The Knowledge Engineering Review, Vol. 00:0, 1–24. © 2004, Cambridge University Press
DOI: 10.1017/S000000000000000   Printed in the United Kingdom

New Trends in Constraint Satisfaction, Planning, and Scheduling: A Survey

Roman Barták[1], Miguel A. Salido[2], Francesca Rossi[3]

1*Charles University in Prague,*
[2] *Universidad Politécnica de Valencia,*
[3] *University of Padova*
E-mail: bartak@ktiml.mff.cuni.cz, msalido@dsic.upv.es, frossi@math.unipd.it

Abstract

During recent years the development of new techniques for constraint satisfaction, planning, and scheduling has received increased attention, and substantial effort has been invested in trying to exploit such techniques to find solutions to real life problems. In this paper, we present a survey on constraint satisfaction, planning and scheduling from the Artificial Intelligence point of view. In particular, we present the main definitions and techniques, and we discuss possible ways of integrating such techniques. We also analyze the role of constraint satisfaction in planning and scheduling, and we hint at some open research issues related to planning, scheduling and constraint satisfaction.

1 Introduction

Planning and scheduling techniques have recently seen important advances thanks to the application of constraint satisfaction models and tools (Nareyek *et al.*, 2005). Most real-world problems can be cast as highly coupled planning and scheduling problems, where resources must be allocated so as to optimize overall performance objectives. Therefore, solving these problems requires an adequate mixture of planning, scheduling and resource allocation to competing goal activities over time in the presence of complex state-dependent constraints. Solutions to these problems must integrate resource allocation and plan synthesis capabilities, which can be efficiently managed by using constraint techniques. The aim of this survey is to give a general overview of three different but interrelated areas: constraint satisfaction, planning, and scheduling.

Constraint satisfaction is a technology for solving combinatorial optimization problems. It is based on idea of modeling a real-life problem as a set of decision variables, each having a set of possible values, and a set of constraints restricting the allowed combinations of values to variables. The task is to instantiate the variables with the values while satisfying all the constraints. Most constraint solvers are based on a careful combination of search and inference (also called constraint propagation). In this survey, we introduce the fundamental notions of constraint programming and present some of the solving techniques.

Planning and scheduling problems are combinatorial optimization problems, hence constraint satisfaction seems to be an appropriate technology for solving these problems and for facilitating a cross-fertilization among the two areas. Planning deals with finding a sequence of actions transferring some initial state of the world into a desired state while scheduling focuses on an optimal allocation of actions to time and resources. In this survey, we will describe some pure planning and scheduling techniques and we will also show how to model these problems as constraint satisfaction problems.

The paper is organized as follows. In the next section we introduce classical CSPs and the main notions needed to present several solution methods. We then present some introductory examples to understand the importance of a good modeling phase. Then we present some constraint propagation techniques, that transform the initially given CSP into an equivalent, but smaller problem. Such techniques are notoriously

very useful to prune the search space. We then present three families of search techniques: Look-Back, Look-Ahead, and local search. Finally, we discuss some variable and value ordering heuristics, that can improve the efficiency of the previous techniques.

Section 3 is focused on planning and scheduling from the Artificial Intelligence perspective. We first present the classical planning problem and the simplest planning algorithms used in most successful planners. Then, we present a brief history of scheduling and the basic scheduling terminology. Due to the variety of scheduling problems, we present some examples to show the basic principles.

Section 4 is centered on the role constraint satisfaction plays in planning and scheduling problems. Many real planning and scheduling problems can be modeled as a constraint satisfaction problem. In this section we present some examples of constraint models for finding plans. and some examples and techniques of using constraint satisfaction for solving scheduling problems.

In section 5 we presents some open research issues which involve some of the topics of this survey. One of the most important open issues is the effective integration on planning and scheduling techniques, which is currently being studied by many researchers. Other interesting open issues are the relationship between knowledge/software engineering and planning and scheduling, temporal reasoning, preferences, and distributed search.

2 Constraint Satisfaction Problems

2.1 Introduction

Constraint programming (CP) (Apt, 2003; Dechter, 2003; Marriott & Stuckey, 1998; Tsang, 1993; Rossi *et al.*, 2006) is a powerful paradigm for solving combinatorial problems. CP was born as a multi-disciplinary research area that embeds techniques and notions coming from many other areas, among which are artificial intelligence, computer science, databases, programming languages, and operations research. Constraint programming is currently applied with success to many domains, such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics.

Constraints are just relations, and a *constraint satisfaction problem* (CSP) states which relations should hold among the given decision variables. For example, in scheduling activities in a company, the decision variables might be the starting times and the durations of the activities and the resources needed to perform them, and the constraints might be on the availability of the resources and on their use by a limited number of activities at a time. Another example is configuration, where constraints are used to model compatibility requirements among components or user's requirements. For example, if we were to configure a laptop, some video boards may be incompatible with certain monitors. Also, the user may pose constraints on the weight and/or the screen size.

Constraint solvers take a real-world problem like this, represented in terms of decision variables and constraints, and attempt to find an assignment to all the variables that satisfies the constraints. Constraint solvers search the solution space either systematically, as with *backtracking* or *branch and bound* algorithms, or use forms of local search which may be incomplete. Systematic methods often interleave search and inference, where inference consists of propagating the information contained in one constraint to the neighboring constraints. Such inference (usually called *constraint propagation*) is very useful, since it usually requires a small time to be computed and it may reduce the parts of the search space that need to be visited. Constraints that are often used in real-life problems, called *global constraints*, come with their own ad-hoc efficient propagation mechanisms, that make them especially efficient to use.

Rather than trying to satisfy a set of constraints, we may want to *optimize* them. This means that there is an objective function that measures the quality of each solution, and the aim is to find a solution with optimal quality, where the quality of a solution can be expressed in terms of preferences. For such problems, techniques such as branch and bound are usually used to find an optimal solution.

2.2 Definitions and Formulations

The concept of a constraint satisfaction problem is fundamental in constraint programming. Hence, we need to give a formal definition of it. We first introduce the concepts of a variable domain and constraint,

and then we define the concept of a constraint satisfaction problem and related concepts. Then, we present different formulations for the same problem to show that a CSP can be modeled in many different manners. One of the most important tasks for solving a real life problem is to represent the problem as a CSP, that is, in terms of variables, domains and constraints.

2.2.1 Definitions

In this section, we give the formal definition for the CSP, and introduce certain characteristics to be referred to when discussing solution methods.

Variable Domain. The *domain* of a variable is a set of all considered values that can be assigned to the variable. Usually, we assume finite discrete domains.

Instantiation. An *instantiation* is a set of pairs (x_i, a_i) such that

- x_i is a variable
- a_i is a value from the domain of x_i
- each variable appears at most once in the instantiation.

We say that the instantiation is *complete* for a set of variables X if each variable from X appears in the instantiation and no other variable appears there. If the instantiation does not contain all variables from X, then the instantiation is incomplete (partial) for X. We say that instantiation I extends instantiation J if $I \supseteq J$.

Constraint. A *constraint* is a pair (t, R), where t is a set of variables (called *scope*; the size of scope is called *arity*) and R is a set of complete instantiations for t (sometimes called a domain of the constraint). We can also see the constraint as a subset of the Cartesian product of domains of variables in t - the constraint restricts the values that the variables can simultaneously take.

Instantiation I *satisfies the constraint* (t, R) if there exists $J \in R$ such that I extends J. Constraint can be specified extensionally as a set of tuples (satisfying instantiations) or intentionally as a formula that defines the satisfying instantiations.

CSP. A *constraint satisfaction problem* consists of:

- a set of variables $X = \{x_1, x_2, ..., x_n\}$
- a set of domains $D = \{D_1, D_2, ..., D_n\}$ such that for each variable $x_i \in X$ there is a domain D_i;
- a set of constraints $C = \{c_1, c_2, ..., c_k\}$ such that the scope of each constraint is a subset of X.

Solution. A *solution* is a complete instantiation of variables from X satisfying all the constraints in C.

Consistency. If a CSP has at least one solution, it is said that the CSP is *satisfiable* or *consistent*, otherwise we say that it is inconsistent.

2.2.2 Formulations of a CSP

Solving a constraint satisfaction problem (CSP) must be carried out in two different phases:

- Modeling the problem as a constraint satisfaction problem. In this phase, the problem is modeled by means of a CSP syntaxis, that is, by a set of variables, domains and constraints.
- Solving the resultant constraint satisfaction problem. Once the problem is modeled as a CSP, there exist many constraint satisfaction techniques to solve it.

Generally, a problem can be formulated in many different manners, even in natural language. For each problem, the representation of a potential solution and its corresponding interpretation implies the search space and its size. This is an important point to recognize: "*The size of the search space is not determined by the problem; it's determined by your representation and the manner in which you handle this encoding*" (Michalewicz & Fogel, 2000). Following, we present classical examples (Van Hentenryck, 1989)(Bistarelli, 2004)(Ruttkay, 1998), which can be modeled in two different ways. We can observe the advantage of an appropriate model to be solved more efficiently.

A crypto-arithmetic puzzle

We consider the classical "crypto-arithmetic" puzzle: SEND + MORE = MONEY. The variables {*s, e, n, d, m, o, r, y*} represent digits between 0 and 9 and the task is finding values for them such that the following arithmetic operation is correct:

$$\begin{array}{r} s\,e\,n\,d \\ +\,m\,o\,r\,e \\ \hline m\,o\,n\,e\,y \end{array}$$

Though it is not stated in the rules directly, it is implied that zero can not be the first digit in any of the three numbers. Thus, the unique solution of the problem is 9567 + 1085 = 10652. The easier way to model the problem is by having a variable for every letter, where the variable stands for the digit associated with the letter ({*0,...,9*}). The constraints are obvious from the problem specification taken into account that all variables must be assigned to different values. Thus the constraints are:

* $10^3(s+m)+10^2(e+o)+10(n+r)+d+e=10^4m+10^3o+10^2n+10e+y$;
* $all-different(s,e,n,d,m,o,r,y)$;

This formulation of the problem is not very appropriate for many search algorithms such as simple Backtracking (BT)[1], due to the fact that all variables must be previously assigned to check these constraints. Thus, the search space can not be pruned to improve the search process.

Following, we present a different formulation of the problem. This model decomposes each constraint into a set of constraints. We assume that zero can not be the first digit in any of the three numbers so that m and s cannot be fixed to 0. Due to the topology of the problem, m from *money* can only take value 0 or 1, but due to the former assumption m must be fixed to 1 then the domain of s is reduced to $\{1,...,9\}$. This new model includes three additional variables c_1, c_2, c_3. The domains of the variables are now:

$e, n, d, o, r, y : \{0, ..., 9\}$

$s : \{1, ..., 9\}$,

$m : \{1\}$,

$c_1, c_2, c_3 : \{0, 1\}$

and the above equation constraint can be decomposed to the following constraints:

- $e+d=y+10c_1$;
- $c_1+n+r=e+10c_2$;
- $c_2+e+o=n+10c_3$;
- $c_3+s+m=10m+o$;
- $s\neq e, s\neq n, ..., o\neq y, r\neq y$;

The second model has the advantage that the constraints are composed by a smaller number of variables than in the previous model, so that these constraints can be more efficiently checked during the backtracking search and thus inconsistencies can be found earlier.

The 8-queens problem

The second example is the popular 8-queens problem. The objective is to place 8 queens on the chess board such that they do not attack each other. In order to formulate this problem as a CSP, the location of the queens should be given by variables, and the fact that they don't attack each other must be modeled in terms of constraints.

One way to do this is to assign a variable to each queen. As the 8 queens must be placed in 8 different columns, we can identify each queen by its column, and represent its position by a variable which indicates

[1]This algorithm will be showed in the next section

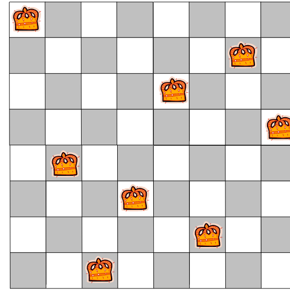


Figure 1 Example of a Constraint Satisfaction problem (8-queens).

the row of the queen. The domain of each of the variables is the number of available rows. Thus the domain of $x_1, ..., x_8$ is $1, 2, ...8$. For any two different variables the following two constraints must hold, expressing that the queens should be in different rows and on different diagonals:

$$x_i \neq x_j$$
$$|x_i - x_j| \neq |i - j|$$

In this formulation of the problem, we have to find a solution out of the total possible instantiations of the variables, which is 8^8. This formulation, though seems natural, does contain the following trick: a part of the requirements of the problem is reflected in the representation, not in the constraints. We could have used the most straightforward representation, namely identifying the squares of the chess board by the numbers $1, 2, ..., 64$, and having 8 variables for each of the 8 queens all with the domain $\{1, 2, ..., 64\}$. In this case, the "different columns" requirement should be expressed too by constraints, and all the three types of constraints become more intrinsic to formulate (Ruttkay, 1998). The total number of possible arrangements becomes as large as 64^8, containing a configuration of queens multiple times due to the identification of the 8 queens. So we have many reasons to prefer the first representation over the second one.

2.3 Consistency Techniques

Consistency techniques were first introduced for improving the efficiency of picture recognition programs, by researchers in artificial intelligence (Waltz, 1972). The number of possible combinations can be huge, while only very few are consistent. By eliminating redundant values from the problem definition, the size of the solution space decreases. Reduction of the problem can be done once, as a pre-processing step for another algorithm, or step by step, interwoven with the exploration of the solution space by a search algorithm. In the latter case, subsets of the solution space may be cut off, saving the search algorithm the effort of systematically investigating the eliminated elements, which otherwise would happen, even repeatedly. If, as a result of reduction, any domain becomes empty, then it is known immediately that the problem has no solution (Ruttkay, 1998).

Local inconsistencies are single values or combination of values for variables that cannot participate in any solution because they do not satisfy some consistency property. For instance, if a value a of variable x is not compatible with all the values in a variable y that is constrained with x, then a is inconsistent and this value can be removed from the domain of the variable x.

One should be careful with not spending more effort on reduction than what will "pay off" in the boosted performance of the search algorithm to be used to find a solution of the reduced problem. The reduction algorithms eliminate values by propagating constraints. The amount of constraint propagation is characterized by the consistency level of the problem, hence these algorithms are also called consistency algorithms. The iterative process of achieving a level of consistency is sometimes referred to as the relaxation process, which should not be mixed up with relaxation of constraints.

Consistency techniques have a long tradition in CSP research. Below we introduce the most well-known and widely used algorithms for binary CSPs.

- A CSP is *node-consistent* if all the unary constraints hold for all the elements of the domains. The straightforward node-consistency algorithm (NC), which removes the redundant elements by checking the domains one after the other, has $O(dn)$ time complexity, where d is the maximum size of the domains. Thus, enforcing this consistency ensures that all values of the variable satisfy all the unary constraints on that variable.
- A CSP is *arc-consistent* if for any pair of constrained variables x_i, x_j, for every value a in D_i there is at least one value b in D_j such that the assignment (x_i, a) and (x_j, b) satisfies the constraint between x_i and x_j. Any value in the domain D_i of variable x_i that is not arc-consistent can be removed from D_i since it cannot be part of any solution.
  The *Revise-Domain procedure* given below eliminates those elements from the domain of a variable x_i for which a given binary constraint cannot be satisfied. (The given pseudo-programs are based on ones in (Tsang, 1993)).

```
Revise-Domain((x_i, x_j), domains, constraints)
  reduced ←FALSE
  D_{x_i} ← get-domain (x_i, domains)
  D_{x_j} ← get-domain (x_j, domains)
  while not all-checked D_{x_i} do
    a ← select-not-checked-value(D_{x_i})
    if not good-value(a, D_{x_j}, contraints) then
      D_{x_i} ← D_{x_i}\{a}
      reduced ←TRUE
  return reduced
```

  Revise-Domain, when applied to pair (x_i, x_j), deletes all the values from the domain of x_i which are not compatible with values from the domain of x_j. The domain of x_j will not be modified by this procedure. The boolean value *reduced* indicates whether a value is deleted or not. Running the *Revise-Domain procedure* for all the binary constraints, may not be enough to assure that the reduced problem is arc-consistent. In fact, it may be necessary to run it more than once on some constraints. Below we give a widely-used arc-consistency algorithm, called AC-3:

```
AC-3 (variables,domains, constraints)
  NC(variables,domains, constraints)            // Node-Consistency is applied
  arcs ← binary-constraints(constraints)
  while arcs ≠ ∅ do
    (x_i, x_j) ← select-one-element(arcs)
    if Revise-Domain ((x_i, x_j), domains, constraints) then
      arcs ←arcs ∪(binary-constraints(constraints)∩{(x_k, x_i)|x_k ≠ x_j})
  return domains
```

  This algorithm reconsiders all binary constraints over (x_i, x_k) if the Revise-Domain procedure has modified the domain of variable x_i. Thus, when AC-3 ends, no domain can be reduced further and thus the problem is arc-consistent.
  Arc-consistency has become very important in CSP solving and is in the heart of many constraint programming languages.
- A CSP is *path-consistent*, if for every pair of values a and b for two variables x_i and x_j, such that the assignments of a to x_i and b to x_j satisfies the constraint between x_i and x_j, there exist a value for each variable along any path between x_i and x_j such that all constraints along the path are satisfied.
  When a path-consistent problem is also node-consistent and arc-consistent, then the problem is said to be strongly path-consistent.
  Path-consistency is not widely used in CSP solving due to its high time and space complexity. However, it is an important consistency in CSPs with special properties such as temporal CSPs (Dechter *et al.*, 1991).

2.4 Search Techniques

A CSP can be solved by systematically exploring the solution space by an uninformed search. The algorithms instantiate variables one after the other in such a way that the partial instantiation is always consistent. If this is not possible for a variable on turn, that is, all the possible values are in conflict with some earlier assignment, then backtracking takes place.

The simplest search scenario is when the order of variables as well as the order of values to be considered next is fixed, and if a dead end occurs, then the latest instantiation is reconsidered. In this uninformed search, a depth-first backtrack search algorithm (Bitner & Reingold, 1975) is used. While the worst-case time complexity of backtrack search is exponential, several heuristics to reduce its average-case complexity have been proposed in the literature (Dechter & Pearl, 1988). However, determining which algorithms are superior to others remains difficult. Theoretical analysis provides worst-case guarantees which often do not reflect average performance. For instance, a backtracking-based algorithm that incorporates features such as variable ordering heuristics will often in practice have substantially better performance than a simpler algorithm without this feature (Haralick & Elliot, 1980), and yet the two share the same worst-case complexity. Also, constraint propagation techniques such as consistency techniques (node consistency, arc-consistency, etc.,) are very appropriate for solving CSPs more efficiently.

Following, we present some well-known systematic search algorithms as well as the main heuristics that improve backtracking-based algorithms (*variable ordering* and *value ordering*) (Sadeh & Fox, 2003). They play an important role in CSP solving and have drawn a lot of attention mainly in binary problems. More information can be found in (Barták, 1998) (Rossi *et al.*, 2006).

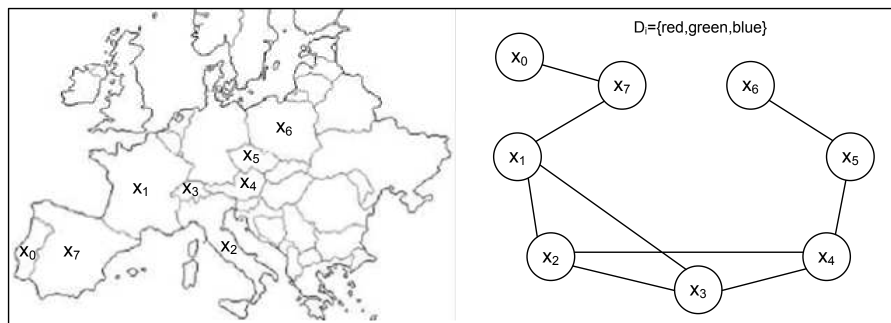


Figure 2 Example of CSP: a modified Europe coloring problem. The domain of each node (Country) is: {red, blue, green}. Adjacent nodes (Countries) must be assigned different colors

2.4.1 Complete Search Algorithms

Most algorithms for solving CSPs search systematically through the possible assignments of values to variables. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem is insoluble. The disadvantage of these algorithms is that they may take a very long time to do so. The actions of a search algorithm can be described by a search tree. We illustrate this with a toy example shown in Figure 2. The CSP in this figure is a small graph-coloring problem, in which the goal is to assign a color to each variable (country) such that connected variables (neighboring countries) do not share the same color.

Generate and Test (GT)

The generate-and-test method originates from the mathematical approach to solving combinatorial problems. First, the GT algorithm guesses the solution and, then, it tests whether this solution is correct, i.e., whether the solution satisfies the original constraints. In this paradigm, each possible combination of the variable assignments is systematically generated and tested to see if it satisfies all the constraints. The first combination that satisfies all the constraints is the solution. The number of combinations considered by this method is the size of the Cartesian product of all the variable domains.

Following the ordering $(X_0; X_1; X_2; X_3; X_4; X_5; X_6; X_7)$ for the CSP described in Figure 2, the GT algorithm assigns value to all variables, for example $(X_0 = red; X_1 = blue; X_2 = red; X_3 = green; X_4 = blue; X_5 = red; X_6 = green; X_7 = red)$, and then it checks whether all constraints are satisfied.

The main disadvantage is that it is not very efficient because it generates many assignments of values to variables which are rejected in the testing phase. In addition, the generator leaves out the conflicting instantiations and it generates other assignments independently of the conflict. Visibly, one can get far better efficiency, if the validity of the constraint is tested as soon as its respective variables are instantiated. In fact, this method is used by the backtracking approach.

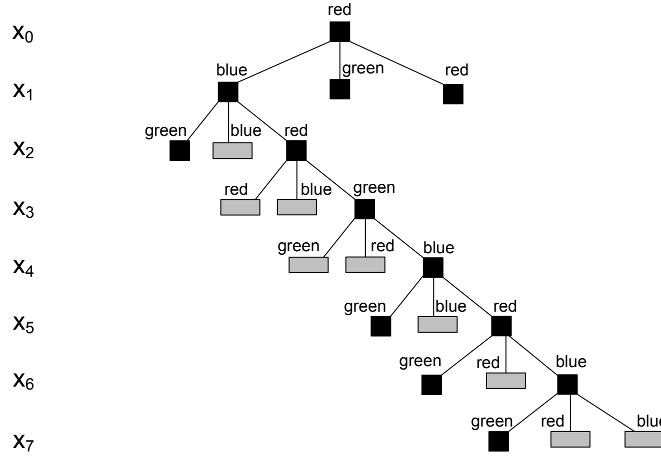


Figure 3 Part of the search tree explored by backtracking on the CSP presented in Figure 2. Only a part of the search tree below $X_0 = red$ and $X_1 = blue$ is drawn. A black square denotes an instantiation from which further search can continue. A gray rectangle denotes a value that is not compatible with some previous value.

Backtracking Search (BT)

A simple algorithm for solving a CSP is backtracking search (Bitner & Reingold, 1975) (Golomb & Baumert, 1965). Backtracking works with an initially empty set of consistent instantiated variables and tries to extend the set to a new variable and a value for that variable. If successful, the process is repeated until all variables are included. If unsuccessful, another value for the most recently added variable is considered. Returning to an earlier variable in this way is called a backtrack. If that variable doesn't have any further values, then the variable is removed from the set, and the algorithm backtracks again. The simplest backtracking algorithm is called chronological backtracking because at a dead-end the algorithm returns to the immediately earlier variable in the ordering.

The pseudo code of the basic Backtracking Search (BT) algorithm is given:

Recursive BT(*free-variables, instantiation, domains*)

  if *free-variables* $= \emptyset$ **then** return *instantiation*
  $x \leftarrow$*select-one-variable(free-variables)*
  $D_x \leftarrow$ *get-domain*$(x, domains)$
  while $D_x \neq \emptyset$ **do**
    $a \leftarrow$ *select-one-value*(D_x)
    if $consistent(instantiation \cup (x, a))$ **then**
      $solution \leftarrow$ **BT**(*free-variables*$\setminus\{x\}$, $instantiation \cup (x, a), domains)$
      if $solution \neq \emptyset$ **then** return *solution*
  return $\emptyset$

Free-variables are the variables that are not instantiated yet by the data structure *instantiation*. We call the the Recursive BT procedure with $free - variables = X$, $instantiation = \emptyset$, and $domains = D$,

where D are in the initial domains of the variables. When the procedure stops, it returns a consistent instatiation of all the variables in the structure *solution*, if the problem is solvable, and $\emptyset$ otherwise.

Backtracking can be seen as a merge of the generate and test phases from the GT approach. In the BT method, variables are instantiated sequentially and as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial solution violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. Consequently, backtracking is strictly better than generate-and-test. However, its running time complexity for most nontrivial problems is still exponential.

Figure 3 shows a part of the search tree expanded when backtracking processes the CSP described in Figure 2, using the ordering $(X_0; X_1; X_2; X_3; X_4; X_5; X_6; X_7)$.

There are three major drawbacks of the standard backtracking scheme. One is thrashing, i.e., repeated failure due to the same reason. Thrashing occurs because the standard backtracking algorithm does not identify the real reason of the conflict, i.e., the conflicting variables. Therefore, search in different parts of the space may keep failing for the same reason. Thrashing can be avoided by intelligent backtracking, i.e., by a scheme in which backtracking is done directly to the variable that caused the failure. The other drawback of backtracking is having to perform redundant work. Even if the conflicting values of variables are identified during the intelligent backtracking, they are not remembered for immediate detection of the same conflict in a subsequent computation. There is a backtracking-based method that eliminates both of the above drawbacks of backtracking. This method is traditionally called dependency-directed backtracking and is used in truth maintenance systems. It should be noted that using advanced techniques adds other expenses to the algorithm that has to be balanced with the overall advantage of using them. Finally, the basic backtracking algorithm still detects the conflict too late as it is not able to detect the conflict before the conflict really occurs, i.e., after assigning the values to all the variables of the conflicting constraint. This drawback can be avoided by applying consistency techniques to forward check the possible conflicts. We will analyze these techniques in the next section.

Look-Back Algorithms

Backtracking can suffer from thrashing; the same dead-end can be encountered many times. If X_i is a dead-end, the algorithm will backtrack to X_{i-1}. Suppose a new value for X_{i-1} exists, but that there is no constraint between X_i and X_{i-1}. The same dead-end will be reached at X_i again and again until all values of X_{i-1} have been exhausted.

Look-back algorithms try to exploit information from the problem to behave more efficiently in dead-end situations. Like BT, look-back algorithms perform consistency checks *backwards* (between the current variable and past variables).

Backjumping (BJ) (Gaschnig, 1979) is an algorithm similar to BT except that it behaves in a more intelligent manner when a dead-end (X_i) in found. Instead of backtracking to the previous variable (X_{i-1}), BJ backjumps to the deepest past variable $X_j, j < i$ that is in conflict with the current variable X_i. It is said that variable X_j is in conflict with the current variable X_i if the instantiation of X_j precludes one of the values in X_i. Changing the instantiation of X_j may make it possible to find a consistent instantiation of the current variable. Thus, BJ avoids any redundant work that BT does by trying to reassign variables between X_j and the current variable X_i.

Conflict-directed backjumping (Prosser, 1993), backmarking (Gaschnig, 1977), learning (Frost & Dechter, 1994) are examples of look-back algorithms.

Look-Forward Algorithms

As we have explained, look-back algorithms try to enhance the performance of BT by a more intelligent behavior when a dead-end is found. Nevertheless, they still perform only backward consistency checks and they ignore the future variables.

Look-forward algorithms make *forward* checks at each step of the search. Let us assume that when searching for a solution, the variable X_i is given a value which excludes all the possible values for the variable X_j. In case of uninformed search this will only turn out when X_j will be considered to be instantiated. Moreover, in case of BT, thrashing will occur: the search tree will be expanded again and

again till X_j, as long as the level of backtracking does not reach X_i. Both anomalies could be avoided by recognizing that the chosen value for X_i cannot be part of a solution as there is no value for X_j which is compatible with it. Lookahead algorithms do this, by accepting a value for the current variable only if after having looked ahead, it could not be seen that the instantiation would lead to a dead-end. When checking this, problem reduction can also take place, by removing the values from the domain of the future variables which are not compatible with the current instantiation. The algorithms differ in how far and thorough they look ahead and how much reduction they perform.

Forward-checking (FC) (Haralick & Elliot, 1980) is one of the most common look-forward algorithms. It checks the satisfiability of the constraints, and removes the values which are not compatible with the current variable's instantiation. At each step, FC checks the current assignment against all the values of future variables that are constrained with the current variable. All values of future variables that are not consistent with the current assignment are removed from their domains. If a domain of a future variable becomes empty, the assignment of the current variable is undone and a new value is assigned. If no value is consistent then backtracking is carried out. Thus, FC guarantees that at each step the current partial solution is consistent with each value in each future variable. Thus, FC can identify dead-ends and prune the search space sooner.

2.4.2 Incomplete Search Algorithms

Besides systematic search techniques such as those described above, that always return a solution if there is one, we may sometimes want to use other kinds of techniques, with different features. Complete algorithms try to explore the whole search space in order to find one solution, all the solutions or to detect that the CSP is not consistent (Kumar, 1992),(Tsang, 1993). Complete methods are mainly based on local consistency techniques which allow the algorithms to prune the search space by deleting inconsistent values from variables domains. A complete solver usually builds a search tree by applying domain reduction, splitting and enumeration. Unfortunately, these algorithms require an important computational effort and therefore encounter some difficulties with large scale problems.

Incomplete search methods (Michalewicz & Fogel, 2000) do not explore the whole search space. They search the space either non-systematically or in a systematic manner, but with a limit on some resource. Unfortunately, these approaches do not ensure to collect all the solutions nor to detect inconsistency, but their computational time is significantly reduced. However, they may be sufficient when just some solution is needed. Another application is to seek a feasible solution of an optimization problem.

This class of methods, known as metaheuristics, covers a very large panel of resolution paradigms from evolutionary algorithms to local search techniques.

There are two basic approaches for incomplete search (Hatamlou & Meybodi, 2007):

- The constructive algorithms gradually extend a partial solution to a complete one. They are based on tree search algorithms for satisfaction problems and may benefit from constraint propagation.
- The Iterative repair methods start with an initial solution, (found by another approach) and they incrementally alter the values to get a better one, and eventually an optimal or at least a good enough solution.

The non-systematic nature of these methods generally does not provide the guarantee of completeness, but they are often able to get quickly close-to-the-optimal solutions and overcome the complete algorithms based on tree search. The search will move throughout all assignments and the quality of the assignment will be determined by the number of violated constraints. Important iterative repair methods are the so called local search methods (Michalewicz & Fogel, 2000).

Local search does not instantiate one variable at a time, but start with a complete assignment to all the variables, and then modifies it slightly to pass to a new complete assignment which is closer to be a solution. The modification performed in one local search step usually involves a small part of the current complete assignment (often a single variable). When there is no modification (among the allowed ones) that produces a new assignment which is better than the current one, the algorithm stops returning the

current assignment of values to variables. However, the returned assignment could be non-optimal, since we just looked in the *neighborhood* of the current assignment, and not everywhere in the solution space.

2.5 Variable and Value Ordering Algorithms

We have assumed, in describing the tree-search algorithms in the previous sections, that the order of the variables and values is static, that is, unchanging as the algorithm proceeds. In practice this is not necessarily the case, which requires modifying some algorithms. A search algorithm for constraint satisfaction requires the order in which variables are to be considered to be specified as well as the order in which the values are assigned to the variable on backtracking. Choosing the right order of variables (and values) can noticeably improve the efficiency of constraint satisfaction.

Variable Ordering

The order in which variables are assigned during search may have a significant impact on the size of the search space explored. The ordering may be either a static ordering, in which the order of the variables is specified before the search begins, and it is not changed thereafter, or a dynamic ordering, in which the choice of next variable to be considered at any point depends on the current state of the search.

Waltz (Waltz, 1975) proposed a variable ordering heuristic, motivated by the *fail-first principle*. In general constraint satisfaction terms, we can interpret this principle as *select a variable which has few values in the domain*". Intuitively, if we want the size of the search tree to be as small as possible, it is probably better to put nodes with small branching factor first.

The minimum width (MW) heuristic (Freuder, 1982) orders the variables from last to first by selecting, at each stage, a variable in the constraint graph that connects to the minimal number of variables that have not been selected yet. For instance, in the CSP from Figure 2, we could choose X_0 and X_6 to be the last variables, since each is connected to one other variable. If we select X_0 to be last, then there is a choice between X_6 and X_7 to be second-to-last. Continuing in this manner, the final ordering might be X_1; X_2; X_3; X_4; X_5; X_7; X_6; X_0. There is usually more than one minimum width ordering of a CSP.

Under a dynamic variable ordering scheme, the order of the variables is determined as search progresses, and may differ from one branch of the search tree to another. Most dynamic variable ordering heuristics are based on the above *fail-first principle*. The main idea of the heuristic is to select the future variable with the smallest remaining domain. Haralick and Elliot (Haralick & Elliot, 1980) show via a probabilistic analysis that choosing the variable with the smallest number of remaining values minimizes the probability that the variable can be consistently instantiated, and thus minimizes the expected length or depth of any branch. Dynamic ordering is not feasible for all search algorithms, e.g., with simple backtracking there is no extra information available during the search that could be used to make a different choice of ordering from the initial ordering. However, with forward checking, the current state includes the domains of the variables as they have been pruned by the current set of instantiations, and so it is possible to base the choice of next variable on this information.

Value Ordering

Comparatively little work has been done on general heuristics for value ordering. The basic idea is based on the *succeed-first principle* that can be resumed as *select the value for the current variable which is most likely to succeed*, that is, try to identify the branch that is more likely to lead to a solution. Thus, once the decision is made to instantiate a variable, it may have several values available. However, if no consistent assignment exist then the order of values is not important due to all values must be tried.

A well known value ordering heuristic is the *min-conflict* heuristic. This heuristic associates with each value a of the current variable the total number of values in the domains of the adjacent future variables that are incompatible with a. The value selected is the one associated with the lowest sum.

A different value ordering will rearrange the branches emanating from each node of the search tree. This is an advantage if it ensures that a branch which leads to a solution is searched earlier than branches which lead to death ends. For example, if the CSP has a solution, and if a correct value is chosen for each variable, then a solution can be found without any backtracking.

3 Planning and Scheduling

A scheduling task is usually formulated as a problem of allocating known activities to scarce resources, such as machines, over time. The scheduling task is typically preceded by a planning task, where the problem is to decide which activities are necessary to achieve a given goal. Frequently, especially in the industrial environment, the notions of planning and scheduling are used as substitutes - planning is usually used in the meaning of long term scheduling, for example to allocate contract work to departments for the horizon of months, while scheduling means detailed scheduling, for example allocating jobs or operations to machines with minute precision. In this introductory survey, we will stick to the original meaning of planning and scheduling where planning means finding activities to achieve the goal and scheduling means allocating the activities to time and resources.

Despite a close relationship between planning and scheduling, both areas were studied separately for a long time. Planning is a typical topic of artificial intelligence where it is motivated by general problem solving. Scheduling is traditionally studied by operations research community and it is motivated by everyday industrial requirements for efficient production. Different origins are naturally reflected in different solving approaches. Planning techniques are usually based on search and they cover a broad range of problems, while scheduling techniques are frequently ad-hoc algorithms dedicated to a particular scheduling problem. This section surveys these traditional planning and scheduling techniques.

3.1 Planning

Planning is an important aspect of rational behavior and it is a fundamental topic of artificial intelligence since its beginning. Planning capabilities are necessary for autonomous controlling of vehicles of many types including space ships (Muscettola *et al.*, 1998) and submarines (McGann *et al.*, 2008), but we can also find planning problems is areas such as manufacturing, games or even printing machines (Ruml *et al.*, 2005). In this section we introduce the planning terminology and present the basic techniques for solving classical planning problems.

Classical planning deals with finding a sequence of actions that transfer the world from some initial state to a desired state. The state space is large but finite. It is also *fully observable* (we know precisely the state of the world), *deterministic* (the state after performing the action is known), and *static* (only the entity for which we plan changes the world). Moreover, we assume the actions to be instantaneous so we only deal with action sequencing. Naturally, there exist extensions of planning problems dealing with durative and parallel actions with uncertain effects, the state of the word may not be fully known or may be changed by other entities such as nature. However, these extensions are out of scope of this introduction, for a detailed survey see (Ghallab *et al.*, 2004).

Typically, the world *state* is described as a set of predicates that hold in the state, such as $location(robot_1, city_{23})$ saying that $robot_1$ is located in $city_{23}$. In other words, for each predicate and for each state we describe whether the predicate holds in the state or not. This is called a classical representation of planning problems. *Actions* are described using a triple ($Prec, Eff^+, Eff^-$), where $Prec$ is a set of predicates that must hold for the action to be applicable (preconditions), Eff^+ is a set of predicates that will hold after performing the action (positive effects), and Eff^- is a set of predicates that will not hold after performing the action (negative effects). For example, action $move(robot_1, city_{12}, city_{23})$, describing that $robot_1$ moves from $city_{12}$ to $city_{23}$, is specified as a triple $(\{location(robot_1, city_{12})\}, \{location(robot_1, city_{23})\}, \{location(robot_1, city_{12})\})$. Formally, action a is applicable to state s if $Prec(a) \subseteq s$. The result of applying action a to state s is a new state $\gamma(s, a) = (s - Eff^-(a)) \cup Eff^+(a)$. Notice that this description assumes a *frame axiom*, that is, other predicates than those mentioned among the effects of the action are not changed by applying the action. The set of predicates together with the set of actions is called a *planning domain*. We assume both sets of predicates and actions to be finite. The *goal* is specified as a set of predicates that must hold in the goal state, that is, if g is a goal then any state s such that $g \subseteq s$ is a goal state. The *classical planning problem* is defined by the planning domain, the initial state s_0, and the goal g, and the task of planning is to find a

sequence of actions $< a_1, a_2, ..., a_n >$, called a *plan*, such that a_1 is applicable to the initial state s_0, a_2 is applicable to state $\gamma(s_0, a_1)$ etc., and $g \subseteq \gamma(\gamma(\gamma(s_0, a_1), a_2)..., a_n)$.

There exists an alternative to the above logical formalism that is based on so called *multi-valued state variables*. For each feature of the world, there is a variable describing this feature, for example $location(robot_1, S)$ describes the position of $robot_1$ at state S. Instead of specifying the validity of the predicate in some state S, say $location(robot_1, city_{23})$, we can specify the value of the state variable in a given state, in our example $location(robot_1, S) = city_{23}$. Hence the evolution of the world can be described as a set of state-variable functions where each function specifies the evolution of the values of certain state variable. The actions are described as entities changing the values of state variables. We can still use preconditions specifying required values of certain state variables, but the positive and negative effects are merged to effects of setting the values of certain state variables. Notice that this multi-valued formulation is more compact than the logical formulation, where, for example, one needs to express explicitly that if $robot_1$ is in $city_{23}$ then it is not present in another location. For example, the action of moving $robot_1$ from $city_{23}$ to $city_{24}$ needs to explicitly describe (in negative effects) that, after performing the action, the predicate $location(robot_1, city_{23})$ is no more valid. In the multi-valued representation assigning value $city_{24}$ to state variable $location(robot_1, S)$ implicitly means that $robot_1$ is not at a different location at state *S*.

As we already mentioned, automated planning is usually interested in domain-independent solving approaches. We will present two classical domain-independent planning approaches, namely state-space planning and plan-space planning. There exist also other planning techniques, for example based on finding decompositions of the goal task to primitive actions so called hierarchical task network planning first introduced in (Tate, 1977), but their description is beyond this introduction.

The simplest and most natural planning technique is based on exploring the state space by a search algorithm - this is called *state-space planning* (Fikes & Nilsson, 1971). The search space is typically a subset of the state space where the nodes correspond to states of the world, arcs correspond to state transitions described by actions, and plans are equivalent to paths in the state-space graph. Hence, planning reduces to finding a path from the initial state to one of the goal states in the state space graph. Obviously, the main problem is that complete state space cannot be kept in memory because, even for small planning problems, the number of states is extremely large (larger than the number of atoms in the universe). Nevertheless, as described above, we can represent states and state transitions implicitly using finite information.

One of the simplest planning algorithms is the *forward-search algorithm* used in successful planners such as FF (Hoffmann & Nebel, 2001). In forward search we start with the initial state, we select an applicable action, and we go to the next state until one of the goal states is reached. The formal code of the forward-search planning algorithm is as follows:

```
forward-planning(A, s0, g)
// A is a set of all actions, s0 is an initial state, and g is a goal
    s ← s0
    π ← empty plan
    while ¬g ⊆ s do                              // while we did not reach goal
        applicable ← {a|a ∈ A, Prec(a) ⊆ s}      // actions applicable to state s
        if applicable = ∅ then return failure    // dead-end
        non-deterministically select a ∈ applicable
        s ← γ(s, a)                              // find next state
        π ← π.a                                  // add action to plan
    end while
return π
```

Naturally, the actual implementation of the forward-search algorithm should be deterministic; one can use techniques like depth-first search, best-first search, or other search algorithms. For deterministic

implementations the branching factor (the number of actions applicable to a given state) plays an important role as it might be large and heuristics are necessary to guide the algorithm.

Opposite to forward-search, we can start search in the goal state and try to reach the initial state (Fikes & Nilsson, 1971). However, the problem is that there could be more goal states. Nevertheless, we can search in the space of goals rather than in the space of states. Each node corresponds to some goal, that is, a set of states. To apply this backward-search approach, it is first necessary to define which actions are *relevant for the goal*. Action *a* is relevant for goal *g* if $Eff^+(a) \cap g \neq \emptyset$ and $Eff^-(a) \cap g = \emptyset$, that is, action *a* contributes to goal *g* and can be the last action in the plan. If action *a* is relevant for *g*, then we can define the sub-goal that must be satisfied by the state before we can apply action *a* to reach goal *g*. This sub-goal is also called a regression set (hence backward-search planning is also called regression planning) and it is formally defined in the following way: $\gamma^{-1}(g, a) = (g - Eff^+(a)) \cup Prec(a)$. Now it is straightforward to write the code of the backward-search (regression) planning algorithm:

```
backward-planning(A, s0, g)
// A is a set of all actions, s0 is an initial state, and g is a goal
    π ← empty plan
    while ¬g ⊆ s0 do
        relevant ← {a|a ∈ A, a is relevant for g}     // actions relevant for g
        if relevant = ∅ then return failure           // dead-end
        non-deterministically select a ∈ relevant
        g ← γ^-1(g, a)                                // find a sub-goal
        π ← a.π                                       // add action before the plan
    end while
return π
```

Similarly to the forward-search algorithm, a deterministic implementation is necessary for backward-search. The branching factor is typically smaller for backward-search than for forward-search, but the definition of the regression set becomes more complicated for more complex view of planning problems. That is the reason why backward search is used less frequently in current planning systems.

So far we assumed that plans are sequences of actions, but notice that the sequence is used primarily to ensure that causal relations between the actions hold (Sacerdoti, 1990). The *causal relation* between actions *a* and *b* says that action *a* has some predicate *p* among its positive effects that is used as a precondition of action *b*, we write $a \rightarrow^p b$. Hence action *a* must be processed before action *b* in the plan and there must not be any action between *a* and *b* that deletes predicate *p*. Clearly, to capture such relations it is enough to specify a partial order of actions rather than the linear order. This is the basic idea behind *plan-space planning* where we are exploring partial plans until we obtain a complete plan (Penberthy & Weld, 1992). In plan-space planning we start with a plan containing just two actions: the initial state s_0 is modeled using a special action a_0 such that $Prec(a_0) = Eff^-(a_0) = \emptyset$, $Eff^+(a_0) = s_0$, the goal is modeled using special action a_∞ such that $Prec(a_\infty) = g$ and $Eff^-(a_\infty) = Eff^+(a_\infty) = \emptyset$, and action a_0 is before a_∞. Now we will be refining the partial plan by finding a causal relation for each predicate in preconditions of actions in the partial plan, so called *open goals*. This can be done by defining the causal relation between the actions already in the plan (causal relations also define partial order between the actions) or by adding a new action to the plan. The new actions must always be after a_0 and before a_∞ in the partial order of actions. Notice that a causal relation $a \rightarrow^p b$ can be *threatened* by another action *c* in the partial plan that deletes *p* and can be ordered between *a* and *b* in the partial order. This threat can be resolved either by ordering *c* before *a* or by ordering *c* after *b*.

In summary, plan-space planning is based on exploring the space of partial plans starting with a trivial plan containing only the actions a_0 and a_∞ and refining the plan by adding causal relations for open goals, which may also add new actions to the plan, and removing threats until we obtain a plan with no open goals and no threats. It is possible to prove that the partial plan with no threats and no open goals is a solution plan, in particular, any linearly ordered sequence of actions that follows the partial order is a plan. The following pseudo code shows the basic plan-space planning algorithm.

```
plan-space-planning(π)
// π is an initial plan with actions a_0 and a_∞
    flaws ← OpenGoals(π) ∪ Threats(π)                // flaws to be repaired
    while flaws ≠ ∅ do
        select any flaw φ ∈ flaws
        resolvers ← Resolve(π, φ)           // possible ways to resolve the flaw
        if resolvers = ∅ then return failure          // dead-end
        non-deterministically select ρ ∈ resolvers
        π ← Refine(π, ρ)                              // resolve the flaw
        flaws ← OpenGoals(π) ∪ Threats(π)            // flaws to be repaired
    end while
return π
```

Similarly to state-space planning, a deterministic implementation of the above algorithm scheme can be realized via various search algorithms. Notice that the selection of flaw (open goal or threat) does not bring any choice point; we need to resolve all flaws. The choice points appear only in the ways to resolve a particular flaw (for example there are two choices to resolve the threat - see above). There is one significant difference between plan-space planning and the state-space planning: the search space is possibly infinite because it consists of all possible partial plans and there is no upper bound on the length of the plan. This feature of the plan-space planning must reflect in the selected search procedure (for example, breadth-first search guarantees finding the shortest plan).

This section described the fundamental planning techniques. For simplicity reasons we assumed several restrictions about the planning problem. Also, we assumed that all actions are given as the input which is not always practical. For example, if we have *m* robots and *n* locations then we need $m.n^2$ actions just to describe possible movements of the robots between two locations. In practice, the description of the actions can be more compact by using operators that define the templates for the actions, for example by moving some robot from some location to another location. The actions can be obtained by filling the actual objects (robots and locations in our example) to the template. This representation together with details about above described and other planning techniques can be found in (Ghallab *et al.*, 2004).

3.2 Scheduling

Scheduling concerns with the allocation of resources to activities with the objective of optimizing some performance measures. Depending on the situation, resources could be machines, humans, runways, processors etc., activities could be manufacturing operations, duties, landings and take-offs, computer programs etc., and objectives could be minimization of the schedule length, maximization of resource utilization, minimization of delays, and others.

Scheduling has been studied since 1950s when researchers were faced with problems of efficient management of operations in workshops (Leung, 2004). The problems studied at that time were relatively simple and a number of efficient algorithms providing optimal solutions were proposed. In late 1960s, scheduling problems were encountered in the area of computer science where the problem of efficient utilization of scarce computational resources became very important. As scheduling problems became more complicated, researchers were unable to develop efficient algorithms for them and the proposed techniques were essentially exponential in time. This is not so surprising, since many scheduling problems have been shown to be NP-hard. Nowadays, an increased attention is paid to approximation and stochastic algorithms that can provide some solutions even to hard problems. The history of scheduling is reflected in the style of research in the area. The focus is almost always on solving a specific scheduling problem or showing its complexity rather than on providing a general scheduling approach. The result is that we have a huge number of scheduling algorithms for a large number of specific problems. This makes scheduling very different from the approach of planning where the focus is on solving general planning problems rather than developing ad-hoc techniques for particular planning problems.

Let us now introduce some basic scheduling terminology more formally. Typically, the scheduling problem consists of a set of *n* jobs that can run on *m* machines. Each job *i* requires some processing time p_{ij} on a particular machine j. This part of a job is usually called an operation. The schedule of each job is an allocation of one or more time intervals (operations) to one or more machines. The scheduling problem is to find a schedule for each job satisfying certain restrictions and optimizing given objectives. The start time of job *j* can be restricted by a *release date* r_j, which is the earliest time at which job j can start its processing. A release date models the time when the job arrives at the system. Similarly, a deadline d_j can be specified for job j, which is the latest time by which the job j must be completed. More frequently, a *due date* δ_j is specified for job j which is an expected time of job completion. The job can complete later (or earlier) the due date, but it will incur a cost. Let C_j be the completion time of job j. Then *lateness* of job j is defined as $L_j = C_j - \delta_j$ and the *tardiness* of job j is defined as $T_j = max(0, L_j)$. Completion time, lateness, and tardiness are the typical participants in traditional objective functions expressing the quality of the schedule.

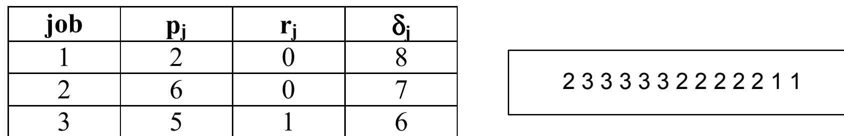
It is possible to specify precedence constraints between jobs, which express the fact that certain jobs must be completed before certain other jobs can start processing. In the most general case, the precedence constraints are represented as a directed acyclic graph, where each vertex represents a job and, if job i precedes job j, then there is a directed arc from i to j. If each job has at most one predecessor and at most one successor, then we are speaking about *chains*; they correspond to serial production. If each job has at most one successor then the constraints are referred to as an *intree*. This structure is typical for assembly production. Similarly, if each job has at most one predecessor, then the constraint structure is called an *outtree*. This structure is typical for food or chemical production where from the raw material we obtain several final products with different "flavors".

Each job (operation) requires certain resource(s) for its processing. Operations can be pre-assigned to particular resources and then we are looking for time of processing only. Or there may be several alternative resources to which the operation can be allocated, and then *resource allocation* is part of the scheduling task. Such alternative resources are either identical then we can union them into a so called *cumulative resource* that can process several operations in parallel until some given capacity, or the alternative resources may be different (for example processing time may depend on the resource). The resource, that can process at most one operation at any time, is called a *unary* or *disjunctive resource*. If processing of operation on a machine can be interrupted by another operation and then resumed possibly on a different machine, then the job is said to be *preemptable*.

There exists a special category of scheduling problems with a particular combination of precedence constraints, so called *shop problems*. If each job has its own predetermined route to follow, that is, the job consists of a chain of operations where each operation is assigned to a particular machine (the job may visit some machines more than once or never) then we are talking about *job-shop scheduling* (JSS). If the machines are linearly ordered and all the jobs follow the same route (from the first machine to the last machine) then the problem is called a *flow-shop problem*. Finally, if we remove the precedence constraints from the flow-shop problem, that is, the operations of each job can be processed at any order, then we obtain an *open-shop problem*.

As we mentioned above, the scheduling research focuses on solving specific scheduling problems, so it is crucial to have a good classification of scheduling problems. The most widely used notation to classify scheduling problems is the well known $\alpha|\beta|\gamma$ notation by Graham et al. (Graham *et al.*, 1979). The α field describes the machine environment, for example whether there is a single machine (1), *m* parallel identical machines (*Pm*), job-shop (*Jm*), flow-shop (*Fm*) or open-shop (*Om*) environment with *m* machines. The β field characterizes jobs and scheduling constraints and it may contain none entry or multiple entries. The typical representatives are restriction of the release dates (r_j) or deadlines (d_j) for jobs, assuming precedence constraints (*prec*) or allowing pre-emption of jobs (*pmtn*). Finally, the γ field contains the objective function to optimize, usually a single entry is present but more entries are allowed. The widely used objectives are minimization of the maximal completion time - a so called $makespan(C_{max})$ or minimization of maximal lateness (L_{max}).

There are many scheduling algorithms for solving particular scheduling problems. We selected few of them to demonstrate some basic principles of scheduling, namely exact polynomial algorithm for $1|r_j, pmtn|L_{max}$, exact exponential algorithm for $1|r_j|L_{max}$, and heuristic algorithm for $Jm||C_{max}$. Let us assume the problem $1|r_j|L_{max}$, that is the problem with a single machine where jobs have arbitrary release dates and the objective is to minimize the maximal lateness. This problem is important because it appears as a sub-problem in heuristic procedures for flow-shop and job-shop problems; unfortunately, the problem is known to be strongly NP-hard ((Pinedo, 2002), p. 43). However, if all release dates are identical, or all due dates are identical, or preemption is allowed, then the problem becomes tractable. These tractable problems can be solved polynomially by applying the *EDD (earliest due date)* rule by Jackson (Jackson, 1955) that says "select the job with the smallest due date and allocate it to the resource". We will present the preemptive version of EDD rule for the problem $1|r_j, pmtn|L_{max}$. In the pre-emptive case we collect all not-yet finished jobs with the earliest release date and among them we select the job with the earliest due date for processing and allocate the job to the earliest possible time. Processing of this job continues until the job is finished or another job becomes available (that is, the next release date is reached). At this point, the next job for processing is selected using the same principle (Figure 4). It could be the current job or a different job in which case the previous job is interrupted (unless it is already finished).



job	p_j	r_j	δ_j
1	2	0	8
2	6	0	7
3	5	1	6

Figure 4 A $1|r_j, pmtn|L_{max}$ problem (table) and it is solution (the sequence of numbers on the right indicates which job is being processed at each time point starting from time point 0)

We should realize now that if the solution of $1|r_j, pmtn|L_{max}$ is not using preemption then we also have a solution to $1|r_j|L_{max}$. In other cases, the preemptive solution provides a lower bound for the maximal lateness of the non-preemptive case. These lower bounds may be exploited when solving the non-preemptive case using the branch-and-bound algorithm. The algorithm basically explores possible sequences of jobs by constructing them from left to right. The lower bound is used to prune partial schedules that cannot be extended to an optimal schedule. The following code shows the algorithm formally.

```
solve(Jobs, T, Bound)
// Jobs is a set of jobs to be scheduled
// T is the first possible start time (initialized to 0 for non-negative release dates)
// Bound is a value of the objective function for the best so far solution (initialized to ∞ )
(Sched, LB) ← solvePreempt(Jobs, T) // find a schedule using pre-emptive EDD rule
if LB ≥ Bound then return (fail, Bound)          // no schedule better than Bound
if Sched is non-preemptive then return (Sched, LB)          // optimum found
First ← {j ∈ Jobs|¬i ∈ Jobs : max(T, r_i) + p_i ≤ r_j}
Best ← fail
foreach j ∈ First do
    (Sched, LB) ← solve(Jobs − {j}, max(T, r_j) + p_j, Bound)
    if LB < Bound then (Best, Bound) ← (Sched, LB)
end
return (Best, Bound)
```

A general job-shop scheduling problem $Jm||C_{max}$ is another NP-complete scheduling problem. The job-shop scheduling problem can be represented in a nice way by a disjunctive graph (Figure 5). The nodes of the graph correspond to operations - let us use node A_i to describe operation of job A running on machine i (for simplicity reasons we assume that each job visits each machine at most once). Moreover,

each node A_i has a weight p_{A_i} that is the processing time of the operation. There are two types of arcs in the disjunctive graph. The *conjunctive* (solid) arcs represent the precedence relations between operations of the same job. If there is a directed arc (A_i, A_k) then job A has to be processed on machine i before it is processed on machine k. Two operations belonging to different jobs that have to be processed on the same machine are connected by so-called *disjunctive* (broken) arcs that go in opposite directions. The disjunctive arcs form cliques in the graph; each clique corresponds to one resource. In addition, there are dummy nodes S and T with weights 0 describing the start and end of the schedule. S is connected by conjunctive arcs to the nodes representing the first operations of each job and there are conjunctive arcs from the nodes representing the last operations of jobs to node T.

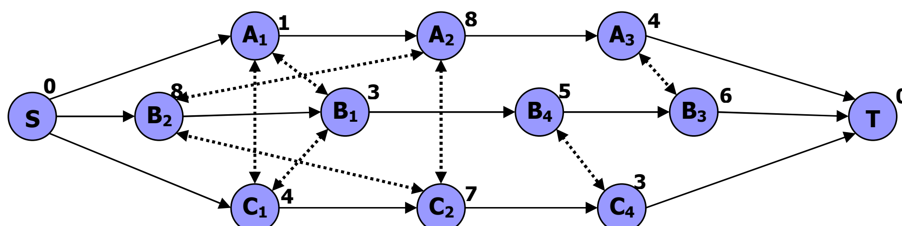


Figure 5 A disjunctive graph representing a job-shop scheduling problem.

The reader should realize that a feasible schedule corresponds to a selection of one arc from each pair of disjunctive arcs such that the resulting graph is acyclic. In particular, the selection of disjunctive arcs in each clique should be acyclic - it corresponds to the sequence in which the operations of particular jobs will be processed on a given resource. Moreover, the longest weighted path from S to T, where the length is measured as a sum of weights of visited nodes, determines the makespan C_{max}. This path is sometimes called a *critical path*. Naturally, the problem of minimizing the makespan reduces to finding a selection of disjunctive arcs that makes the graph acyclic and minimizes the length of the longest path.

We will present now one of the best heuristic algorithms to solve general job-shop scheduling problem - *shifting bottleneck heuristic* (Adams *et al.*, 1988). This algorithm first selects the resource that contributes most to the makespan - a so called *bottleneck resource*. Then it finds the best schedule for this resource, that is, a selection of disjunctive arcs from the corresponding clique, adds this selection to the partial schedule, and continues with another resource until all resources are scheduled. Moreover after scheduling each resource, the already scheduled resources are re-sequenced using the same principle - the removed disjunctive arcs are added back and a new selection for a given resource is found.

The problem of finding the bottleneck resource and generating its schedule can be formulated as the problem $1|r_j|L_{max}$ that we already considered. Assume that $P_{max}(k, n)$ is the longest path from k to n in the current disjunctive graph that includes conjunctive arcs and disjunctive arcs selected in previous iterations of the algorithm. For each not-yet scheduled resource i the problem $1|r_j|L_{max}$ consists of operations p_{A_i} allocated to this resource. The processing time of these operations is defined in the JSS problem, the release date $r_{A_i} = P_{max}(S, A_i)$, and the due date $\delta_{A_i} = P_{max}(S, T) - P_{max}(A_i, T) + p_{A_i}$. Notice that operation p_{A_i} cannot start before r_{A_i} because this is the earliest time when all preceding operations are finished. Also, if operation p_{A_i} finishes after δ_{A_i}, this produces a higher makespan. Hence the bottleneck resource is the resource which contributes to the highest increase of makespan, that is, the resource for which the problem $1|r_j|L_{max}$ solves to the largest lateness. The solution of $1|r_j|L_{max}$ defines which disjunctive arcs are selected for the resource.

This section described the basics of traditional scheduling technology. We introduced the scheduling terminology and gave some examples of scheduling algorithms. As we mentioned, there is a large number of different scheduling problems with different techniques to solve them. The reader is referred to the book (Brucker, 2001) which is a detailed catalogue of traditional scheduling problems with complexity analysis and references to best algorithms to solve particular problems. The book (Pinedo, 2002) also describes many scheduling techniques and it is appropriate for teaching purposes. Probably the most comprehensive coverage of the advanced topics in scheduling is (Leung, 2004).

4 Role of Constraint Satisfaction in Planning and Scheduling

Constraint satisfaction is a general technology for solving combinatorial optimization problems and hence it is not surprising that this technology has been applied to planning and scheduling problems as well. Constraint-based planning is a discipline that studies how to solve planning problems by constraint satisfaction, while constraint-based scheduling deals with applying constraint satisfaction techniques to scheduling problems.

Recall that constraint satisfaction is a technology originated in artificial intelligence where also planning problems are being solved, while scheduling is much closer to the area of operations research. Hence it may seem a bit surprising that constraint-based scheduling is a more mature area than constraint-based planning. Nevertheless, this disproportion can be easily explained by the character of the problem to be solved. Constraint satisfaction techniques typically assume the problem to be fully specified in advance by fixed sets of variables and constraints. As we shall see later, the scheduling problem with a known set of activities is static in the sense that the activities and hence the size of the solution are known in advance so the scheduling problem can be naturally mapped to a constraint satisfaction problem. On contrary, the planning problem is dynamic in the sense that the activities and hence the size of the solution are unknown in advance. So it is more complicated to map a full planning problem to a constraint satisfaction problem - usually a series of constraint satisfaction problems is necessary to solve the planning problem or constraints are used only to model some planning sub-problem such as temporal consistency.

The following sections discuss in more details how constraint satisfaction techniques are applied to planning and to scheduling.

4.1 Role of Constraint Satisfaction in Planning

One of the difficulties of planning is that the length of the plan, that is, the set of used actions, is unknown in advance, so some dynamic technique which can produce plans of ”unrestricted” length is required. Frequently, the shortest plan is being looked for, which is a form of *optimal planning*. As it has been shown in (Kautz & Selman, 1992), the problem of shortest-plan planning can be translated into a series of logical satisfiability (SAT) problems, where each SAT instance encodes the problem of finding a plan of a given length. First, we start with finding a plan of length 1, and, if it does not exist, then we continue with a plan of length 2, etc., until the plan is found. There exist criteria to stop these extensions if the plan does not exist (Ghallab *et al.*, 2004), but for simplicity reasons in this paper we assume that a plan always exists.

Now, the problem of finding a plan of length *n* can be encoded as a constraint satisfaction problem. The most important steps when designing a constraint model are the selections of variables, their domains, and finally constraints defining consistent tuples of the variables. We will present here the straightforward constraint model that has been described in (Ghallab *et al.*, 2004) and a more advanced model called CSP-PLAN (Lopez & Bacchus, 2003). For detailed comparison of these models and their further improvement, the interested reader is referred to (Barták & Toropila, 2008).

We assume sequential planning where the world state is described using *v* multi-valued state variables, the instantiation of which exactly specifies a particular state. A CSP denoting the problem of finding a plan of length *n* consists of *n+1* sets of above mentioned multi-valued variables, with the 1st set denoting the initial state and k^{th} set denoting the state after performing *k-1* actions, for $k \in \{2, \ldots, n+1\}$. We also need *n* action variables A^j, where *j* ranges from 0 to *n-1*, indicating the selected actions (Figure 6).

The two above mentioned constraint models (straightforward and CSP-PLAN) differ in the set of constraints used to describe state transitions.

In the straightforward model we use logical constraints that connect two adjacent sets of state variables through the corresponding action variable between them, that is, for given s we connect state variable layers V_i^s and V_i^{s+1}, $i \in \{0, \ldots, v-1\}$, through the action variable A^s:

$$A^s = act \rightarrow Pre(act)^s, \forall act \in Dom(A^s), \quad (1)$$

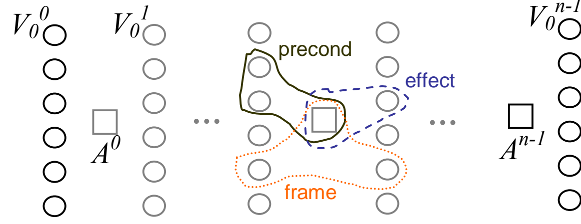


Figure 6 Base decision variables and constraints modeling plans of length n.

$$A^s = act \rightarrow Eff(act)^{s+1}, \forall act \in Dom(A^s), \tag{2}$$

where $Pre(act)^s$ and $Eff(act)^{s+1}$ are conjunctions of equalities setting the values for required state variables corresponding to preconditions of action *act* in layer s, and its effects in layer $s+1$ respectively. We also need constraints representing the *frame axioms*, that is, constraints that would enforce equalities between those state variables V_i^s and V_i^{s+1}, which are not affected by the selected action A^s (the frame assumption is implicit in classical planning representations):

$$A^s \in NonAffAct(V_i) \rightarrow V_i^s = V_i^{s+1}, \forall i \in < 0, v-1 >, \tag{3}$$

where $NonAffAct(V_i)$ is the set of actions that do not have state variable V_i among its effects. Please note that this set depends purely on actions' definition and thus can be pre-computed in advance.

The straightforward model uses separate constraints to describe preconditions and effects of actions and frame axioms specifying that the value of some state variable is not changed by the selected action. There is another more efficient way to describe how the state is changed by merging the constraints for effects and frame axioms into so called *successor state constraints* originally described in (Reiter, 2001). This method has been used in CSP-PLAN (Lopez & Bacchus, 2003) that was originally proposed as a constraint model for a so called planning graph (Blum & Furst, 1997). In the planning graph, several parallel actions can be used in each layer provided that these actions do not influence each other. We will present here a simplified version of the constraint model with a single action per layer. The encoding of action preconditions is again using the constraints of type (1). However, in contrary to the straightforward model we use the successor state axioms instead of effects and frame axioms. In particular, for each possible assignment of state variable $V_i^s = val$, $val \in Dom(V_i^s)$, we have a constraint between it and the same state variable assignment $V_i^{s-1} = val$ in the previous layer. The constraint says that state variable V_i^s takes value *val* if and only if some action assigned this value to the variable V_i^s, or equation $V_i^{s-1} = val$ held in the previous layer and no action changed the assignment of variable V_i. Formally:

$$V_i^s = val \leftrightarrow A^{s-1} \in C(i, val) \vee (V_i^{s-1} = val \wedge A^{s-1} \in N(i)), \tag{4}$$

where $C(i, val)$ denotes the set of actions containing $V_i = val$ among their effects, and $N(i)$ denotes the set $NonAffAct(V_i)$ as described within the previous model.

Notice that the straightforward model uses constraints in the form of implication which is basically an abbreviation for disjunctive constraints. Disjunctive constraints are known for weak propagation so the constraint models with disjunctive constraints do not exploit constraint propagation a lot and hence search is the prevailing technique for solving such problems. The equivalence constraint in CSP-PLAN leads to a stronger domain filtering and hence this model is more efficient.

Designing a constraint model is just the first step to solve the problem using constraint satisfaction technology. The next step is defining the search strategy, that is, the order in which the variables are instantiated and the order in which the values are tried for the variables. One can use generic search techniques for constraint satisfaction problems, for example based on fail-first principle, however, it is usually better to exploit the particular structure of the problem. First, one should realize that it is enough to instantiate just the action variables A^s because when their values are known then the values of remaining

variables, in particular the state variables, are set by means of constraint propagation. Of course, we assume that the values for state variables V_i^0 modeling the initial state were set and similarly the state variables V_i^n in the final layer were set according to the goal (the final state is just partially specified so some state variables in the final layer remain un-instantiated). The action variables can be instantiated in the increasing order A^0 to A^{n-1} to mimic the forward planning or in the decreasing order from A^{n-1} to A^0 to mimic regression (backward) planning. For value ordering (selection of action), one can use the planning heuristics designed for other planning algorithms (Ghallab *et al.*, 2004).

In this section we gave two examples of constraint models for finding a plan of a given length. These models are generated automatically from the description of the planning domain and the planning problem. There also exist handcrafted constraint models, for example CPlan (van Beek & Chen, 1999). All these models deal with sequential plans. Constraints are also used in partial order planners (the plan is a partially ordered structure of actions, it is a specific version of plan-space planning) such as CPT planner (Vidal & Geffner, 2004). Last but not least, constraint satisfaction techniques are frequently applied to planning sub-problems such as temporal constraint satisfaction problems (Dechter *et al.*, 1991) dealing with maintaining temporal relations between actions.

4.2 Role of Constraint Satisfaction in Scheduling

Scheduling is a ”killing application” for constraint satisfaction. The success of constraint-based scheduling in real-life applications is thanks to the fact that two research areas, namely operations research (OR) and artificial intelligence (AI), combined their complementary strengths in the constraint-based approach. As we saw in the previous sections, the traditional OR approach to scheduling focuses on exploiting the combinatorial nature of a relatively simple mathematical model of the scheduling problem. This leads to a high level of efficiency when solving such problems. The drawback of this approach is that, when mapping the real problem to the mathematical model, usually some degrees of freedom need to be discarded and simplifying assumptions need to be taken. Discarding degrees of freedom may eliminate some interesting solutions while discarding side constraints may lead to unacceptable solutions. In contrast, the AI approach traditionally focuses on general problem-solving techniques so all degrees of freedom and side constraints are preserved which may have the disadvantage of poor performance when comparing to dedicated solving algorithms.

Constraint satisfaction provides a very good framework for integrating OR techniques in more general AI solving algorithms. The key technology for this integration is based on the notion of *global constraints*. Global constraints encapsulate a certain part of the constraint satisfaction problem and, rather than using a set of constraints to model this sub-problem, a dedicated ”larger” global constraint is used that can exploit better the structure of the sub-problem. Global constraints together with sophisticated search techniques are the key power behind the success of constraint-based scheduling. Global constraints provide efficient algorithms to solve well-defined sub-problems while they can be still combined with other constraints modeling the side features of the problem. In this section we shall give some examples of scheduling global constraints. In particular, we shall describe how the structure of the sub-problem modeled by the global constraint can be exploited in the filtering algorithm.

Let us first describe the base constraint model of a scheduling problem. Recall that a scheduling problem is a decision problem where we are looking for when and where the jobs will be processed. Jobs typically consist of temporally connected operations that are the basic scheduling objects. For each operation, we can introduce three variables indicating its position in time, namely, the start time, the end time, and the processing time (duration). For operation A we denote these variables by *start(A), end(A),* and *p(A)*. We expect the domains for these variables to be discrete (for example natural numbers representing time) where the release date and the deadline of the operation make natural bounds for them. It is possible to further restrict the domain by assuming the time windows when the operation can be processed (time windows are a typical example of a real-life side constraint). Thought frequently the processing time of operation is a constant number and so the start time is enough to fully specify the allocation of operation to time, we prefer to use all three variables to simplify description of the constraints. Speaking about constraints, for operations without pre-emption, the following constraint

connects the above three variables: $start(A) + p(A) = end(A)$. The preemptive case is slightly more complicated, the reader may look at ((Baptiste *et al.*, 2006), p. 765) for details. If resource allocation is a part of the scheduling problem, then one more variable is required to describe which resource will process the operation: *resource(A)*. Its domain equals the set of numbers, where the numbers are uniquely assigned to resources. The domain contains just the numbers indicating resources that can process a given operation.

Basically, there are two groups of constraints involved in the model: temporal and resource constraints. Temporal constraints describe the direct temporal relations between the operations such as the precedence relations. The relation that operation A must be processed before operation B can be modeled using the constraint: $end(A) \leq start(B)$. In the remaining text, we will denote this constraint as $A << B$. It is easy to generalize this constraint to model a more detailed temporal relation with minimal and maximal delays between the operations. Then the constraint has a form $min_delay(A, B) \leq start(B) - end(A) \leq max_delay(A, B)$. When the temporal constraint network is sparse, as is the usual case in scheduling, then standard arc-B-consistency (Lhomme, 1993) is used to propagate these constraints. For more dense networks it is more appropriate to use path-consistency like algorithms.

Assume we have a unary (disjunctive) resource that can process at most one operation at any time. Two operations A and B processed on the unary resource cannot overlap in time, so either A precedes B or vice versa: $A << B \vee B << A$. The unary resource can be fully modeled by a set of such disjunctive constraints. Unfortunately, the disjunctive constraints do no propagate well as Figure 7 shows. Three operations are displayed there such that neither pair forbids any sequence (*A* can be before *B* as well as after *B*). As we shall see later, the time window of operation *A* can be significantly reduced if we consider all operations together.

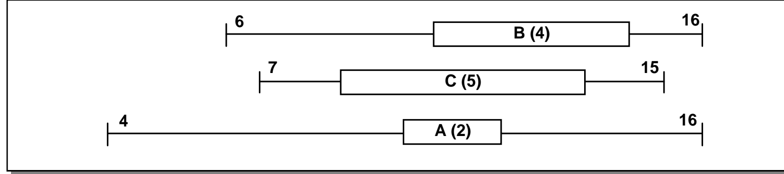


Figure 7 Example of three operations allocated to a unary resource where the disjunctive constraints deduce no pruning of time windows.

Let us assume hypothetically that operation *A* is not processed first. Then we must start either with *B* or *C* so the earliest start time is 6. If we now add all processing times of operations *A, B, C,* which is 11 (= 2+4+5), we get the earliest completion time 17 which is clearly after the latest completion time of all the activities (16). In summary, if we do not start with *A* then there will not be enough time to process all activities. The conclusion is that the sequence of operations must start with *A* which can be used to prune the time window of *A* to interval $[4, 7]$. This technique known as *edge-finding* can be generalized to a set of operations Ω in the following way:

$$min(start(\Omega)) + p(\Omega) + p(A) > max(end(\Omega \cup \{A\})) \Longrightarrow A << \Omega.$$

$A << \Omega$ means that *A* must be processed before any operation from Ω, in particular it must be processed before any $\Omega' \subseteq \Omega$. So we can restrict the completion time of *A* in the following way:

$$A << \Omega \Longrightarrow end(A) \leq min\{max(end(\Omega')) - p(\Omega')|\Omega' \subseteq \Omega\}.$$

A similar rule can be constructed to deduce that A must be processed after Ω:

$$min(start(\Omega \cup \{A\})) + p(\Omega) + p(A) > max(end(\Omega)) \Longrightarrow \Omega << A$$
$$\Omega << A \Longrightarrow start(A) \geq max\{min(start(\Omega')) + p(\Omega')|\Omega' \subseteq \Omega\}.$$

The above rules are encoded in the global constraint modeling the unary resource. There exist incremental algorithms with the time complexity $O(n^3)$ based on task intervals (Baptiste & Le Pape,

1996), algorithms with the time complexity $O(n^2)$, for example the algorithm by (Wolf, 2003) based on the sweep pruning technique, or even edge finding algorithms with the time complexity $O(n.log\ n)$ by (Carlier & Pinson, 1994) or (Vilím *et al.*, 2005), where *n* is the number of operations.

Edge-finding is not the only filtering technique for unary constraints. A complementary technique, called ”not-first/not-last”, deduces that an operation cannot be processed first or last (Torres & Lopez, 2000). (Baptiste & Le Pape, 1996) designed a not-first/not-last algorithm with the time complexity $O(n^2)$ and (Vilím, 2004) proposed a filtering algorithm with the time complexity $O(n.log\ n)$. Some of above mentioned techniques can be extended to cumulative resources, that is, discrete resources with capacity greater than one (more operations can be processed in parallel). For example, the paper (Baptiste *et al.*, 2006) shows a cumulative version of the edge-finding technique. Other techniques have been proposed particularly for cumulative resources, for example the energy precedence propagation (Laborie, 2003) combines information about precedence relations and limited capacity of the resource. As we already mentioned the big advantage of constraint models is their flexibility to combine constraints describing various aspects of the problem. So in addition to ”standard” features of the problem modeled typically by global constraints, the user may define any other restriction on decision variables.

From the section on constraint satisfaction it should be clear that the constraint model is not enough to solve the problem; the constraint model needs to be accompanied by the search procedure that instantiates the variables. It is possible to use any search strategy developed for constraint satisfaction problems; however, the dedicated search strategies for a class of problems frequently give better results. For example, instead of instantiating the decision variables in the model, the scheduling search strategies are usually based on different branching schemes. As scheduling is basically about finding a sequence of operations, the branching is typically based on deciding which operation is processed before another operation. In particular, if operations *A* and *B* are processed on the same resource, we can decide which one will be processed first by exploring two alternatives $A << B$ or $B << A$. The question is which pair of operations should be explored first. (Smith & Cheng, 1993) proposed a heuristic based on slack which describes the flexibility for allocating the operations. More formally, the slack for two operations *A* and *B* is defined as:

$$max\{max(end(A)) - min(start(B)), max(end(B)) - min(start(A))\} - p(\{A, B\}).$$

According to the fail-first principle (select the choice point where no branch leads to a solution with the highest probability), the pair of variables with the smallest slack should be ordered first. According to the succeed-first principle (select the branch that leads to the solution with the highest probability), the ordering with the larger slack should be tried first. This branching scheme requires $O(n^2)$ choices to be resolved during search because each pair among the *n* operations should be ordered. In (Baptiste *et al.*, 1995) a different branching scheme for operation selection is studied. Rather than deciding about the order of two not-yet ordered operations, we can decide about the first operation in the resource - we are resolving the disjunction $A << \Omega \vee \neg A << \Omega$. Again, we can use (resource) slack to find the bottleneck resource: $max(end(\Omega)) - min(start(\Omega)) - p(\Omega)$, where Ω is the set of operations allocated to the resource. The resource with the smallest slack is explored first and the operation with the earliest start time is allocated to the first position ($A << \Omega$). If this decision leads to a failure then the alternative decision ($\neg A << \Omega$) is used which says that A is not first in the resource. This branching strategy uses $O(n)$ choice points during search. Note finally that resource slack can also be used when deciding about the value of *resource(A)* variable. In this case, *A* is allocated first to the resource with the largest slack (succeed-first principle).

So far we focused on modeling the feasibility problem, that is, finding a schedule satisfying the resource and temporal restrictions. The optimization problem is usually solved by a branch-and-bound technique where the objective function *Obj* depending on variables *Vars* is represented as a new constraint *X = Obj(Vars)* with a new variable *X* that is being minimized (or maximized). The filtering procedure behind the constraint typically restricts the domain of variable *X* by computing the lower and bound of the objective function for partial instantiation of variables *Vars*. Again, the relaxation techniques from traditional scheduling that estimate the bounds of the objective function can be transformed into a filtering algorithm of the above mentioned ”objective” constraint.

This section gave some examples and techniques of using constraint satisfaction for solving scheduling problems. A more detailed survey can be found in (Barták, 2005) and (Baptiste *et al.*, 2006), where planning techniques are also covered. Probably the most comprehensive coverage of constraint-based scheduling is (Baptiste *et al.*, 2001) where the filtering algorithms behind the propagation rules for various types of resources are described.

5 Open Research Issues

5.1 Integration on Planning and Scheduling

As we already mentioned, planning and scheduling are closely related areas. Typically, to solve completely a real-life problem, both planning and scheduling components are necessary. The original planning techniques suppressed the role of limited time and resources and focused on causal relations. However, as the planning techniques become more mature and hence closer to real-life problems, the role of limited time and resources started to be important. Planning with time and resources is now a hot research topic where the scheduling techniques might be exploited. Similarly, so far the scheduling algorithms focused on time and resource allocation and the question how to obtain a set of activities was left apart. Typically, the set of activities is generated by a straightforward algorithm based on the description of environment, for example, using a description of processes how to manufacture a given part. If there were some options, for example alternative processes such as outsourcing, the decision was done by a human operator. However, as production environments are becoming more flexible and complicated, the number of options how to select the activities is increasing and hence the choice of the ”right” activities to be scheduled is becoming important. It means that some ideas from planning might be interesting to scheduling as well, for example in flexible manufacturing environments. Naturally, if complete artificial agents are assumed, such as robots, both planning and scheduling components need to closely co-operate. Hence not surprisingly, the research on integrating planning and scheduling is driven mainly by the planning community.

5.2 Knowledge vs. Software Engineering

Traditionally, software engineering plays an important role in the development of planning and scheduling software for real-life applications. The focus is on the software part of the system. In particular, software engineering methods are used to improve reliability and maintainability of the software which is definitely important both for the developers and for the users of the software. So far much less attention has been paid to the knowledge engineering part of planning and scheduling. Knowledge engineering deals with building, maintaining, and development of knowledge-based systems so its goals are close to software engineering. The main topics of knowledge engineering are how to acquire, represent, maintain and use knowledge.

Knowledge engineering is closely related to artificial intelligence so it is not surprising that it is more widely used in planning than in scheduling. As far as scheduling deals with ad-hoc algorithms for solving particular problems, the knowledge engineering part is less important. However, as soon as the scheduling software has to deal with a scheduling problem in general, without knowing explicitly to which category the scheduling problem belongs, the role of knowledge engineering becomes eminent. This is in particular the case of constraint-based scheduling which is typically oriented to general scheduling problems. For such systems it is important how to convert data describing parameters of the scheduling problem into an ”executable” model that can be solved (Barták *et al.*, 2007). Also, the automated extraction of knowledge about the problem is important for efficient problem solving. It could be automated construction of implied constraints (Barták, 2007), detection and breaking of symmetries (Puget, 2005) or selection of the most promising solving algorithm (Carchrae & Beck, 2005).

The role of knowledge engineering has already been recognized in the planning community where a competition on knowledge engineering techniques for planning and scheduling (Barták & McCluskey, 2007) is a forum for presenting KE (knowledge engineering) techniques. Knowledge engineering for AI Planning can be defined as the process that deals with the acquisition, validation and maintenance of planning domain models, and the selection and optimization of appropriate planning machinery to work

on them. The typical representatives of KE for planning are the tools for problem modeling, such as the GIPO system (McCluskey *et al.*, 2003), methods of problem reformulation, such as the methods used in Fast Downward system (Helmert, 2006), or the methods for learning control knowledge (Borrajo & Veloso, 1996).

5.3 Temporal Reasoning

Being able to reason about time is crucial in many real life problems, such as planning and scheduling. Several approaches have been proposed to reason about temporal information. *Temporal constraints* have been among the most successful in practice.

In temporal constraint problems, variables either represent instantaneous events, such has "when a plane takes off", or temporal intervals, such as "the duration of the flight". Temporal constraints allow one to put temporal restrictions either on when a given event should occur, e.g. "the plane must take off before 10am", or on how long a given activity should last, e.g. "the flight should not last more than two hours".

Several quantitative and qualitative constraint-based temporal formalisms have been proposed, stemming from pioneering works by Allen (Allen, 1983) and by Dechter, Meiri, and Pearl (Dechter *et al.*, 1991). A qualitative temporal constraint defines which temporal relations, e.g. *before , after, during*, are allowed between two temporal intervals representing two activities. For example, one could say "fueling must be completed before boarding the plane". The quantitative version of this constraint would instead be "the time difference between the end of the fueling task and the start of the boarding must be between 5 and 20 minutes". Disjunctions can be expressed in both formalisms, as in "I will call you either before or after the flight" or "I'll be flying home either between 2pm and 4pm or between 6pm and 8pm".

Once the constraints have been stated, the goal is to find an occurrence time, or duration, for all the events respecting all the constraints. In general, solving temporal constraint problems is difficult. However, there are tractable classes, such as quantitative temporal constraint problems where there is only one temporal interval for each constraint (Dechter *et al.*, 1991). These are problems that consists of a set of variables, and, for each pair of variables, say x and y, at most one constraint of the form $a \leq x - y \leq b$, where a and b are time instants.

5.4 Representing Preferences and Coping with Uncertainty

Preferences and constraints occur in real-life problems in many forms. As said above, constraints are restrictions on the possible scenarios. For a scenario to be feasible, all its constraints must be satisfied. For example, if we want to buy a PC, we may pose a lower limit on the size of its screen. Only PCs that respect this limit will be considered.

Preferences, on the other hand, express desires, satisfaction levels, rejection degrees, or costs. For example, we may prefer a tablet PC to a regular laptop, we may desire having a webcam, and we may want to spend as little as possible. In this case, all PCs will be considered, but some will be more preferred than others. Such concepts can be expressed in either a qualitative or a quantitative way.

Preferences and constraints are closely related notions, since preferences can be seen as a form of "tolerant" constraints. For this reason, there are several constraint-based frameworks to model preferences. One of the most general framework, based on *soft constraints* (Rossi *et al.*, 2006), extends the classical constraint formalism to model preferences in a *quantitative* way, by expressing several degrees of satisfaction (that can be either totally or partially ordered). When there are both levels of satisfaction and levels of rejection, preferences may be called *bipolar*, and can be modeled by extending the soft constraint formalism (Bistarelli *et al.*, 2006).

Preferences can also be modeled in a *qualitative* way (also called *ordinal*), that is, by pairwise comparisons. In this case, soft constraints (or their extensions) are not suitable. Other AI preference formalisms are however able to express preferences qualitatively, such as CP-nets (Boutilier *et al.*, 2004). Fortunately, CP-nets and soft constraints can be combined, providing a single environment where both qualitative and quantitative preferences can be modeled and handled.

Specific types of preferences come with their own reasoning engines. For example, *temporal* preferences are quantitative preferences that pertain to distances and durations of events in time. Soft constraints can be embedded naturally in a temporal constraint framework to handle such a kind of preferences (Khatib *et al.*, 2001; Peintner & Pollack, 2004).

While soft constraints generalize the classical constraint formalism providing a way to model several kinds of preferences, this added expressive power comes at a cost, both in the modeling task as well as in the solving process. To mitigate these drawbacks, various AI techniques have been adopted. For example, *abstraction theory* (Cousot & Cousot, 1977) has been exploited to simplify the process of finding a most preferred solution of a soft constraint problem (Bistarelli *et al.*, 2002). Also, *explanations* have been considered to ease the understanding of the result of the solving process (Freuder *et al.*, 2003).

On the modeling side, it may be tedious, or unreasonably demanding, for a user to specify all the soft constraints. In this respect, *machine learning techniques* have been used to learn the missing preferences from the known ones (Rossi & Sperduti, 1998; Vu & O'Sullivan, 2007). Alternatively, *preference elicitation* techniques (Chen & Pu, 2004), interleaved with search and propagation, have been exploited to minimize the user's effort in problem specification while still being able to find the most preferred solution (Gelain *et al.*, 2007).

Preferences can be seen as a way to model uncertainty in a constraint problem: if we don't have enough data to know whether a partial assignment of values to variables should be allowed or forbidden, we resort to preferences to say something less crisp. However, there are other kinds of uncertainty that can arise and can be modeled within a CSP. For example, there could be variables whose value cannot be decided by us. Such variables, usually called *uncontrollable*, represent events that are not under our control, but can be decided only by Nature or by some other agent.

In temporal constraint problems with preferences, much work has been done to understand how to best deal with uncontrollable events (Vidal & Fargier, 1999). In fact, three notions of *controllability*, that is, strong, weak, and dynamic, are usually considered in this setting. They correspond to more or less pessimistic approaches to deal with this kind of uncertainty.

5.5 Distributed agents

Distributed constraint satisfaction plays an important role in many real planning and scheduling problems where variables and/or constraints must be distributed among a set of agents. Indeed, many real planning and scheduling problems can be modeled as a distributed constraint satisfaction problem and solved using distributed CSP techniques. The development of distributed CSP techniques was pioneered by Yokoo et al. (Yokoo *et al.*, 1992) in their asynchronous backtracking algorithm.

If all knowledge about the problem can be gathered into one agent, this agent could solve the problem alone using traditional centralized constraint satisfaction algorithms. However, such a centralized solution is often inadequate or even impossible due to inner properties. Faltings and Yokoo (Faltings & Yokoo, 2005) present some reasons why distributed methods may be desirable:

- *Cost of creating a central authority:* A constraint satisfaction problem may be naturally distributed among a set of peer agents. In such cases, a central authority for solving the problem would require adding an additional element that was not present in the architecture. Examples of such systems are sensor networks, or meeting scheduling.
- *Knowledge transfer costs:* In many cases, constraints arise from complex decision processes that are internal to an agent and cannot be articulated to a central authority. Examples of this range from simple meeting scheduling, where each participant has complex preferences that are hard to articulate, to coordination decisions in virtual enterprisers that results from complex internal planning. A centralized solver would require such constraints to be completely articulated for all possible situations. This would entail prohibitive costs.
- *Privacy/Security concerns:* Agents constraints may be strategic information that should not be revealed to competitors, or even to a central authority. This situation often arises in e-commerce and virtual enterprisers. Privacy is easier to maintain in distributed solvers.

- *Robustness against failure:* The failure of the centralized server can be fatal. In a distributed method, a failure of one agent can be less critical and other agents might be able to find a solution without the failed agent. Such concerns arise for example in sensor networks, but also in web-based applications where participants may leave while a constraint solving process is ongoing.

Nevertheless, the more cited papers related to distributed CSP make the following assumptions for simplicity in describing the algorithms:

1. Each agent has exactly one variable.
2. All constraints are binary.
3. Each agent knows all constraint predicates relevant to its variable.

Although the great majority of real problems are naturally modeled as non-binary CSPs, the second assumption is comprehensible due to there exist some techniques that translate any non-binary CSP into a equivalent binary one (Bacchus & van Beek, 1998).

However, the first assumption in too restrictive and the main basic research is focused to small instances, so that the following question is straightforward: *Why it is assumed a Variable per Agent?* (Salido, 2007).

Only some works include a set of variables into an agent (Salido & Barber, 2006),(Ezzahir *et al.*, 2007). Therefore, more research must be done to solve more realistic problems.

Multi-agent planning and scheduling seems to fall in the intersection of the fields of planning and scheduling, distributed systems, parallel computing/algorithms, and multi-agent systems. However, much of the research appears to build on ideas from either planning or multi-agent systems (and usually not both). From the viewpoint of planning, planning for multiple agents means supporting concurrent actions, and planning by multiple agents means parallelizing a planning algorithm. One might argue that the former has been done and the latter should be solved using parallel computing techniques and is dependent on hardware. On the other hand, from a multi-agent systems perspective, multi-agent planning is not about just solving planning problems but also how agents should behave and interact given that they have plans or planning capabilities.

6 Acknowledgements

Roman Barták is supported by the Czech Science Foundation under the contract 201/07/0205.

Miguel A. Salido is supported by the research projects TIN2004-06354-C02-01 (Min. de Educacin y Ciencia, Spain-FEDER).

References

Adams, J., Balas, E., & Zawack, D. 1988. The shifting bottleneck procedure for job shop scheduling. *Management Science*, **34**, 391–401.

Allen, J. F. 1983. Maintaining knowledge about Temporal Intervals. *Communications of the ACM*, **26**(1), 832–843.

Apt, K.R. 2003. *Principles of Constraint Programming*. Cambridge University Press.

Bacchus, F., & van Beek, P. 1998. On the conversion between non-binary and binary constraint satisfaction problems. *In Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98),*, 311–318.

Baptiste, P., & Le Pape, C. 1996. Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group (PLANSIG).*

Baptiste, P., Le Pape, C., & Nuijten, W. 1995. Constraint-Based Optimization and Approximation for Job-Shop Scheduling. *Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI-95.*

Baptiste, P., Le Pape, C., & Nuijten, W. 2001. Constraint-based scheduling: applying constraint programming to scheduling. *Kluwer Academic Publishers*.

Baptiste, P., Laborie, P., Le Pape, C., & Nuijten, W. 2006. Constraint-Based Scheduling and Planning. *Handbook of Constraint Programming*, 761–799.

Barták, R. 1998. On-line Guide to Constraint Programming. *http://kti.mff.cuni.cz/∼bartak/constraints/index.html*.

Barták, R. 2005. Constraint Satisfaction for Planning and Scheduling. *In Vlahavas, Vrakas (eds.): Intelligent Techniques for Planning*, 320–353.

Barták, R. 2007. Generating Implied Boolean Constraints via Singleton Consistency. *In Abstraction, Reformulation, and Approximation (SARA 2007), LNAI 4612, Springer-Verlag,*, 50–64.

Barták, R., & McCluskey, L. 2007. Knowledge Engineering Tools and Techniques for Automated Planning and Scheduling Systems. *The knowledge engineering review - Special Issue*, **22(2)**.

Barták, R., & Toropila, D. 2008. Reformulating Constraint Models for Classical Planning. *Proceedings of the 21st International Florida AI Research Society Conference (FLAIRS 2008)*, 525–530.

Barták, R., Little, J., Manzano, O., & Sheahan, C. 2007. From Enterprise Models to Scheduling Models: Bridging the Gap. *In Miguel A. Salido, Juan Fdez-Olivares (Eds.) Planning, Scheduling and Constraint Satisfaction,*, 44–56.

Bistarelli, S. 2004. *Semirings for Soft Constraint Solving and Programming* . Spriger Verlag, LNCS 2962.

Bistarelli, S., Codognet, P., & Rossi, F. 2002. Abstracting soft constraints: Framework, properties, examples. *AI Journal*, **139**(2), 175–211.

Bistarelli, S., Pini, M.S., Rossi, F., & Venable, K.B. 2006. Bipolar Preference Problems: Framework, Properties and Solving Techniques. *Pages 78–92 of: CSCLP*. Lecture Notes in Computer Science, vol. 4651. Springer.

Bitner, J.R., & Reingold, E.M. 1975. Backtracking Programming Techniques. *Communications of the ACM 18*, 651–655.

Blum, A., & Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence*, **90**, 281–300.

Borrajo, D., & Veloso, M. 1996. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning*, 1–34.

Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H.H., & Poole, D. 2004. CP-nets: A Tool for Representing and Reasoning with Conditional Ceteris Paribus Preference Statements. *JAIR*, **21**, 135–191.

Brucker, P. 2001. *Scheduling algorithms*. Spriger Verlag.

Carchrae, T., & Beck, J.C. 2005. Applying Machine Learning to Low Knowledge Control of Optimization Algorithms. *Computational Intelligence*, **21**, 372–387.

Carlier, J., & Pinson, E. 1994. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, **78(2)**, 146–161.

Chen, L., & Pu, P. 2004. *Survey of Preference Elicitation Methods*. Tech. rept. IC/200467. Swiss Federal Institute of Technology in Lausanne (EPFL).

Cousot, P., & Cousot, R. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Pages 238–252 of: POPL.*

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Dechter, R., & Pearl, J. 1988. Network-Based Heuristics for Constraint Satisfaction Problems. *Artificial Intelligence*, **34**, 1–38.

Dechter, R., Meiri, I., & Pearl, J. 1991. Temporal Constraint Network. *Artificial Intelligence*, **49**, 61–95.

Ezzahir, R., Bessiere, C., Belaissaoui, M., & Bouyakhf, El-H. 2007. DisChoco: A platform for distributed constraint programming. *In Proceedings of IJCAI-07 Workshop on Distributed Constraint Reasoning*.

Faltings, B., & Yokoo, M. 2005. Introduction: Special Issue on Distributed Constraint Satisfaction. *Artificial Intelligence*, **161**, 1–5.

Fikes, R., & Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, **2(3-4)**, 189–208.

Freuder, E. 1982. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, **29**, 24–32.

Freuder, E.C., Likitvivatanavong, C., M., Manuela, Rossi, F., & Wallace, R.J. 2003. Computing Explanations and Implications in Preference-Based Configurators. *Pages 76–92 of: CSCLP*.

Frost, D., & Dechter, R. 1994. Dead-end driven learning. *In Proc. of the National Conference on Artificial Intelligence*, 294–300.

Gaschnig, J. 1977. *A general backtrack algorithm that eliminates most redundant tests.* In Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI).

Gaschnig, J. 1979. *Performance measurement and analysis of certain search algorithms*. Carnegie-Mellon University: Technical Report CMU-CS-79-124.

Gelain, M., Pini, M.S., Rossi, F., & Venable, K.B. 2007. Dealing with Incomplete Preferences in Soft Constraint Problems. *Pages 286–300 of: CP*. Lecture Notes in Computer Science, vol. 4741. Springer.

Ghallab, M., Nau, D., & Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Golomb, S.W, & Baumert, L.D. 1965. Backtrack Programming. *Communications of the ACM*, **12(4)**, 516–524.

Graham, R.L., Lawler, E.L., Lenstra, J.K., & Rinnooy-Kan, A.H.G. 1979. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, **5**, 287–326.

Haralick, R., & Elliot, G. 1980. Increasing Tree Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, **14**, 263–314.

Hatamlou, A.R., & Meybodi, M.R. 2007. A Hybrid Search Algorithm for Solving Constraint Satisfaction Problems. *In Proc. of World Academy of Science, Engineering and Techniology*, **25**, 362–364.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, **26**, 191–246.

Hoffmann, J., & Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, **14**, 253–302.

Jackson, J.R. 1955. *Scheduling a production line to minimize maximum tardiness*. Research report 43. Management Science Research Project, University of California.

Kautz, H., & Selman, B. 1992. Planning as satisfiability. *Proceedings of ECAI,*, 359–363.

Khatib, L., Morris, P.H., R.A., Morris, & Rossi, F. 2001. Temporal Constraint Reasoning With Preferences. *Pages 322–327 of: IJCAI*. Morgan Kaufmann.

Kumar, V. 1992. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, **13**, 32–44.

Laborie, P. 2003. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, **143**, 151–188.

Leung, J.Y.T. 2004. *Handbook of scheduling: algorithms, models, and performance analysis*. Chapman & Hall.

Lhomme, O. 1993. Consistency techniques for numeric CSPs. *Proceedings of 13th International Joint Conference on Artificial Intelligence*, 232–238.

Lopez, A., & Bacchus, F. 2003. Generalizing GraphPlan by Formulating Planning as a CSP. *Proceedings of IJCAI*, 954–960.

Marriott, K., & Stuckey, P.J. 1998. *Programming with constraints: an introduction*. MIT Press.

McCluskey, T.L., Liu, D., & Simpson, R.M. 2003. GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment. *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2003). AAAI Press.*, 90–101.

McGann, C., Py, F., Rajan, K., Ryan, J., & Henthorn, R. 2008. Adaptive Control for Autonomous Underwater Vehicles. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, 1319–1324.

Michalewicz, Z., & Fogel, D.B. 2000. *How to Solve It: Modern Heuristics*. Springer-Verlag.

Muscettola, N., Nayak, P., Pell, B., & Williams, B. 1998. Remote Agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, **103**, 5–47.

Nareyek, A., Freuder, E.C., Fourer, R., Giunchiglia, E., Goldman, R.P., Kautz, H., Rintanen, J., & Tate, A. 2005. Constraints and AI Planning. *IEEE Intelligent Systems*, **20(2)**, 62–72.

Peintner, B., & Pollack, M.E. 2004. Low-cost Addition of Preferences to DTPs and TCSPs. *Pages 723–728 of: Proc. of AAAI'-04*. AAAI Press / The MIT Press.

Penberthy, J., & Weld, D.S. 1992. UCPOP: A sound, complete, partial order planner for ADL. *In Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, 103–114.

Pinedo, M. 2002. *Scheduling: theory, algorithms, and systems*. Prentice Hall.

Prosser, P. 1993. Hybrid Algorithm for the Constraint Satisfaction Problem. *Computational Intelligence*, **9**, 268–299.

Puget, J.F. 2005. Automatic detection of variable and value symmetries. *Principles and Praktice of Constraint Programming (CP 2005). LNCS 3709, Springer Verlag*, 475–489.

Reiter, R. 2001. *Knowledge in Action: Logical Foundation for Specifying and Implementing Dynamic Systems*. MIT Press.

Rossi, F., & Sperduti, A. 1998. Learning solution preferences in constraint problems. *J. Exp. Theor. Artif. Intell.*, **10**(1), 103–116.

Rossi, F., Van Beek, P., & Walsh, T. 2006. *Handbook of constraint programming*. Elsevier.

Ruml, W., Do, M.B., & Fromherz, M. 2005. On-line planning and scheduling for high-speed manufacturing. *In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 30–39.

Ruttkay, Z. 1998. Constraint Satisfaction - a Survey. *CWI Quarterly*, **11**(2&3), 123–162.

Sacerdoti, E. 1990. The nonlinear nature of plans. *In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 206–214.

Sadeh, N., & Fox, M.S. 2003. Hybrid Solving for CSP. *ALP newsLetter*, **16**.

Salido, M.A. 2007. Distributed CSPs: Why It Is Assumed a Variable per Agent? *Abstraction, Reformulation, and Approximation, 7th International Symposium, SARA 2007*, 407–408.

Salido, M.A., & Barber, F. 2006. Distributed CSPs by Graph Partitioning. *Applied Mathematics and Computation*, **183**, 491–498.

Smith, S.F., & Cheng, Ch.-Ch. 1993. Slack-Based Heuristics For Constraint Satisfaction Scheduling. *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 139–144.

Tate, A. 1977. Generating project networks. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 888–893.

Torres, P., & Lopez, P. 2000. On Not-First/Not-Last conditions in disjunctive scheduling. *European Journal of Operational Research*, **127**, 332–343.

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

van Beek, P., & Chen, X. 1999. CPlan: A Constraint Programming Approach to Planning. *Proceedings of AAAI-99*, 585–590.

Van Hentenryck, P. 1989. *Constraint Satisfaction in Logic Programming*. MIT Press.

Vidal, T., & Fargier, H. 1999. Handling contigency in temporal constraint networks. *J. of Experimental and Theoretical Artificial Intelligence Research*, **11**(1), 23–45.

Vidal, V., & Geffner, H. 2004. Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming. *Proceedings of AAAI-04*, 570–577.

Vilím, P. 2004. O(n log n) Filtering Algorithms for Unary Resource Constraint. *Proceedings of CP-AI-OR*.

Vilím, P., Barták, R., & Cepek, O. 2005. Extension of O(n log n) filtering algorithms for the unary resource constraint to optional activities. *Constraints*, **10(4)**, 403–425.

Vu, X.H., & O'Sullivan, B. 2007. Semiring-Based Constraint Acquisition. *Pages 251–258 of: Proc. ICTAI 2007*. IEEE Computer Society.

Waltz, D.L. 1972. *Generating Semantic description from drawings of scenes with shadows*. Technical Report AI-TR-271, MIT, Cambridge.

Waltz, D.L. 1975. Understanding line drawings of scenes with shadows. *The Psychology of Computer Vision*, 19–91.

Wolf, A. 2003. Pruning while Sweeping over Task Intervals. *Principles and Practice of Constraint Programming (CP 2003)*, 739–753.

Yokoo, M., Durfee, E.H., Ishida, T., & Kuwabara, K. 1992. Distributed constraint satisfaction for formalizing distributed problem solving. *In 12th International Conference on Distributed Computing Systems (ICDCS-92)*, 614–621.