRFP problems, which relies on computing resource overlap at activity start times.

The empirical experiments and comparisons with other techniques (see [SW00]) show that Constraint Backtracking algorithm is quite ineffective at obtaining the minimum and proof of optimality. The propagation methods described above fail to prune efficiently and the objective function does not reduce the search space in a satisfying manner.

The experiments find Constraint Backtracking relatively ineffective in finding the optimum and prove of optimality. The propagation methods are shown to fail in performing efficient pruning in the search tree. In particular, since implications of individual search decisions with respect to the cost are not discovered until most of the variables are fixed, the optimization function fails to efficiently reduce the search space.

4.2 Probe Backtrack Algorithms

Probe backtracking is an extended version of constraint backtracking, where the backtrack search procedure is supported by look–ahead procedures, so called *probe generators*, generating potentially good assignments, *probes*.

The purpose of creating a probe is to direct backtrack search and limit the size of the search space. After the probe is created the search concentrates on regions where the probe violates constraints. The main difference between the constraint backtrack search and the probe backtrack search is that the probe backtrack search calls the procedure `obtain_probe_assignment`. It represents a call to the probe generator used to focus search decision at this search node. The only conditions which must be guaranteed by the prober is satisfying the variables' domain constraints. It is preferable that assignments generated by the prober are of good quality, usually super–optimal with respect to the objective function if such a function exists.

Algorithm 4.2 Probe Backtrack Search

```
Require: Set of monitored Constraints Monitored Constrs
  Assignment = obtain_probe_assignment
  ViolatedContentionConstrs = constraint_filter(MonitoredConstrs,Assignment)
  if ViolatedContentionConstrs == $\emptyset$ then
    return TRUE
  else
    Constrs = select_constraint(ViolatedContentionConstrs,Assignment)
    Decision = select_decision(Constr,Assignment)
    if PushConstrStore(Decision) then
      if probe_backtrack(MonitoredConstrs) then
        return TRUE
      end if
      PopConstrStore
      if PushConstrStore not(Decision) then
        if probe_backtrack(MonitoredConstrs) then
          return TRUE
        end if
        PopConstrStore
        return FALSE
      end if
    end if
  end if
```

The result returned from the probe makes it possible to filter and select monitored constraints. Similarly to the filtering process in constraint backtracking, constraints which are not subject to a conflict are filtered out. Moreover, any constraint which is satisfied by a tentative assignment returned by the prober is removed from the set of violated constraints.

To exemplify the filter mechanism consider the Resource Utilization Problem from Section 3.2. Let $X[Dv]$ to represent the variable X with domain D and a probe value v. A constraint

$$Q_{rs_i}[\{0..1\}1] = B_{s_i\mathcal{A}_j}[\{0,1\}0] + B_{s_i\mathcal{A}_k} : [\{0,1\}0] + B_{s_i\mathcal{A}_l}[\{0,1\}1]$$

is removed even if in contention, because its satisfied by the probe assignment.

Even if constraints satisfied by current probe may be later invalid, it is assumed that the current probe will lead to a solution. If all constraints are satisfied after the probe a solution is found and probe backtrack terminates.

Otherwise a decision is made about the constraint `Constr` which has been violated by the probe assignment to force new probes and obtain assignments that are closer to satisfying `Constr`. Selecting decisions that allow the prober generator to return the same probe would not progress the search.

Some necessary heuristics are used to select a decision about choosing an appropriate constraint `Constr`. In RUP studied in [SW00] contention reduction is achieved by forcing apart two temporal variables in the fashion similar to `CB`. The secondary heuristics is to order decision constraints using least–commitment or estimating the impact of a new ordering constraint on the assignment returned by the probe.

The probe backtracking algorithm generates a probe closely based on the old solution, which is only partially consistent with the new problem constraints. With every call to `obtain_probe_assignment` the local consistency propagation prunes the domain of the temporal variables. Since `MonitoredConstrs`' parameter includes only resource constraints the temporal constraints are not monitored for violation, which makes pure probe backtracking algorithm incomplete.

Completeness of the algorithm is restored by introducing an intermediate phase between the probe backtracking and temporal optimization phases to relieve any remaining contention from resources that were filtered out.

Nevertheless, since the number of temporal constraints increases at every decision step the probes generated in the probe backtracking phase may be of little of none relevance because the tentative values for remaining variables, even if still super–optimal, may violate so many temporal constraints that they no longer provide accurate information about feasible solution.

4.3 Unimodular probing

The unimodular probing algorithm combines constraint programming techniques such as constraint propagation with linear programming optimization. It relies on a constraint programming platform which can invoke an LP solver and extract the information about the optimal solution computed by the LP solver.

The Linear Programming solver can handle only a set of linear constraints and treats all its variables as continuous, relaxing any discrete constraints. It computes an optimum value of a given objective function and records the values of the variables in this optimum.

The earlier itegrations of constraint programming and linear programming, like those described in [BB93] and [RWH98], pass the whole set of constraints into the LP solver. The unimodular probing algorithm, in contrast, passes to the LP solver only a restricted class of constraints which has a special property called *total unimodularity.* The consequence of this property is that the optimum solution is guaranteed to be integer–valued. A set of constraints with total unimodularity property is referred to as an "easy set".

Below we define total unimodularity and describe its characteristics. It follows definitions given in [NW88], but it can be found in any book on operations research.

Definition 4.1 *An mn matrix A is* ***totally unimodular*** *(TU) if the determinant of each square submatrix of A is equal to* 0,1 *or* -1.

Let $P(b)$ represent the polyhedron of solutions to the set of linear inequalities $Ax \leq b$.

Theorem 4.3.1 *If A is totally unimodular, then $P(b) = \{x \in \mathbb{R}^n_+ : Ax \leq b\}$ is integral for all $b \in \mathbb{Z}^n$.*

The proof of the Theorem 4.3.1 may be found in e.g. [NW88].

This property is applied by unimodular probing algorithm on an easy set of constraints to assure that the solution returned by the probe is an optimal integer solution. The probing procedure proceeds by applying local consistency techniques to the constraints of the problem. After a solution was found by the prober, the search procedure generates the constraints in the full set violated by the unimodular probe. If there are no violated constraints then the search found a solution. Otherwise, the algorithm selects a violated constraint, and rules out the present unimodular probe by imposing an additional easy constraint. If a choice of possible repairs exists then the algorithm backtracks through these choices in its search for optimality. When a repair has been selected, local consistency methods derive further easy constraints which follow from the choice. We will explain the propagation mechanism on KRFP example later on.

The Algorithm 4.3 gives an outline of the unimodular probe algorithm.

Below we give an example how the unimodular probing algorithm can be applied on the Resource Feasibility Problem defined in Section 3.2.

The first task of constructing a unimodular probing algorithm is the identification of a suitable easy set with the total unimodularity property. In a Linear Programming problem, the set of constraints and variables is represented by a constraint matrix, where the rows represent constraints, and columns represent variables, with the element e_{ij} of the matrix denoting the coefficient of variable j in a particular constraint i.

Sufficient conditions for unimodularity of constraint matrix are as follows:

1. All the variable coefficients are 0,1,-1,
2. No more than two nonzero coefficients appear in each row (column).
3. The columns (rows) of the matrix can be partitioned into two subsets Q_1 and Q_2 such that:
    (a) If a row (column) contains two nonzero coefficients with the same sign, one element is in each of the subsets.
    (b) If a row (column) contains two nonzero elements of opposite sign, both elements are in the same subset.

For a proof see [NW88].

The temporal constraints of KRFP satisfy the sufficient conditions for the TU property if the set of variables (columns) is partitioned into a subset S_1 containing all temporal variables and an empty subset S_2. Since the characteristics and representation of $\mathcal{TC}$ is identical for KRFP and RUP, then the temporal

Algorithm 4.3 Unimodular probing search

```
Require: Set of new constraints to be added $C_{new}$
  $C_{cpnew}$ = push_cp($C_{new}$)
  if ($C_{cpnew} \neq FALSE$) then
    $C_{lpnew}$ = obtain_easy_constraints($C_{new} \cup C_{cpnew}$)
    $S_{lp}$ = push_lp($C_{lpnew}$)
    if ($S_{lp} \neq FALSE$) then
      $C_{conf}$ = violated_constraints($S_{lp}$)
      if ($C_{conf} = \emptyset$) then
        $S_{lp}$
      else
        $C_{repair}$ = select_repair_choice($C_{conf}, S_{lp}$)
        $S_{lp}$ = unimodular_probing_search($C_{repair}$)
        if ($S_{lp} \neq FALSE$) then
          return $S_lp$
        else
          $S_{lp}$ = unimodular_probing_search($NOT(C_{repair})$)
        end if
        if ($S_{lp} \neq FALSE$) then
          return $S_{lp}$
        end if
      end if
    end if
    pop_lp
  end if
  pop_cp
  return $S_{lp}$
```

constraints of the RUP constitute a totally unimodular set. For a proof see [SRW97].

The Resource Utilization Problem perturbation function δ is defined to be an absolute change over the temporal variables. If x is a temporal variable then the absolute change in x is $d_x = |x - c|$, where c represents the value of x in previous solution. Since d_x, which is a non–linear function, is introduced in the objective function the following constraints are added:

$$d_x \geq x - c \tag{4.1}$$

$$d_x \geq c - x \tag{4.2}$$

Even if (4.2) violates the sufficient conditions for total unimodularity this characteristics is preserved on the incremental addition of new variable d_x and the constraints (4.1) and (4.2), which is proved in [SW00].

The experimental evaluation of the unimodular probing algorithm was done on a set of commercial aircraft utilization problems. The performance of the algorithm was compared with specialized constraint programming algorithm (a detailed description of the algorithm may be found in [Pot97], a modified version of the algorithm is given in Section 4.1) and mixed integer programming (MIP) search. In comparison with pure constraint programming algorithm unimodular probing is capable to find an optimal solution for many problems where the pure constraint algorithm fails.

In comparison with MIP search unimodular probing achieves the substantial reductions in the number of search nodes. Another advantage of the unimodular probing algorithm over MIP techniques lies in the search ordering heuristics. MIP sees all violations in terms of integrity of problem variables that results in facing large amount of violations which are vary similar in their nature. Unimodular probing faces instead much fewer number of violations, but they seem much more heterogenous in their character. Moreover, the non–discrete values returned by the LP–solver for the relaxed MIP problem have limited importance in a discrete context, which results in quite poor information about search choices.

In contrast, unimodular probing gives a relatively high level view of the sources of violation, which enables the use of meaningful heuristics for repair selection and bottleneck prioritization.

4.4 Local probing

Local search is a search method based on the idea that an exhaustive exploration of the search space of a problem is in most cases not necessary in order to find a feasible or optimal solution. A solution can be found by exploring assignments which lie in the neighborhood of current assignments.

Definition 4.2 *A neighborhood* $\mathbf{N}(\mathbf{x}, \sigma)$ *of a solution* $\mathbf{x}$ *is a set of solutions that can be reached from* $\mathbf{x}$ *by a simple operation* σ*.*

Examples of such operations are adding or removing an object from the solution.

If a solution $\mathbf{y}$ is better than any other solution in its neighborhood $\mathbf{N}(\mathbf{y}, \sigma)$ then $\mathbf{y}$ is a local optimum with respect to this neighborhood [Ree95].

Preliminary research on local search as a probing method was presented in [KSL01]. The authors evaluate hill–climbing search combined with several heuristics: limited variable search and limited shift search, both based on minimal discrepancy search [HG95], and minimal perturbation search [SW00], where the neighborhood operator correspond to the perturbation function δ in MPP, see Definition 3.2 in Section 3.2.

In the limited variable search the search decisions are variable assignments. The neighborhood solution is searched by letting one variable change. If a feasible solution is not found after exploring all possibilities the discrepancy limit is increased by one. This is continued until a feasible solution is found or the whole search is explored.

Limited shift search changes the number of variable re–assignments. The measure of discrepancy is the absolute distance of the new assignment from the basic solution. The variables for re–assignment are chosen in random order. The new value which a chosen variable takes is the one which gives a biggest improvement.

The minimum perturbation search heuristic assigns a value to one variable and uses linear programming to assign the remaining variables in a way which minimizes changes to previous assignment. The objective function and the minimal perturbation constraints are modeled as follows:

$$min \sum_{i=1}^{N} d_{x_i} \tag{4.3}$$

s.t.

$$d_{x_i} \geq x_i - c_i \qquad i = 1, 2, \ldots, N \tag{4.4}$$
$$d_{x_i} \geq c_i - x_i \qquad i = 1, 2, \ldots, N \tag{4.5}$$

where $d_x = |x - c|$ represents the absolute change between variable x and its initial value c and N is the total number of noninstantiated variables. The objective function in (4.3). is linear and can be solved vary quickly.

A performance evaluation for a local prober was conducted on the set of benchmarks for resource feasibility problems [ICP00](see Section 3.2). Since heuristic value selection in the Minimal Perturbation Search tends to lead hill–climbing to local minima, a *semi–random* value selection was added to the neighborhood operator. This heuristics divides the variable domain into the set of values which potentially can improve and those which can deteriorate a solution. Values which improve the solution are explored first.

In addition tests with neighborhood operators using depth–first search were conducted for comparison. These tests show that minimum perturbation search is the best choice for quickly finding a feasible solution to the problem and outperforms both minimum discrepancy heuristics. Depth first search performs almost as well as minimum perturbation search.

As mentioned before the investigations presented in the paper are preliminary, there is for example no comparison between using a local prober and those presented in [SW00] or [SRW97].

4.5 Hybridization based on Benders decomposition

The idea of decomposing an optimization problem into two subproblems which can be solved by cooperating solvers, presented in Section 4.3, is based on the fact that the cost function involved in the optimization process is linear or can be approximated by a linear or piecewise linear functions. Since linear programming offers efficient constraint solvers, which can quickly return optimal solutions to the problems, it seems advantageous to combine constraint and linear programming to solve complex optimization problems.

The form of hybridization referred in this section is based on the concept of a "master" problem, for which the optimal solution is found, and another subproblems which interact with the master problem. In the simplest case the subproblem examines the latest optimal solution of the master problem and determines whether this optimal solution violates any constraint of the subproblem. In case of such a violation the subproblem returns one or more alternative solutions to the master problem, to reduce the probability that such a violation occurs again. One of these constraints is added to the master problem and a new optimal solution is found. To prove global optimality each of the returned alternatives is added to the master problem on different branches of the search tree.

Using operation research language this would correspond to *"row generation"* in contrast to *column generation.* The unimodular probing method described in Section 4.3 is such a "row generation" method.

Another form of hybridization based master/slave problem concept is *Benders decomposition.* The idea of Benders decomposition was first presented in [Ben62] and generalized in [Geo72]. Integration of Benders decomposition and constraint programming was described in [EW01] and [Tho01].

The classical Benders decomposition is presented below.

Consider the linear program **P** given by;

$$\begin{aligned}
\mathbf{P}:\ & min\ \mathbf{f^T x} + \sum_{i=1}^{I} \mathbf{c_i^T y_i} & & \quad (4.6)\\
subject\ to\ & \mathbf{G_i x} + \mathbf{A_i y_i} \geq \mathbf{b_i} & \forall i & \\
& \mathbf{x} \in \mathcal{D_X} & & \\
& \mathbf{y_i} \geq 0 & \forall i &
\end{aligned}$$

Fixing **x** to some value $\mathbf{x^k}$ results in a linear program in $\mathbf{y_i}$ which can have a special structure or can be easy to solve. This program may be partitioned as follows:

$$\begin{aligned}
\mathbf{P}:\ & \min_{\mathbf{x}\in\mathcal{D_X}} \left\{ \mathbf{f^T x} + \sum_{i=1}^{I} (min\ \{\mathbf{c_i^T y_i} : \mathbf{A_i y_i} \geq \mathbf{b_i} - \mathbf{G_i x}, \mathbf{y_i} \geq \mathbf{0}\}) \right\} \\
& = \min_{\mathbf{x}\in\mathcal{D_X}} \left\{ \mathbf{f_T x} + \sum_{i=1}^{I} (max\{\mathbf{u_i}(\mathbf{b_i} - \mathbf{G_i x}) : \mathbf{u_i A_i} \leq \mathbf{c_i}, \mathbf{u_i} \geq \mathbf{0}\})) \right\}
\end{aligned} \quad (4.7)$$

where the inner optimization has been dualized. If the $U_i = \{\mathbf{u_i} : \mathbf{u_i Ai} \leq \mathbf{c_i}, \mathbf{u_i} \geq \mathbf{0}\}$ is non–empty for each i there exists an optimal solution to each inner optimization or it is unbounded along extreme rays. By letting $u_i^1, \ldots, u_i^{t_i}$ and $d_i^1, \ldots, d_i^{s_i}$ be the extreme points and directions of U_i equation (4.7) can be rewritten as a mixed integer *Master Problem* **MP**:

$$\begin{aligned}
\mathbf{MP}:\ & min\ z = \mathbf{f^T x} + \sum_{i=1}^{I} \beta_i \\
& s.t.\ \beta_i \geq \mathbf{u_i^k}(\mathbf{b_i} - \mathbf{G_i x}) \quad \forall i\ \ \forall k \\
& \mathbf{0} \geq \mathbf{d_i^l}(\mathbf{b_i} - \mathbf{G_i x}) \quad \forall i\ \ \forall l \\
& \mathbf{x} \in \mathcal{D_X}
\end{aligned} \quad (4.8)$$

where β_i is a bound on the optimal value that depends on x called *Benders cut.*

Since there may exist many extreme points in the direction of each U_i which has its impact on number of constraints in 4.8 the master problem must be relaxed. If for some relaxed problem $\mathbf{RMP^k}$ the optimal relaxed solution $(z^k, \mathbf{x^k})$ satisfies all the constraints of (4.8) then $(z^k, \mathbf{x^k}, \mathbf{y_1^k}, \ldots, \mathbf{y_I^k})$ is an optimal solution of (4.6). Otherwise there exists some constraint or *Benders cut* in (4.8) which is violated for $\mathbf{x} = \mathbf{x^k}$. The violated element is then added to $\mathbf{RMP^k}$ creating $\mathbf{RMP^{k+1}}$, the ground for the next iteration.

The Benders cut is determined by fixing $\mathbf{x} = \mathbf{x^k}$ in (4.7):

$$\begin{aligned} \mathbf{SP_i^k} : \ & max\ \beta_i^k = \mathbf{u_i}(\mathbf{b_i} - \mathbf{G_i x^k}) \\ subject\ to\ \ & \mathbf{u_i A_i} \le \mathbf{c_i} \\ & \mathbf{u_i} \ge \mathbf{0} \end{aligned} \tag{4.9}$$

If any subproblem $\mathbf{SP_i^k}$ has an unbounded optimal solution for $\mathbf{x^k}$ then the primal of subproblem is infeasible for $\mathbf{x^k}$. If any subproblem $\mathbf{SP_i^k}$ is infeasible for $\mathbf{x^k}$ then its infeasible for any $\mathbf{x}$. In these cases the *Homogenous Dual* to the primal of the subproblem is considered:

$$\begin{aligned} & max\ \mathbf{u_i}(\mathbf{b_i} - \mathbf{G_i x^k}) \\ & subject\ to\ \ \mathbf{u_i A_i} \ge \mathbf{0} \\ & \mathbf{u_i} \ge \mathbf{0} \end{aligned} \tag{4.10}$$

This problem is always feasible since $\mathbf{u_i} = \mathbf{0}$ is a solution and can have an unbounded optimum when the primal is infeasible and finite optimal solution when the primal is feasible. In case of an unbounded solution a cut $\mathbf{u_i^k}(\mathbf{b_i} - \mathbf{G_i x}) \le 0$ which corresponds to an extreme direction of $U_i = \{\mathbf{u_i} : \mathbf{u_i A_i} \le \mathbf{0}, \mathbf{u_i} \ge \mathbf{0}\}$ is obtained.

The complete Benders decomposition is presented as Algorithm 4.4.

Algorithm 4.4 Benders Decomposition Algorithm

```
1: Initialization: From the original linear program $\mathbf{P}$ (4.6) construct the *relaxed master problem* $\mathbf{RMP^0}$ (4.8) with the initial constraint set $\mathbf{x} \in \mathcal{D_X}$ and set $k = 0$.
2: Iterative step: From the current relaxed problem $\mathbf{RMP^k}$ with the optimal solution $(z^k, \mathbf{x^k})$ construct $\mathbf{RMP^{k+1}}$ with optimal solution $(z^{k+1}, \mathbf{x^{k+1}})$: fix $\mathbf{x} = \mathbf{x^k}$ in $\mathbf{P}$ and solve the resulting subproblems $\mathbf{SP_i^k}$ (4.9) according to following cases:
     • $\mathbf{SP_i^k}$ is primal unbounded for some $i$ – stop with the original problem having unbounded solution.
     • $\mathbf{y_i^k}, \mathbf{u_i^k}$ are, respectively, primal and dual solutions of subproblem $\mathbf{SP_i^k}$ with objective values $\beta_i^k$ for each $i$.
          – if $\sum_{i=1}^{I} \beta_i^k = z^k$ then stop with $(z^k, \mathbf{x^k}, \mathbf{y_1^k}, \dots, \mathbf{y_I^k})$ as the original solution to the original problem.
          – if $\sum_{i=1}^{I} \beta_i^k > z^k$ then add the *Benders cuts* $\beta_i \ge \mathbf{u_i^k}(\mathbf{b_i} - \mathbf{G_i x})$ to $\mathbf{RMP^k}$ to form the new relaxed master problem $\mathbf{RMP^{k+1}}$ set $k = k + 1$ and return to step 2.
     • $\mathbf{SP_i^k}$ is dual unbounded or both primal and dual unfeasible for some $i$ – find an extreme direction $\mathbf{d_i^k}$ of the homogeneous dual leading to unboundness; add the cut $\mathbf{d_i^k}(\mathbf{b_i} - \mathbf{G_i x}) < 0$ to $\mathbf{RPM^k}$ to form a new relaxed master problem $\mathbf{RMP^{k+1}}$, set $k = k + 1$ and return to step 2.
```

The classical Benders decomposition can be generalized for the problems with nonlinear constraints and nonlinear objective function. In most general form the original problem:

$$\begin{aligned}\mathbf{P}:\ & \mathit{min} f(f_1(\mathbf{x},\mathbf{y_1}),\ldots,f_I(\mathbf{x},\mathbf{y_I}))\\ \mathit{subject\ to}\ & g_i(\mathbf{x},\mathbf{y_i}) \geq \mathbf{b_i} \quad \forall\, i\\ & \mathbf{x} \in \mathcal{D_X}\\ & \mathbf{y_i} \in \mathcal{D_Y} \qquad\quad \forall\, i\end{aligned} \tag{4.11}$$

is decomposed into a master problem:

$$\begin{aligned}\mathbf{MP}:\quad \mathit{min}\quad & z = f(\mathbf{x},\beta_1,\ldots,\beta_I)\\ \mathit{subject\ to}\quad & \beta_i \geq \beta_i^k(\mathbf{x}) \qquad \forall i\ \forall k\\ & \mathbf{0} \geq \beta_i^l(\mathbf{x}) \qquad \forall i\ \forall l\\ & \mathbf{x} \in \mathcal{D_X}\end{aligned} \tag{4.12}$$

and subproblems:

$$\begin{aligned}\mathbf{SP_i^k}\ :\ \mathit{min}\quad & f_i(\mathbf{x^k},\mathbf{y_i})\\ \mathit{subject\ to}\quad & g_i(\mathbf{x^k},\mathbf{y_i}) \geq \mathbf{b_i}\\ & \mathbf{y_i} \in \mathcal{D_Y}\end{aligned} \tag{4.13}$$

An evaluation of the algorithm (see [EW01]) was performed on instances of the minimal perturbation problem from [ICP00]. Correct and optimal solutions were returned but the performance was of one magnitude slower than for the unimodular probing described in [SRW97]. The reason for such a behavior may be the composition of benchmarks. Benders Decomposition proves to be very efficient in case the problem breaks down into a master problem and multiple subproblems. Since the benchmarks investigated in the paper involve a single kind of resource they do not have an apparent decomposition into multiple subproblems, which may be one of the reasons why this algorithm performs slower then unimodular probing.

Chapter 5

Other approaches

Apart from the techniques described in previous chapters there exist several other approaches to dynamic scheduling which have evolved in the field of artificial intelligence. In this chapter we will describe some of the most interesting of these approaches.

5.1 Reactive Scheduling in MicroBoss

In MicroBoss [SOS93, Sad91] there exist two levels of control used to handle disruptions to the schedule:

1. **Control level:** small disruptions which require fast responses are handled by simple control rules like "process the operation with earliest scheduled start time first" or "in case of machine breakdown, reroute critical jobs to any available equivalent machine".
2. **Scheduling level:** in a case of more severe disruption the scheduling algorithms inside MicroBoss scheduler are used to repair or re-optimize the schedule from the global perspective while attending more immediate actions.

One of the most important decision which must be taken by scheduler is which disruption is to be handled by which control level. There exists a significant tradeoff between slower but more optimal and immediate but local decisions.

When repairing a schedule at scheduling level, MicroBoss first selects a number activities to unschedule and then reschedule them using a micro–opportunistic search procedure. The rescheduling occurs in the context of the remaining current schedule which is not validated by the schedule breakage.

There is a number of propagation procedures proposed in [SOS93], that can be used on the control level to reschedule activities or at the scheduling level to identify activities to be rescheduled. Nevertheless, there exists a possibility that all them will fail to generate a new solution. In this case, the activities which could not be rescheduled are expanded incrementally until some solution is found.

As an example of a scheduling MicroBoss rule the `Right Shift Rule` is presented in Figure 5.1. In this example there are three resources R_1, R_2, R_3 and

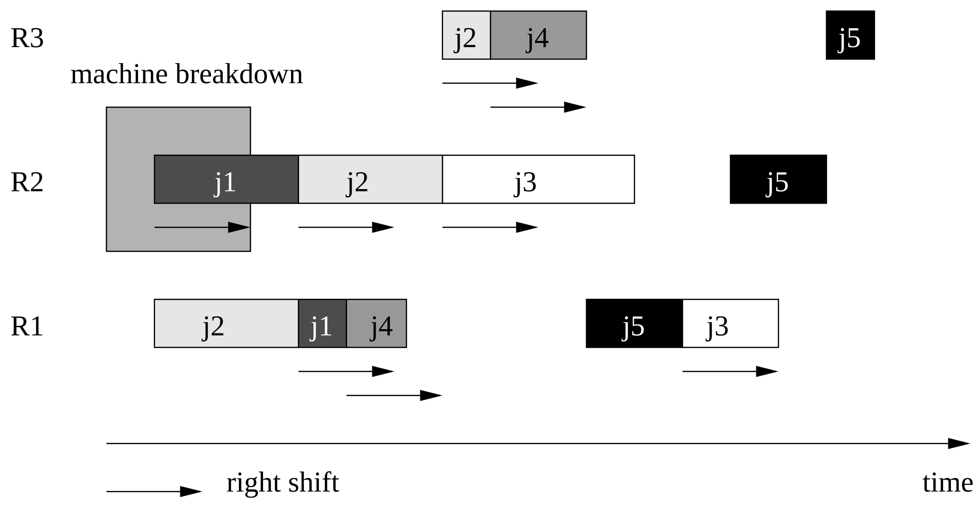


Figure 5.1: The Right Shift Rule

five jobs j_1 to j_5. In the example the resource breaks down near the beginning of the schedule. The Right Shift Rule is applied here by moving forward in time all the activities which are affected by the breakdown, either because they were executing on the resource in the time of breakdown or because of precedence rule. Since this rule is very simple and straightforward it can produce very poor solutions. Therefore better solutions can be achieved by using the rule to identify the set of activities to be rescheduled at scheduling level, where resequencing can occur.

A number of such procedures are evaluated on a set of randomly generated problems with a single simulated machine breakdown [SOS93]. The evaluation shows that total rescheduling of remaining activities produced the best quality solutions, however, the resulting schedule differs significantly from the original and took the longest time to compute.

5.2 Artificial Immune Systems

Preliminary work on using artificial immune system model for dynamic scheduling has been done by Hart, Ross et. al. and was presented in series of publications including [RHC97, HNR98, HR99a, HR99b]. The basic approach of the authors is that most of the unexpected (unscheduled) events causing breakage of the currently executed schedule are to some degree predictable. For example there can exist a pattern for work loads in the specific time periods or there can exist patterns in customer orders etc. Those patterns may have a set of corresponding actions which can be taken to during rescheduling to minimize their impact on the executed schedule.

The authors propose to evolve a set of such actions by creating a set of partial schedules based on some historic data, which can be used as building blocks to respond to unexpected events. Those partial schedules are evolved for every used resource off–line using genetic algorithms. They are represented by a sequence of activities and some “wildcard” activities which can be replaced during execution.

The fitness criterion for evolving partial schedules is based on matching the created fragment against historical schedules. The fragment is evaluated as more fitted if it contains a sub–sequence which can be recognized in any historical schedule.

The on–line phase of scheduling is then to use evolved fragments to build a new schedule as a response to unexpected events. The wildcard matches any activity fitting to the schedule, which gives a possibility to create more complex non–contiguous schedules. Since this model is based on the assumption that any unexpected event during has been encountered in the past, a combination of the evolved sequences is likely to encode a good schedule.

The experiments on using artificial immune systems in scheduling have so far been focused on the creation of the schedule fragments and on the techniques of matching them against historical data. They show that, depending on parameter settings during the evolution, it is possible to create partial schedules which can be successfully matched against fragments of existing schedules.

Since this technique is quite new it is impossible to compare it in its current state with other techniques. The significant step from matching valid schedules to incorporating schedule fragments into the currently executed schedule, as a reaction to standard types of unexpected events, is still to be taken.

Chapter 6

Conclusions

This report describes the state of the art in the field of dynamic scheduling. The Constraint Satisfaction framework for dynamic scheduling was presented and techniques which can be used to solve those problems were described. Also an overview of cooperating solvers and so called hybrid methods for complex dynamic sheduling problems was given.

Several future research directions in the area have been discussed. Of these, the idea of cooperative solvers which combines methods known from Operation Research and Constraint Programming seems most promising. We believe that cooperating solvers can be used to solve constraint optimization problems in general. The current investigation of hybridization methods does not take advantages of the dynamic nature of the problem. Any changes in scheduling environment creates a new Constraint Satisfaction Problem which is solved from scratch. Propagating changes in configuration of the problem in context of cooperative solvers should be investigated further.

As it was shown, many of the problems which include temporary changes, e.g., Minimal Perturbation Problem, are usually modelled as Dynamic CSPs. The possibility to model those problems using the Recurrent DCSP framework should be investigated.

Moreover, further research on the methods for propagating changes for Dynamic Constraint Satisfaction Problems should be conducted.

On the other hand there is no work, that we are aware of, on modeling dynamic scheduling problems using the framework of the global constraints. This approach should be further investigated.

Finally, methods which gather the information about the nature of the changes to the scheduling problem and use the gathered information to guide the search for solution to the new problem should be further investigated.

Bibliography

[AB92] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. In *Actes des Journees Francophones de Programmation end Logique, Lille*, 1992.

[BB93] H. Beringer and B. De Backer. Satisfiability of boolean formulas over linear constraints. In *IJCAI*, pages 296–304, 1993.

[BE94] N. Beldiceanu and E.Contejean. Introducing global constraints in chip. *Methematical Computation Modelling*, 20(12):97–123, 1994.

[Bel00] N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In *Principles and Practice of Constraint Programming*, pages 52–66, 2000.

[Ben62] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

[Bes92] C. Bessiere. Arc-consistency for non-binary dynamic CSPs. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 23–27, Vienna, Austria, August 1992. John Wiley & Sons.

[BFR95] C. Bessière, E. C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In *Proceedings of IJCAI'95, Montréal, Canada*, pages 592–598, 1995.

[BvB98] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *AAAI/IAAI*, pages 310–318, 1998.

[CF90] W. Y. Chiang and M. S. Fox. Protection against uncertainty in a deterministic schedule. In *Proceedings 4th International Conference on Expert Systems and the Leading Edge in Production and Operations Management*, pages 184–197, Hilton Head Island, 1990.

[Cha87] P. Chatalic. Incremental techniques and prolog. Technical Report TR-LP-23, European Computer-Industry Research Centre, June 1987.

[Cha89] P. Chatalic. IMPRO: an environment for incremental execution in prolog. Technical Report TR-LP-42, European Computer-Industry Research Centre, May 1989.

[CP89] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.

[CP94] J. Carlier and E. Pinson. Adjustments of heads and tails for the job-shop scheduling problem. *European Journal of Operational Research*, 78:146–161, 1994.

[DB00] A.J. Davenport and J.Ch. Beck. Managing uncertainty in scheduling: a survey. Preprint, 2000.

[DBS94] M. Drummond, J. Bresina, and K. Swanson. Just-in-case scheduling. In *Proc. of the Twelth National Conference on Artificial Intelligence (AAAI-94)*, pages 1098–1104, Seattle, WA, 1994. AAAI Press.

[DD88] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI-88*, pages 37–42, 1988.

[dK89] J. de Kleer. A comparison of ATMS and CSP techniques. In N. S. Sridharan, editor, *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 290–296, Detroit, MI, USA, August 1989. Morgan Kaufmann.

[Doy81] J. Doyle. A truth maintenance system. In Bonnie Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 496–516. Morgan Kaufmann, Los Altos, California, 1981.

[EKR96] A. El-Kholy and B. Richards. Temporal and resource reasoning in planning: the parcplan approach. In *Proc. ECAI-96*, 1996.

[EW01] A. Eremin and M. Wallace. Hybrid benders decomposition algorithms in constraint logic programming. In T. Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, Lecture Notes in Computer Science, pages 1–15. Springer, 2001.

[Gao95] H. Gao. Building robust schedules using temporal protection(an empirical study of constraint based scheduling under machine failure uncertainty). Master's thesis, Department of Industrial Engineering, University of Toronto, 1995.

[Geo72] A.M. Geoffrion. Generalized benders decomposition. *Journal of Optimization theory and Applications*, 4(10):237–260, 1972.

[Gho96] S. Ghosh. *Guaranteeing Fault-Tolerance through Scheduling in Real-Time Systems.* PhD thesis, University of Pittsburgh, 1996.

[Gin93] M.L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, August 1993. (electronic journal).

[GMM95] S. Ghosh, R. Melhem, and D. Mosse. Enhancing real-time schedules to tolerate transient faults. In *IEEE Real-Time Systems Symposium*, pages 120–129, 1995.

[HG95] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *Proc. of the Fourteen International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 607–615. Morgan Kaufman, 1995.

[Hil94] D. W. Hildum. Flexibility in a knowledge-based system for solving dynamic resource-constrained scheduling problems. Technical Report UM-CS-1994-077, Department of Computer Science,University of Massachusetts, Amherst,MA, 1994.

[HNR98] E. Hart, J. Nelson, and P. Ross. Solving a real-world problem using an evolving heuristically driven schedule builder. *Evolutionary Computation*, 6(1):61–80, 1998.

[HP91] P. Van Hentenryck and T. Le Provost. Incremental search in constraint logic programming. *New Generation Computing*, 9:257–275, 1991.

[HR99a] E. Hart and P. Ross. The evolution and analysis of portential antibody library ror job-shop scheduling. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 185–202. McGraw-Hill, 1999.

[HR99b] E. Hart and P. Ross. An immune system approach to scheduling in changing environments. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M., and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1559–1566, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[ICP00] Introduction to resource utilization benchmarks. Available at www.icparc.ic.ac.uk/˜ hhe/RFPBenchmarks/new_benchmark_intro.html, 2000.

[KCSÅ01] P. Kreuger, M. Carlsson, T. Sjöland, and E. Åström. Sequence dependent task extensions for trip scheduling. Technical report, Swedish Institute of Computer Science, 2001.

[KSL01] O. Kamarainen, H. El Sakkout, and J. Lever. Local probing for resource constrained scheduling. In *International Conferences Constraint Programming & Logic Programming. Workshop Proceedings. CoSolv'01: Cooperative Solvers in Constraint Programming.*, pages 73–86, 2001.

[Lho93] O. Lhomme. Consistency techniques for numeric CSPs. In Ruzena Bajcsy, editor, *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 232–238, Chambery, France, 1993. Morgan Kaufmann.

[LP86] L. Lovasz and M.D. Plummer. Matching theory. *North-Holland mathematic studies*, 121, 1986.

[LW94] V. Leon and S. Wu. Storer: Robustness measures and robust scheduling for job shops, 1994.

[Mac77] A.K. Mackworth. Consistency in network of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[McA90] D. McAllester. Truth maintenance. In R. Smith and T. Mitchell, editors, *Proceedings of the Eighth National Conference on Artificial Intelligence*, volume 2, pages 1109–1116. AAAI Press, 1990.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[MJPL92] S. Minton, M. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[Mon74a] U. Montanari. Networks of constraints fundamental properties and applications to picture processing. *Information Sciences*, 66(7):95–132, 1974.

[Mon74b] U. Montanari. Optimization methods in image processing. In *Proceedings IFIP*, pages 727–732, North-Holland, 1974.

[MS89] M. Maher and P. Stuckey. Expanding query power in contraint logic programming languages. In *Proceedings of the North American Conference on Logic Programming (NACLP–89), Cleveland, Ohio, U.S.A*, pages 20–36, 1989.

[NW88] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization.* Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Ltd., 1988.

[OTF86] M. Ohki, A. Takeuchi, and K. Furukawa. A framework for interactive problem-solving based in interactive query revision. In E. Wada, editor, *Proc. 5th Conf. on Logic Programming'86*, Springer LNCS 264, pages 137–146, 1986.

[Pot97] D. Pothos. A constraint-based approach to the bristish airways schedule re-timing problem. Technical Report 97/04-01, IC-Parc, Imperial College, 1997.

[Ree95] C.R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* McGraw-Hill Book Company, 1995.

[Rég94] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[RHC97] P. Ross, E. Hart, and D. Corne. Some observations about GA-based exam timetabling. In *PATAT*, pages 115–129, 1997.

[RWH98] R. Rodosek, M. Wallace, and M. Hajian. A new approach to integrating mixed integer programming with constraint logic programming. *Annals of Operations Research*, 1998.

[Sad91] N. Sadeh. *Look-ahead Techniques for Micro-opportunistic Job Shop Scheduling.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

[Sga97] J. Sgall. On-line scheduling - a survey. In A. Fiat and G. Woeginger, editors, *On-Line Algorithms.* Springer-Verlag, Berlin, 1997. Available from citeseer.nj.nec.com/sgall97line.html.

[SOS93] N. Sadeh, S. Otsuka, and R. Schnelbach. Predictive and reactive scheduling with the micro-boss production scheduling and control system. In *Proc. IJCAI-93 Workshop on Knowledge–based Production Planning, Scheduling, & Control*, Chambery, France, Aug 1993.

[SRW97] H. El Sakkout, T. Richards, and M. Wallace. Unimodular probing for minimal perturbance in dynamic resource feasibility problems. In *Proc. of the CP97 workshop on Dynamic Constraint Satisfaction*, 1997.

[SRW98] H. El Sakkout, T. Richards, and M. Wallace. Minimal perturbation in dynamic scheduling. In Henri Prade, editor, *ECAI98. 13th Europen Conference on Artificial Intelligence*. John Wiley & Sons, Ltd., 1998.

[SW99] K. Stergiou and T. Walsh. Encodings of non-binary constraint satisfaction problems. In *Proceedings of AAAI/IAAI 1999*, pages 163–168, Orlando, Florida, USA, July 1999. AAAI Press / The MIT Press.

[SW00] H. El Sakkout and M. Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints. An International Journal. Special Issue on Industrial Constraint-Directed Scheduling*, 5(4):359–388, October 2000.

[Tho01] E. Thorsteinsson. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. In T. Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, Lecture Notes in Computer Science, pages 16–30. Springer, 2001.

[Tsa93] Tsang, E. *Foundation of Constraint Satisfaction*. Academic Press, 1993.

[vEOT86] M. van Emden, M. Ohki, and A. Takeuchi. Spreadsheets with incremental queries as a user interface for logic programming. *New Generation Computing*, 4:287–304, 1986.

[VS94a] G. Verfaillie and T. Schiex. Dynamic backtracking for dynamic constraint satisfaction problems. In *Proceedings of the ECAI'94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications, Amsterdam, The Netherlands*, pages 1–8, 1994.

[VS94b] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the 12th National Conference on Artificial Intelligence. Volume 1*, pages 307–312, Menlo Park, CA, USA, July 31–August 4 1994. AAAI Press.

[Wal75] D.L. Waltz. Understanding line drawing of scenes with shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[Wal98] J.R. Wallace. Analysis of heuristic methods for partial constraint satisfaction problems. In M. Maher and J.-F. Puget, editors, *Principle and Practice of Constraint Programming - CP'98*, Lecture Notes on Computer Science No. 1520, pages 482–496. Springer, 1998.

[WF98] R.J. Wallace and E.C. Freuder. Stable solutions for dynamic constraint satisfaction problems. In *Principles and Practice of Constraint Programming*, pages 447–461, 1998.

[WF99] R.J. Wallace and E.C. Freuder. Representing and coping with recurrent change in dynamic constraint satisfaction problems. In *CP'99 Workshop on Modelling and Solving Soft Constraints*, 1999.