Modeling and Optimization of Embedded System with Constraint Programming: Principles and Practice

Krzysztof Kuchcinski

Lund, January 7, 2019

Contents

One

Introduction

A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.

Antoine De Saint-Exupéry

1.1 Embedded Systems

Embedded system is usually considered as a device that includes a programmable computer but is not itself a general-purpose computer [91, 98]. This means that it is built for a specific application area in mind. There exist many different examples of embedded systems that we meet and use everyday. We use cell phones, digital cameras, camcoders, MP3 players, satellite navigation systems, life-support systems as well as washing machines. The common characteristics of these computer systems is that they are not user-programmable as general purpose computers. They, however, contain microprocessors, DSP processors and often specialized hardware. Currently a common configuration for embedded systems is the system on a chip (SoC) for which we put on a single chip a processor or processors and DSPs, specialized hardware blocks and interconnection structure (for example network on a chip (NoC)).

Embedded systems are reactive real-time systems. They react to external environment and maintain this interaction permanently. They ideally never terminate. Their real-time requirements mean also that these systems are subject to external timing constraints. These systems often implement sophisticated functionality. However, in addition to typical correctness requirements which are the same as for all computer programs or systems they need to provide correct timing behavior for real-time operations.

Many embedded systems are produced in the tens of thousands to millions of units range. Cell phones, for example, are produced in extremely big series. This creates a new requirement which is low manufacturing cost. Designers must be very careful in their design decisions to select solutions that provide required functionality but reduce cost. Another important constraint for many embedded systems is power or energy consumption of a final product. This is justified by the fact that many embedded systems are battery driven and their operation time is an important parameter for customers.

Designer of embedded systems must consider carefully different resources and select these which offer best trade-off. For example, it is important to select the processor which provides enough performance but it is not too expensive and its power consumption is acceptable. Right division of functionality between hardware and software is another example of such trade-offs. Sometimes we use term "resource conscious" design to underline importance of considering resources and different trade-offs during design.

They are different types of embedded systems that require different design methods. We usually distinguish between data and control dominated embedded systems. This classification is not always obvious but it helps to develop different modeling, analysis and synthesis methods. *Data dominated* systems process data arriving in regular streams (samples). The value arriving is most important and is used in the computations. Media applications, such as audio and video processing are examples of data dominated systems. *Control dominated* systems, on the other hand, do not know when data arrive but time of arrival often matters more than the value. Typical control systems, such as elevator control system, different kinds of alarm systems are included in this category.

Data dominated systems are typically modeled by some kind of data-flow graphs while control dominated systems use different types of state based formalisms (e.g., finite state machines, Petri nets, StateCharts). Different design problems arise for both types of systems. For control dominated systems, a designer is interested in response time and therefore schedulability analysis need to provide an answer, for example. For data dominated systems an important question is whether a system is able to provide correct data in a given time. Obviously, a schedulability analysis or static schedule can answer this question. Other questions, such as cost of a design, its power consumption and memory requirements might also be of interest.

Since designers always need to make trade-offs between different system parameters, such as performance, cost, timing and power there exist need for optimization of these parameters under heterogeneous constraints. For example, for an MPEG encoding system the rate of incoming images is known in advance, for example 25 frames/s. Therefore a designer need to find a cheapest solution which can provide this encoding rate. This solution can be a mixture of software and hardware applications. In addition, he might to minimize power consumption for his solution.

1.2 Optimization

Electronic design automation has developed many methods to solve combinatorial optimization problems found in this area [35]. While lower levels of design abstraction, such as physical level, logic level and to some degree register-transfer level (RTL) are quite well understood system level is still not easy to grasp with a single method or formalism and provide efficient algorithms. There exist many methods for different optimization problems. Systems are usually modeled as some kind of graphs, Petri nets or similar formalisms. For optimization one can either use exact methods that provide optimal solutions or heuristics. Mixed Integer Linear Programming (MILP) and Dynamic Programming are well known methods that can be used for optimization and can potentially provide optimal solutions. There exist also a set of specific heuristic methods or (meta-)heuristic methods for optimization.

Domain specific heuristic approaches provide specialized and efficient algorithms but they are usually limited to handle different constraints. They are also restricted to a particular problem and it is difficult to extend them to a slightly modified problem.

Local search methods, such as *simulated annealing* and *tabu search* [45, 37], provide a general frame work for optimization but a programmer has to develop domain specific methods for these algorithms. For example, a *move* in simulated annealing has to be defined very carefully to obtain good performance and good quality results [29]. Similar considerations has to be taken into account for genetic algorithms or evolution programs [59] when defining problem encoding and genetic operators.

MILP approaches, on the other hand, are quite flexible in modeling power but they usually suffer from long execution times and explosion of 0/1 decision variables and related inequalities. They are also limited to linear constraints and therefore linearization of the model often needs to be performed first. It is also difficult to extend MILP solvers with new search methods, which are based on domain specific knowledge. MILP solvers usually use a simplex method first to find a solution to a relaxed LP problem and then a branch-and-bound algorithm or similar to find a final solution. Advanced MILP solvers extend this classical method by using more advanced techniques, such as cutting plane method and improve their performance.

Dynamic programming is a technique that systematically constructs the optimal solution of a problem instance by defining the optimal solution in terms of optimal solutions of smaller size instances. The fact that the optimal solution can be constructed using subproblems and does not need to be recomputed is essential to dynamic programming. This idea is called the principle of optimality. There are successful examples of application of dynamic programming to optimization problems found in design automation (see for example [21]) but general application of this method to many problems in the area is not obvious [35].

In this book, we propose to use constraint programming (CP) methods to model and solve embedded system design problems. We do not argue that this is the only possible method but we try to indicate its advantages. CP can also be combined with other approaches. For example, LP can be used to find lower bounds for relaxed problems in the same way as it is used for MILP. Heuristics can provide hints on how to build efficient search methods. Finally, constraints can also be solved using local search methods.

CP is an exact method that can provide optimal solution together with optimality prove. It can also be used with heuristic search methods. In such cases, the execution time can be improved, rather good quality results can be obtained but obviously the optimality cannot be guaranteed. CP uses primitive arithmetic constraints, similar to MILP constraints, but it can also use more advanced constraints. A big advantage of CP is that, in contrast to MILP, it can handle non-linear constraints. It can also handle, so called combinatorial constraints, i.e., constraints that encapsulate specific combinatorial algorithms for particular combinatorial problems (see section 3.4, for example). An important feature of CP is its ability to define search heuristics that can use domain specific knowledge for improving search. In contrast to MILP the problem structure is not lost and one can basically use knowledge existing in the area on specific optimization heuristics. Another big advantage of CP is its use of combinatorial constraints and ability to combine different constraints and their reasoning strength in one problem definition.

1.3 Constraint Programming over Finite Domain

Constraint (logic) programming C(L)P is the study of computational systems based on constraints [5]. The basic idea is to solve problems by exploring constraints which must be satisfied by the solution. Therefore we do not concentrate on particular algorithms but on constraints which specify our problem. The specialized constraint programming systems are defined for different application domains. There exist systems for real or rational constraints (usually abbreviated as CLP(R) and CLP(Q) respectively) [83, 23], boolean constraints (abbreviated as CLP(B)), interval constraints (abbreviated as CLP(I)) [66, 94, 71] and finite domains constraints (abbreviated as CLP(FD)) [83, 80, 23, 26]. This book concentrates on finite domain constraints since they can be used to specify and solve combinatorial optimization problems.

Constraints have several interesting properties that can be summarized as follows [5].

- constraints may specify partial information, i.e., constraint need not uniquely specify the values of its variables,
- constraints are non-directional, typically a constraint on (say) two variables X, Y can be used to infer a constraint on X given a constraint on Y and vice versa,
- constraints are declarative, i.e., they specify what relationship must hold without specifying a computational procedure to enforce that relationship,
- constraints are additive, i.e., the order of imposition of constraints does not matter, all that matters at the end is that the conjunction of constraints is in effect,
- constraints are rarely independent, typically constraints in the constraint store share variables.

We model embedded systems by a set of constraints over variables. Each variable contains several integer values in its domain and therefore it is called *finite domain variable* (FDV). The FDV eventually can obtain an integer value that specifies a solution. The constraints are given as arithmetic expressions, equalities, inequalities and specialized combinatorial constraints.[1] They define specific application constraints (e.g., operation precedence constraints), resource constraints, specialized implementation dependent constraints (e.g., pipelining) as well as general requirements such as performance or cost. The constraint solving techniques are then used to find different solutions, optimal or suboptimal ones, which satisfy given constraints and optimize a given cost function. In this book, we will use optimization algorithms based on branch-and-bound method with different search methods (complete and heuristics) to find solutions to the imposed constraints.

All imposed constraints are stored in a so-called *constraint store*. Constraints in this store propagate information as soon as some domain variables become instantiated or their domains are narrowed. The propagation removes values from the domains of the remaining domain variables. The information about values left in the domains of variables is always available since a domain is represented explicitly in *CP* systems. The propagation detects inconsistency in many situations as well as reduces the search space significantly which makes the search in the design space more efficient.

The approach, based on CP techniques, for embedded system modeling and optimization has several advantages that can be summarized as follows.

[1]The combinatorial constraints are also called global constraints [23] or complex constraints [80]

- Constraints can be used to formalize both a system model and non-functional requirements of different types. The final set of constraints specifies uniformly all requirements on an implementation.
- A set of constraints can be easily extended with new constraints to accommodate new requirements on a design without changing related synthesis algorithms.
- Different constraint solving techniques, heuristics and complete optimization methods, can be used to solve the constraint model of the system.
- The use of constraints improves the general quality of a design process by providing a way to control all constraints in the same environment and assuring fulfillment of all constraints.
- It provides a unifying framework for different constraint consistency algorithms which, by narrowing domains of finite domain variables, contribute to a solution satisfying all constraints.

1.4 A Simple Example

In this book, we will not use Prolog, as a framework for CLP. We will use a pseudo-code which roughly corresponds to Java language. This pseudo-code can be directly converted to programs in Java with our prototype constraint solver JaCoP (**Ja**va **Co**nstraint **P**rogramming) [51]. This constraint solver has been implemented in Java and it has currently ∼20,000 lines of code.

JaCoP solver

In this section we will use a graph coloring problem to illustrate how a simple constraint program is built. Graph coloring is often used in to solve different design problems, for example it can be used for solving register allocation. Consider coloring of a graph depicted in Fig. 1.1. The following pseudo-code defines constraints for proper coloring of this graph together with a search for a single solution.

```
for (int i=0; i<4; i++)
  $v_i$ :: {1..4};
impose $v_0 \neq v_1$;
impose $v_0 \neq v_2$;
impose $v_1 \neq v_2$;
impose $v_1 \neq v_3$;
impose $v_2 \neq v_3$;
searchOne(v, search(), indomainMin(), delete(inputOrder));
```

The constraints impose inequality relation for FDVs representing two adjacent vertices. For example, vertices v_0 and v_1 need to have different colors and therefore the constraint $v_0 \neq v_1$ is imposed. After imposing all constraint we perform search to find a solution which satisfies all constraints. The search method requires four parameters. First, parameter is a vector $[v_0, v_1, v_2, v_3]$ of FDVs. These FDVs will get values assigned when the search is completed. The second parameter specifies the search methods. In this example, we use chronological backtracking search (see section 5.1.1). Third parameter specifies how values are assigned to FDVs. In this example, they are assigned from minimal to maximal value, i.e., we first try to assign smallest value in the domain of FDV. Finally, the last parameter specifies how we select FDVs for assignment. In this example we select them based on their position in the vector, i.e., first v_0 then v_1, and so on.

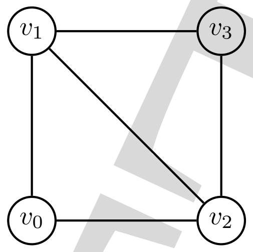


Figure 1.1: An example graph.

Note that we use a special keyword **impose** to denote declaration of a constraint and its registration in the constraint store. The corresponding detail code in Java with the methods provided by JaCoP library is presented in appendix A.1.

Our constraint program produces the following output indicating that vertices v_0, v_1 and v_2 get different colors (1, 2 and 3) while vertex v_3 gets color 1.

```
Solution: [v0=1, v1=2, v2=3, v3=1]
*** Yes
```

1.5 Complexity Considerations

Computational complexity considerations can be applied to different parts of constraint programming. First, computational complexity can be considered in relation to a particular problem and a specific algorithm for solving it. For example, a graph coloring problem and a specific search method for solving this problem, either complete or heuristic. Complexity analysis can also be applied to different algorithms that are used to build the solver. In this section, we will shortly discuss these two parts of the computational complexity of CP.

Finite domain constraint programming offers methods to solve problems modeled by constraints over finite domain variables. It can be used to find a *feasible solution* or to find an *optimal solution* to the problem defined by constraints. The feasible solution is any valid solution and optimal solution is a feasible solution that minimizes or maximizes a given cost function. As we have seen in the example discussed in the previous section a feasible solutions cannot usually be obtained directly using constraint processing methods and search has to be used to find such solution. Optimization also uses search and, by adding additional constraints on cost, finds an optimal solution. Basically it applies consecutive search methods with additional inequality constraints on cost (see section 5.3).

Many decision problems considered by CP approach are NP-complete problems. These problems require, in general, the exponential time complexity algorithm to find a solution. However, if the solution is known its correctness can be verified in polynomial time. Optimization version of the decision problem that is NP-complete is NP-hard problem. An algorithm for solving these problems has also exponential time complexity. However, verification that the solution is correct and optimal requires the exponential time complexity algorithm as well. In practice, even if a solver finds an optimal solution it may never verify its optimality due to the complexity of this problem.

For example, our graph coloring problem discussed in the previous section is NP-complete problem. Formally, the problem whether graph G is K-colorable for $K \geq 3$ is NP-complete [32]. Finding the minimal number of colors for coloring a given graph is NP-hard problem. Many other problems considered in this book and by CP approach are also NP-complete or NP-hard. Therefore one cannot expect that CP can, in general, solve all these problems efficiently. It can provide a framework for formalization of these problems and development of both heuristic and meta-heuristic algorithms that can be used to find acceptable solutions.

Finite domain constraint programming uses different methods to build a solver. Constraints are implemented using *consistency algorithms* that prune domains of variables. This provides often guidelines for controlling search and making efficient heuristic decisions during search. These consistency algorithms can have different pruning strength and computational complexity. We will discuss different algorithms and their complexity and will try to show trade-offs between consistency algorithms and search efficiency.

Two

Basics of Constraint Programming

God made the integers, all else is the work of man.

Leopold Kronecker (1823-1891),
Jahresberichte der Deutschen
Mathematiker Vereinigung.

In this chapter, we first introduce a *constraints satisfaction problem* (CSP) for finite domain constraints. Based on this theoretical formulation we will later define a framework which is useful in practice.

2.1 Constraint Satisfaction Problem

CSP is a 3-tuple $\mathcal{S} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where

- $\mathcal{V}$ $= \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, also called finite domain variables (FDVs) ,
- $\mathcal{D}$ $= \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$ is a finite set of domains, and
- $\mathcal{C}$ is a set of constraints restricting the values that the variables can simultaneously take.

For each *variable* x_i, a finite set $\mathcal{D}_i \in \mathcal{P}(\mathbb{Z}) \setminus \emptyset$ of possible values constitutes its domain, called a finite domain (FD). In this book, we limit a finite set of values to a finite set of integers. This assumption is usually assumed for FD constraint solvers since it does not limit expressiveness but unifies underlying solver procedures. An empty domain is excluded since it indicate inconsistency and is not a valid domain. For example, the specification $x :: \{1..10\}$ defines FDV x, which can have values 1, 2, …, and 10 while the specification $y :: \{23, 56\}$ defines FDV y, which can have a value of either 23 or 56.

A *constraint* $c(x_1, x_2, \ldots, x_n) \in \mathcal{C}$ between variables of $\mathcal{V}$ is a subset of the Cartesian product $\mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_n$ that specifies which values of the variables are compatible with each other.

Example 2.1: Assume that we define a constraints $x < y$, i.e., x must be lower than y. We assume also that $x :: \{0..2\}$ and $y :: \{0..2\}$. The constraint is defined by the following 2-tuples of $\langle x, y \rangle$ values.

$\{\langle 0,1\rangle, \langle 0,2\rangle, \langle 1,2\rangle\}$

In example 2.1 we have defined explicitly all values of x and y that are compatible with each other. The enumeration of allowed values indicates, for example, that x will never get value 2 and y will never be assigned value 0. This can be used to make *pruning* of the domain of x and y. The new domain of x is $\{0..1\}$ and the new domain of y is $\{1..2\}$. The domain pruning is achieved by executing *consistency* procedure. Consistency procedures used in constraint programming are discussed in section 2.4.

A *solution* s to a CSP $\mathcal{S}$, denoted by $\mathcal{S} \models s$, is an assignment to all variables $\mathcal{V}$, such that it satisfies all the constraints. This assignment is often called *label* or *compound label* [90] and process of finding a label is called *labeling*. We may want to find a *single solution*, *all solutions* or an *optimal solution*. In example 2.1, all solutions are explicitly specified and each pair (x, y) specifies a label. A single solution can be picked up by selecting any 2-tuple from specified tuples.

An optimal solution s to a CSP $\mathcal{S}$ is a solution ($\mathcal{S} \models s$) which minimizes or maximizes a value v assigned to a selected variable x_i. To find an optimal solution we need to define a cost function. If we assume that x is the cost function for our example we have two solutions $\{\langle 0,1\rangle, \langle 0,2\rangle\}$ that give a minimal value for this cost function. Picking up one of them gives us the minimal solution for the problem defined by constraint $x < y$. In most design problems, we are interested in optimal solutions that minimize or maximize this cost function.

Modeling of combinatorial problems involves many different constraints which share FDVs. The shared variables help to propagate information between constraints. Consider an extension to our simple problem defined in example 2.1.

Example 2.2: Assume that we add an additional constraint to example 2.1 specifying that $x > 0$. It is defined by the the tuple $\{\langle 1\rangle, \langle 2\rangle\}$. When putting constraints $x < y$ and $x > 0$ together we get 2-tuples for the constraint $x < y$ pruned to $\{\langle 1,2\rangle\}$. The domains of our variables become $x = 1$ and $y = 2$. In this case, these values are the solution for the imposed constraints.

The enumeration of all possible combinations of values which are compatible with each other is difficult but some solvers support this method of specifying constraints. GNU Prolog [26] offers two constraints, `fd_relation` and `fd_relationc`, that can define which are allowed values for a given tuple of FDVs. The relation is specified as a a list of tuples of integers.

Constraint programming environments makes it possible to specify constraints using equations, inequalities, combinatorial constraints, or programs defining compatible values. All these methods define, in fact, restrictions on the values which can be assigned to the constraint variables simultaneously. For example, an inequality $t_1 + d_1 \leq t_2$ defines a constraint on three FDVs t_1, d_1 and t_2.

Constraints are called *unary* if they have a single FDV, *binary* if they have two FDVs, and *n*-ary if they have *n* FDVs. An important class of CSPs are *binary constraint problems*, i.e., problems with unary and binary constraints only. A lot of research has focused on such problems and many techniques for these problems have been developed [90]. It is also interesting to note that all CSPs can be transformed to binary constraint problems [90]. Whether this kind of transformation will provide any benefit for a given problem is another question. To present this transformation assume that there is a constraint on k FDVs ($k > 2$) with variables $x_1, x_2, \ldots, x_n$. Basically, the transformation introduces a new FDV v and replaces the constraint on

k variables by k binary constraints on v and x_i. The new variable v is called *encapsulated variable* since its domain captures a Cartesian product of the domains of individual variables. Each of the newly created binary constraints connects v and one of the k FDVs. The example below illustrates this method.

Example 2.3: Assume that we would like to binearize the constraint $x + y = z$ with $x :: \{1..2\}, y :: \{1..2\}, z :: \{2..3\}$. This constraint defined by the following 3-tuples of $\langle x, y, z\rangle$ values.
$\{\langle 1,1,2\rangle, \langle 1,2,3\rangle, \langle 2,1,3\rangle\}$

The binearization introduces a new variable v with domain $\{112, 123, 213\}$. We have transformed 3-tuples specifying the constraint to a single number to simplify this example. Therefore a value $\langle 1,1,2\rangle$ is replace by 112, for example. The three new constraints introduced here for $(v, x), (v, y)$ and (v, z) are defined by the following 2-tuples definitions

for (v, x)	$\{\langle 112, 1\rangle, \langle 123, 1\rangle, \langle 213, 2\rangle\}$
for (v, y)	$\{\langle 112, 1\rangle, \langle 123, 2\rangle, \langle 213, 1\rangle\}$
for (v, z)	$\{\langle 112, 2\rangle, \langle 123, 3\rangle, \langle 213, 3\rangle\}$

The transformation of a general CSP into a binary CSP looses structural information about the original constraints. The original constraints have usually specialized techniques to prune the search space and therefore are more efficient. Therefore this theoretical opportunity has usually very little or no application in practice.

2.2 Constraint Graph

Traditionally, binary constraint problems can be defined using a graph representation. Since this representation plays an important role in CSP research we will first review it here. Later an extension to n-ary constraint problems with some practical considerations will be discussed.

A binary constraint satisfaction problem can be depicted by a *constraint graph* (sometimes referred also as a constraint network). Each node in this graph represents a finite domain variable, and each arc represents a constraint between these variables, represented by the end points of the arc. Unary constraints have arcs originating and terminating at the same node (called self connecting arcs).

Example 2.4: Consider example 2.2 which defines two constraints $x < y$ and $x > 0$, i.e., one binary constraint and one unary constraint. Its constraint graph is depicted in Figure 2.1. It has are two nodes representing FDVs x and y and two arcs. The arc connecting nodes x and y represents constraint $x < y$ and self connecting arc for node x depicts constraint $x > 0$.

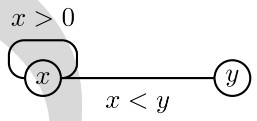


Figure 2.1: Constraint graph for example 2.4.

A natural extension of constraint graphs are *constraint hypergraphs*. A constraint hypergraph for a CSP is the hypergraph with nodes representing FDVs and hyperedges correspond to constraints of n variables. Constraint hypergrahs can represent general CSP problems but just as general CSPs can be reduced to binary CSP, constraint hypergraphs can also be reduced to constraint graphs.

In the next section we will discuss constraint satisfaction that is a basic property of constraints. Then we will extend our discussion of constraint graphs and show how they can be used to define different techniques for constraint handling. We will define constraints consistency techniques based on constraint graphs and discuss in detail node and arc consistency as well as path consistency. Finally, bounds consistency method which is used very much in practice will be presented.

2.3 Constraint Entailment

The most basic question one can ask about a constraint is whether it is or it is not *entailed* or also called *satisfied*. This question for finite domain constraints cannot always be answered. The solver is not complete and sometimes the positive or negative answer is not possible. In such cases we need to postpone the answer and therefore we say that the constraint's satisfiability is *unknown* indicating that the constraint may or may not be satisfied.

In practice, each constraint can be in one of three states: *satisfied*, *not satisfied* or in a state that cannot yet determine whether the constraint is satisfied or not (*'unknown'* state). If the constraint is in the 'unknow' state a consistency algorithm for this constraint can be applied. Different consistency algorithms and examples of propagation rules implementing bounds consistency for primitive constraints can be found in section 2.4.

Example 2.5: Consider constraint $x < y$ with different domains. Depending on the domain, the constraint can be satisfied, not satisfied or its satisfiability is unknown as indicated below.

$x :: \{1..5\}, y :: \{6..10\}$	$x < y$ satisfied
$x :: \{6..10\}, y :: \{1..5\}$	$x < y$ not satisfied
$x :: \{1..10\}, y :: \{1..10\}$	$x < y$ unknow
	after application of propagation rules
	$x :: \{1..9\}, y :: \{2..10\}$ (still unknown)

For the purpose of this book we introduce procedure `satisfied(constraint)` which returns `true` if `constraint` is satisfied and `false` if `constraint` is either not satisfied or its satisfiability is unknown. This procedure can be used by the solver to remove constraints that are satisfied and do not need to be considered for consistency checking. Note that not satisfied constraints will cause fail of their consistency procedures since they will generate an empty domain for one of FDVs involved in the considered constraints (consistency procedures are discussed in section 2.4).

2.4 Consistency Techniques

Consistency techniques are basically used to remove these values from the domain of FDV which are incompatible with values of domains of other FDVs in a given constraint. This prunes the domains of FDVs and possibly reduces search efforts. It can also discover earlier inconsistency in a given set of constraints.

Consistency methods are implemented using special methods that are usually called *propagation rules* or *propagators*. The propagation rules implement in fact constraints. In the description below we use p to denote a function that returns a new domain of a variable after applying a propagator. The propagators must have the following properties:

- *correct*– no solution of a constraint is removed,
- *assignment complete*– failure of a constraint is signaled at latest for the final assignments of values to FDVs,
- *contracting*– domains of variables do not become larger after applying a given propagator (in practice we would like them to be narrowed), i.e., $p(D_x) \subseteq D_x$,
- *monotonic*– propagator application to smaller domains will result in smaller domains than application to larger domains, i.e., $D_x \subseteq D_y \rightarrow p(D_x) \subseteq p(D_y)$.

The propagator may also be

- *idempotent*– always computes the fixpoint, i.e., several applications of the same propagator produces always the same result, i.e., $p(p(D_x) = p(D_x)$.

An implementation of a propagator has to, at least, fulfill the first four properties defined above. These properties leave, however, space for different implementations that influence solver performance. The easiest and very inefficient implementation can be build using the following strategy. The propagator never removes values from the domains but checks the constraint for failure when all FDVs have their final assignment. In the case when it detects inconsistency it signals fail for the solver. Such a propagator fulfills all four rules but is inefficient since often one can discover inconsistencies earlier or one can determine values that are inconsistent and can be removed. This is why many advanced consistency methods have been proposed and implemented in FD solvers. Below we will discuss different consistency methods which can be used to build realistic propagators for constraints. In subsection 2.4.3 we will concentrate on bounds and domain consistency methods that are usually used to build efficient solvers.

2.4.1 Node and Arc Consistency

Node and arc consistencies has been defined for constraint graphs. They are applied to constraints assigned to nodes (or more precisely to self connecting arcs) and arcs of this graph which explains their name.

The *node consistency* is applied to unary constraints and removes values from the domain of FDV that are incompatible. The procedure is defined in Figure 2.2. It deletes values v from the FDV domain D_N if it is inconsistent. Note that the procedure needs to be run once for all unary constraints to make them consistent. The constraint can then be removed. This corresponds to removal of all self connecting arcs in the constraint graph. One can therefore consider constraints graphs with binary constraints only.

Example 2.6: Consider the constraint graph depicted in Figure 2.1 with FDVs $x :: \{0..10\}$ and $y :: \{0..10\}$. The node consistency algorithm can be applied

```
void NodeConsistency()
  for each $N \in nodes(G)$ do
    for each $v \in D_N$ do
      if the unary constraint on $N$ is inconsistent with $v$
        $D_N \leftarrow D_N \setminus \{v\}$
```

Figure 2.2: Node consistency algorithm.

to x and the domain of x is narrowed to $\{1..10\}$ since value 0 is inconsistent for inequality constraint $x > 0$. The self connecting arc can be removed from the constraint graph.

The *arc consistency* is applied to the binary constraints of the constraint graph, i.e., constraints of two variables x_i and x_j. The constraint graph is arc consistent if it is arc consistent for every arc (x_i, x_j) in this graph. We define arc consistency as follows. A variable x_i is arc consistent relative to variable x_j if and only if for every value $v_i \in D_i$ there exist a value $v_j \in D_j$ that is a consistent assignment $\langle v_i, v_j \rangle$, i.e., it is a solution for constraint $c(x_i, x_j)$. This is used to define procedure *revise* used by arc consistency algorithms.

There exist different versions of the arc consistency algorithms [90]. The simple arc consistency procedure, called also AC-1, is presented in Figure 2.3.

```
boolean revise($x, y$)
  reduced $\leftarrow$ false
  for each $u \in D_x$ do
    if there is no such $v \in D_y$ that $(u, v)$ is consistent
        $D_x \leftarrow D_x \setminus \{u\}$
        reduced $\leftarrow$ true
  return reduced

void AC-1()
  for each $\langle x, y \rangle \in arcs(G) \wedge x \neq y$
    $Q \leftarrow Q \cup \{\langle x, y \rangle, \langle y, x \rangle\}$
  do
    changed $\leftarrow$ false
    for each $(x, y) \in Q$ do
      changed $\leftarrow$ revise($x, y$) $\vee$ changed
  while changed
```

Figure 2.3: AC-1 arc consistency algorithm.

AC-1 algorithm applies procedure *revise* to all binary constraints in the constraint graph until there is no further change in the domains of FDVs. Since procedure revise checks consistency of variable x relative to y we apply this procedure to both pairs (x, y) and (y, x). The revise procedure takes two FDVs and removes such values from the domain of the first FDV that make the constraint inconsistent, i.e., the constraint is not satisfied. The AC-1 algorithm has the worst case time complexity $\mathcal{O}(d^3ne)$, where d is domain size, n the number of nodes in the constraint graph (FDVs) and e

is the number of edges with binary constraints.

Obviously the algorithm is not very efficient. The successful revision of one FDV forces all the arcs to be revised again. However, in many cases only a small number of arcs is affected by the current revision. This drawback is removed in more advanced consistency algorithms, such as AC-3 (see Figure 2.4). This algorithm deletes from queue Q revised arcs and adds back only these arcs which depend on the revised variables. It is time complexity is $\mathcal{O}(d^3e)$. It can be noted that there exist more advanced arc consistency algorithms [90] with better worst case time complexity. For example, AC-4 has worst case time complexity $\mathcal{O}(d^2e)$. These algorithms does not test many pairs (u, v) which are already known from previous iterations to be consistent. They need additional data structure to make it efficient.

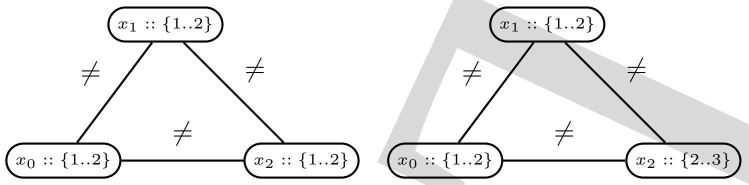
```
void AC-3()
  for each $\langle x, y\rangle \in arcs(G) \wedge x \neq y$
    $Q \leftarrow Q \cup \{\langle x, y\rangle, \langle y, x\rangle\}$
  while $Q \neq \emptyset$
    $Q \leftarrow Q \setminus (x, y)$
    if revise($x, y$)
      $Q \leftarrow Q \cup \{\langle z, x\rangle | (z, x) \in arcs(G) \wedge z \neq x \wedge z \neq y\}$
```

Figure 2.4: AC-3 arc consistency algorithm.

Example 2.7: Consider the constraint graph depicted in Figure 2.1 with FDVs $x :: \{1..10\}$ (the domain after applying node consistency algorithm) and $y :: \{0..10\}$. The arc consistency algorithm prunes the domains to $x :: \{1..9\}$ and $y :: \{2..10\}$. Values 0 and 1 have been removed from the domain of y since there is not any value in the domain of x which is lower than 0 or 1. Similarly value 10 has to be removed from the domain of x since it is not lower than any value in the domain of y. The graph is arc consistent but the arc between x and y cannot be removed. The arc consistency algorithm will be executed again if any change in the domains of x or y will occur.

Example 2.8: Consider constraint graphs depicted in Figure 2.5. They depict three binary inequality constraints on variables x_0, x_1 and x_2. Arc consistency can be applied to both constraint graphs. Application of arc consistency to the example depicted in figure (a) does not affect the domains of x_0, x_1 and x_2. All three binary constraints are consistent separately and arc consistency cannot prune the domains of their FDVs. The FDVs from the constraint graph of figure (b) cannot be pruned either for the same reason. Better consistency method could discover that value 2 can be removed from the domain of x_2.

As indicated in example 2.8 arc consistency is limited to pruning domains for binary constraints. There exist methods to handle higher levels of consistency, such as generalized arc consistency (GAC) or path consistency (discussed in the next section). The generalized arc consistency [11, 78] can be applied to any constraint of arity n. A simple implementation of GAC algorithm is presented in Figure 2.6. The algorithm checks each constraint c_i for possible inconsistency. This is achieved by checking



(a) inconsistent constraint graph (b) not pruned constraint graph

Figure 2.5: Constraint graphs for inequality constraints.

each variable x of this constraint and each value v from the domain of this variable. Therefore GAC has to check nd times consistency of constraint c_i. If we assume that there exist algorithm that can be used for checking existence of a solution for constraint c_i and it has complexity $\mathcal{O}(c_i)$ then the total complexity of GAC for constraint c_i is $\mathcal{O}(nd \cdot \mathcal{O}(c_i))$.

```
boolean GAC()
  for each constraint $c_i \in C$
    for each $x \in c_i$
      for each $v \in D_x$
        if does not exist a solution for constraint $c_i$
          $D_x \leftarrow D_x \setminus \{v\}$
          if $D_x = \emptyset$ return false
  return true
```

Figure 2.6: General consistency algorithm.

The simple implementation of generalized arc consistency is quite inefficient. There exist a better algorithm [11, 78] that is built based on AC-7 [10]. The AC-7 algorithm for arc consistency is based on the notion of supported values and it searches of a single support for each value. Therefore it needs to check only a subset of values that are in the support set. This algorithm has optimal worst-case time complexity.

Example 2.9: Consider again the constraint graph depicted in Figure 2.5(b) and assume that it is given as a single constraint that implements generalized arc consistency. Application of GAC to this constraint graph will prune the domain of FDV x_2. It will discover that value 2 can be removed from the domain of x_2 since variables x_1 and x_2 cannot have two different values than 2.

In practice GAC is usually not used and constraint programming uses instead specialized consistency algorithms. These algorithms are developed for combinatorial constraint that handle specific constraint problems. Selected algorithms for such constraints are discussed in section 3.4. For example, `alldifferent` constraints can be implemented using algorithms which handle the situations indicated in Example 2.8 correctly with lower computational complexity than GAC(see section 3.4.1).

2.4.2 Path Consistency

A natural extension of node and arc consistency is *path consistency*. The basic idea of path consistency is that, in addition to check arcs of the constraint graph between variables x_i and x_j, further consistency must be checked for variables x_i, x_k, x_j that form a path.

A two variable set $\{x_i, x_j\}$ is path consistent relative to variable x_k if and only if for every consistent assignment $\langle v_i, v_j \rangle, v_i \in D_i, v_j \in D_j$ there is a value $v_k \in D_k$ such that the assignment $\langle v_i, v_k \rangle$ is consistent and $\langle v_k, v_j \rangle$ is consistent. The constraint graph is path consistent if and only if for every binary constraint $c(x_i, x_j)$ and for every k $(k \neq i, k \neq j)$, $c(x_i, x_j)$ is path consistent relative to x_k.

The node and arc consistency detects inconsistent values and removes them from domains of FDVs. The path consistency algorithms can additionally detect that some tuples of values are inconsistent and can be removed from related constraints. This inconsistency cannot always be represented directly using domains of FDVs.

An algorithm for implementing path consistency, called PC-1, that is based on boolean matrices is presented in [90]. Boolean matrices explicitly represent all valid 2-compound labels for constraints by assigning 1 for legal values of (i, j) at related matrix positions and 0 otherwise. Then a special composition operation $\otimes$ on these matrices is defined. It is just like ordinary matrix multiplication except that number multiplication is replaced by logical "and" and addition by logical "or". The value that x and y can take simultaneausly is constrained by the constraint represented by matrix $C_{x,y}$ plus the conjuction of all $C_{x,z} \otimes C_{z,z} \otimes C_{z,y}$ for all $z \in \mathcal{V}$, where $C_{z,z}$ represents a unary constraints on z in form of a binary constraint matrix.

A version of PC-1 is presented in Figure 2.7. It does note use original formulation with binary matrices representing constraints but explicitly operates on pairs of values allowed for a given constraint [25], i.e. the constraint $c(x, y)$ is defined as a set of pairs (u, v) that satisfy this constraint. Deleting a pair from constraint $c(x, y)$ enforces path consistency between this binary constraint and another binary constraint involving the same variables. The worst case time complexity of this algorithm is $\mathcal{O}(d^5 n^5)$.

```
void revise($(x, y), z$)
  for each $(u, v) \in D_x \times D_y$ do
    if there is no such $q \in D_z$ that $(u, q)$ and $(q, v)$ are
       consistent delete $(u, v)$ from constraint $c(x, y)$

void PC-1()
  do
    for k = 1..n
      for i = 1..n
        for j = 1..n
          revise($(x_i, x_j), x_k$)
  while there are changed constraints
```

Figure 2.7: PC-1 path consistency algorithm.

Example 2.10: Consider constraint graphs from Figure 2.5 discussed in Example 2.8. Path consistency can directly discover that the constraints from example (a) are inconsistent since there is no assignment for x_2 when $x_0 = 1, x_1 = 2$ or $x_0 = 2, x_1 = 1$ (the only allowed assignments of x_0 and x_1). It can also prune value domain of x_2 for case (b) since the only allowed value for x_2 for two valid assignments to x_0 and x_1 is 3.

The above example illustrates a simple situation when path consistency can improve propagation when three variables are considered instead of two, as in arc consistency. In general, path consistency can do more powerful pruning by deleting pairs of values from constraints and therefore this information cannot be directly reflected in domains of FDVs. The following example illustrates this ability of path consistency method.

Example 2.11: Consider three constraints $x \leq y$, $y \leq z$ and $x \geq z$ with domains of x, y and z $\{0..2\}$. Arc consistency is not able to deduce that variables must be equal and have value 0, 1 or 2. Path consistency can find out that $y = z$ and $z = x$.

Better algorithms, such as PC-4, have worst case time complexity $\mathcal{O}(d^3n^3)$. Path consistency is too computationally expensive for practical problems and therefore it is usually not used in practice.

Arc and path consistency algorithms enforce consistency on constraint graphs of size 2 and 3, respectively. One can generalize this concept to *k-consistency* (see for example [25]). Basically, a constraint graph is k-consistent if and only if given any consistent instantiation of any $k-1$ distinct variables, there exist an instantiation of any kth variable such that the k values taken together satisfy all of constraints among the k variables. In this context we can say that for binary constraint arc consistency is equivalent to 2-consistency and path consistency to 3-consistency. If there exist non-binary constraints we need to extend path consistency to test ternary constraints as well. We also say that the constraint graph is *strongly k-consistent* if and only if it is j-consistent for all $j \leq k$.

2.4.3 Bounds Consistency

Consistency methods discussed in the previous sections can be applied to arbitrary constraints that can be represented by constraint graphs. In practice, arithmetical constraints that are defined using equalities, inequalities and arithmetical operators, such as add and multiply, are used. This kind of constraints have well defined properties and a solver can be implemented using efficient consistency methods. *Bounds consistency* is one of these methods. This method uses interval arithmetic to derive which values are consistent instead of considering all combinations of values which are allowed by a given constraint.

There are two basic ideas behind bounds consistency. First, the FDV domain is approximated using a lower and upper bound. Second, we use real number consistency of primitive constraints rather than integer consistency. The limitation of considering a lower and upper bounds is quite acceptable and it is often true in practice. In sequel we will denote a minimal and a maximal value in the domain of FDV x as $\min(x)$ and $\max(x)$ respectively.

Formally [56], an arithmetic constraint c is *bounds consistent* if for each variable x of this constraint, there is:

- an assignment of *real* numbers, say $v_1, v_2, \ldots, v_k$ to remaining variables in c, say $x_1, x_2, \ldots, x_k$, such that $\min(x_j) \leq v_j \leq \max(x_j)$ for each v_j and

$x = \min(x), x_1 = v_1, \ldots, x_k = v_k$ is a solution of c, and

- an assignment of *real* numbers, say $v'_1, v'_2, \ldots, v'_k$ to $x_1, x_2, \ldots, x_k$, such that $\min(x_j) \leq v'_j \leq \max(x_j)$ for each v'_j and $x = \max(x), x_1 = v'_1, \ldots, x_k = v'_k$ is a solution of c.

Given the ranges for each FDV we can derive efficient methods for calculating a new range for each variable which is bounds consistent. These methods are used to build efficient propagators for constraint.

To be able to discuss propagators for bounds consistency methods we will first define a functional rule of the form

$$x \ \mathtt{in}\ \{min \mathinner{..} max\} \tag{2.1}$$

which restricts the domain of variable x to interval $\{min \mathinner{..} max\}$. Assuming that the domain can be represented as a set, the semantics of this rule can be defined using set intersection between min/max domain and the original domain of FDV as defined below.

$$D_x \leftarrow \{min \mathinner{..} max\} \cap D_x \tag{2.2}$$

Additionally, this rule detects creation of an empty domain for D_x and notifies its propagator or a solver. This indicates that there is no solution for a given set of constraints and assignment of values to FDVs. An appropriate actions need to be undertaken by the solver as discussed in section 2.5.

It can be noted that functional rule 2.1 fulfills contracting property of the propagator since the intersection of the current domain of a FDV intersected with a given min/max domain will never produce the larger domain for this variable.

The bounds consistency propagator operates on numerical intervals and considers only minimal and maximal values of these intervals. For example, FDV $x :: \{0..10\}$ has an numerical interval which has the minimal value 0 and the maximal value 10. The consistency for primitive arithmetic constraints can be defined using previously defined functional rule 2.1 and intervals computed using lower and upper bounds of FDVs' domains. A typical definition of a propagator for a constraint of three FDVs x_1, x_2 and x_3 is of the form

$$\begin{aligned} &x_1 \ \mathtt{in}\ \{min_1 \mathinner{..} max_1\} \\ &x_2 \ \mathtt{in}\ \{min_2 \mathinner{..} max_2\} \\ &x_3 \ \mathtt{in}\ \{min_3 \mathinner{..} max_3\} \end{aligned} \tag{2.3}$$

where values min_i and max_i are new allowed minimal and maximal values for respective variables. The functional rule 2.1 is used here to define a propagator for a constraint of three variables. Since values min_i and max_i may depend on values of $x_j, i \neq j$ (as it is the case in Example 2.13) a single execution of this propagator may not compute the fixpoint. In such the case our propagator will not be idempotent (as one of properties defined for propagators). The propagator can be easily made idempotent when the new values for all variables involved in bounds consistency computation 2.3 are computed using the *fixpoint iteration*. The fixpoint iteration simply computes iteratively new values for all involved variables until no further changes of their domains occur. The decision whether the propagator is idempotent or no is an implementation decision. If the propagator is not idempotent it needs to be placed in the queue of constraints for recomputation when any of its FDVs change the domain otherwise the recomputation is not required. Obviously more advanced techniques for recomputation of selected functional rules can also be used to improve solver performance.

Example 2.12: Consider constraint $x < y$. The propagation rules for this constraints are as follows.

$$x \texttt{ in } \{-\inf\ ..\ \max(y) - 1\} \tag{2.4}$$
$$y \texttt{ in } \{\min(x) + 1\ ..\ \inf\} \tag{2.5}$$

When rules 2.4 and 2.5 are applied to $x :: \{0..10\}$ and $y :: \{0..10\}$ the following rules are executed (assuming that the minimal and maximal integers allowed in our solver, denoted by -inf and inf, are -100000 and 100000 respectively).

$$x \texttt{ in } \{-100000\ ..\ 9\}$$
$$y \texttt{ in } \{1\ ..\ 100000\}$$

This directly yields $x :: \{0..9\}$ and $y :: \{1..10\}$ which are correct values.

Bounds consistency is not limited to one or two variables like node and arc consistency techniques. It can be applied to constraints with more than two variables and prune their domains. Unlike path consistency it does not prune pairs of values. In this sense it can be considered as a hyper-arc consistency method. The following example illustrates application of bounds consistency to a constraint of three variables.

Example 2.13: Consider constraint $x + y = z$. The propagation rules for this constraints are as follows.

$$x \texttt{ in } \{\min(z) - \max(y)\ ..\ \max(z) - \min(y)\} \tag{2.6}$$
$$y \texttt{ in } \{\min(z) - \max(x)\ ..\ \max(z) - \min(x)\} \tag{2.7}$$
$$z \texttt{ in } \{\min(x) + \min(y)\ ..\ \max(x) + \max(y)\} \tag{2.8}$$

When rules 2.6 – 2.8 are applied to $x :: \{1..10\}$, $y :: \{1..10\}$ and $z :: \{1..10\}$ the following result is obtained $x :: \{1..9\}$, $y :: \{1..9\}$ and $z :: \{2..10\}$.

The bounds consistency works fine for numerical intervals that do not contain "holes", i.e., all integers between minimal and maximal values are present in the domain. Consider the following example when the pure bounds consistency does not reduce the domains as it could be possible.

Example 2.14: Consider constraint $x + 3 = y$ with domains $x :: \{1..3, 7..10\}$ and $y :: \{1..10\}$, i.e., we have a "hole" in the domain of x since values 4, 5 and 6 are not present. The bounds consistency algorithm defines the propagation rules as follows

$$x \texttt{ in } \{\min(y) - 3\ ..\ \max(y) - 3\} \tag{2.9}$$
$$y \texttt{ in } \{\min(x) + 3\ ..\ \max(x) + 3\} \tag{2.10}$$

and produces $x :: \{1..3, 7\}$ and $y :: \{4..10\}$ while a more detail analysis using arc consistency, for example, reveals that the domain of y can be pruned to $y :: \{4..6, 10\}$.

The problem with "holes" in domains indicated in Example 2.14 can be solved by using more elaborated method for bounds consistency and does not require to use more computationally intensive propagators based on arc consistency. Instead of using minimum and maximum values in the domain we can apply the required operation on the sub intervals. We will call this consistency as *domain consistency*. To define domain consistency we extend first our functional rule 2.1 to be able to operate on domains and not only on intervals. Its semantics is similar to one defined in 2.2. An intersection of the actual domain of FDV and defined domain is written back to the domain of FDV.

For the constraint from example 2.14 our propagation rules are redefined to the following ones (for the general constraint $x + Const = y$).

$$x \textbf{ in } \{D_y - Const\} \tag{2.11}$$

$$y \textbf{ in } \{D_x + Const\} \tag{2.12}$$

Operations "+" and "–" are defined to operate on an interval and a constant as explained in the algorithm below for add operation.

```
domain $out = \emptyset$;
for interval $i \in D_x$
  $out \leftarrow out \cup \{\min(i) + Const \;..\; \max(i) + Const\}$
```

Similar solutions can be used for other primitive constraints, such as $x + y = z$. However, they influence solvers execution time substantially if domains of FDVs has a number of subintervals. This can pay back when better pruning of domains will speed-up the search procedure.

2.4.4 Generalized Consistency

Consistency methods discussed in the previous sections can be directly used to build propagation rules for a solver. The solver can use these rules to prune domains of FDVs. Node, arc, bounds and domain consistency can be used to implement propagation rules which narrow domains. The narrowing of domains is a basic constraints' mechanism to communicate and collaborate to solve a given constraint problem. Path consistency, on the other hand, does not fit here very well since it removes infeasible tuples from constraints and therefore it cannot make propagation of narrowing of domains to other domains. It can be used to narrow design space and cut infeasible solutions using different techniques.

We have discussed simple consistency techniques which can directly be used to simple arithmetic constraints. However, the consistency techniques can further be extended to operate on larger sets of FDVs which define more complex relations between FDVs. The constraint's consistency methods can be implemented as specialized algorithms that are able to efficiently prune the domains of its FDVs. These algorithms can implement specific reasoning methods coming from different areas, such as operation research, geometry and combinatorics. In this way, the solver offers a good environment for different algorithms to cooperate for solving a given problem. This cooperation is provided by using common FDVs and narrowing their domains using different methods and different algorithms.

Assume that our solver can use different consistency methods. The solver calls for this purpose a special procedure provided by each constraint. We call this procedure `consistency()`, for example. This procedure narrows domains of FDVs that are defined for a given constraint using a specific knowledge of this constraint. For example, for constraint $x < y$ can implement procedure `consistency()` using a fix-point iteration of propagation rules 2.4 and 2.5. The consistency procedure is called by the solver when there is need to recalculate domains of constraint variables.

It is quite straightforward to implement most of consistency procedures for primitive constraints, such as equalities, inequalities and arithmetic constraints. An efficient method involves usually some kind of bounds or domain consistency. It is also possible to implement several different consistency methods for the same constraints and provide "different" constraint for a programmer. This feature can be used carefully when selecting right trade-offs between complexity of a consistency method and its ability to efficiently prune domains of FDVs.

Once the solver can be implemented using different consistency methods one can implement relevant algorithms to support a wide spectrum of constraints. It is possible to provide non-linear constraints (e.g., $x \cdot y = z$) as well as specialized combinatorial constraints (e.g., `cumulative` constraint, see section 3.4.3). Combinatorial constraints are specially interesting since we can build specific algorithms around them and obtain efficient pruning of the involved FDVs. It is also possible to combine a number of primitive constraints into a single combinatorial constraints and reduce in this way an overhead of constraint scheduling and invocation.

Example 2.15: Consider a simple implementation of `alldifferent` constraint that is equivalent in pruning ability to imposing $\frac{n \cdot (n-1)}{2}$ constraints of the form $x_i \neq x_j, i \neq j$.

```
FDV[] V;
for (FDV $v \in V$)
  if $v$ has changed and has a single value
    for (FDV $w \in V \wedge v \neq w$)
      $w$ in $\{value(v)\}'$
```

where $\{value(v)\}'$ denotes a complement of a set containing a single value assigned to v. That is, it has all values between minimum and maximum value allowed in the solver except value of v. The last statement means basically that domain for w can take any values except a value assigned to v.

Advanced combinatorial constraints makes it possible using generalized consistency notion to define specific algorithms for pruning domains of their variables. They use usually specific algorithms to do it. The difference between consistency algorithms and algorithms that are used to solve combinatorial problems is in their goals. An algorithm for solving a combinatorial problem tries to *construct a solution* to this problem. Consistency algorithms try to *remove values* which can never produce a valid solution. The consistency algorithms restrict the possible values that can produce a solution while the other algorithms find partial solutions and from them build a final solution.

There exist many interesting examples of methods used for consistency algorithms. Constraint `alldifferent` can be efficiently implemented using bipartite graph matching as indicated in [75]. `Cumulative` constraint used in scheduling applications has many possible consistency algorithms (see, for example [4]). They are based on operation research methods and can be applied to different scheduling problems. The constraint that constraints nodes in a graph to form a Hamiltonian circuit (`circuit` constraint, see section 3.4.6) can use algorithms for finding tightly connected components in the graph as well as specific data structures to prevent subcycles in the graph [20]. Specific methods for this constraints are discussed in chapter 3.

2.5 Solver Implementation

The solver works with FDVs and constraint, and maintains consistency of conjunction of all imposed constraints with their FDVs. All FDVs and constraints are kept in a so called *constraint store*. The constraint store maintains all necessary data structures for the solver and provides access to all constraints consistency (propagators) and satisfiability procedures as well as access to FDVs and their actual domains. It also maintains solver state during search and makes it possible to organize an efficient backtracking.

The main procedure that each finite domain solver needs to implement is a procedure that enforces consistency of all constraints registered in the constrained store. This procedure simply iteratively calls all individual constraint propagators [17, 51] and therefore it is often called *propagation loop*. The procedure is depicted in Figure 2.8. It calls only propagators for these constraints that need to be evaluated, i.e., their FDVs have changed since the last evaluation. This is handled by the constraints queue Q. It also removes satisfied constraints from the constraint store. This is implemented by the if-statement that checks constraint satisfiability using procedure `satisfied()` of a considered constraint. Note that the propagation loop does not need to know which consistency method is defined by each constraint and therefore it is possible to keep it local and combine different methods in the same solver.

```
boolean solver.consistency()
  while ($Q \neq \emptyset$)
    c $\leftarrow$ fetch constraint from $Q$;
    try
      c.consistency();
      $modifiedFDVs \leftarrow \{x|$ modified by c.consistency()$\}$
      $constraintsToEvaluate \leftarrow \{c|x \in var(c), x \in modifiedFDVs\}$
      $Q \leftarrow Q \cup constraintsToEvaluate$
      if (c.satisfied())
        remove c from constraint store;
    catch (Fail exception)
      return false;
  return true;
```

Figure 2.8: The procedure for propagation loop.

There is several implementation issues related to organization of the propagation loop. First, how the queue of constraints is organized. There is several options here. The usual solution is to organized this queue as FIFO queue but in some cases LIFO might be more appropriate [96]. The priority queue can be a complex implementation structure but offers a lot of flexibility for constraint scheduling for evaluation. Finally, there is a number of implementation choices on how to find which constraints needs to be added to queue Q. The solver can add all constraints with variables that were changed or a more selective strategy can be used. In such a strategy the solver assigns to each changed variable an event that indicates a reason for variable's change. Typical events are:

- *value*– the variable get a single value assigned,
- *min*– the minimum value of a variable has been changed,
- *max*– the maximum value of a variable has been changed,
- *minmax*– the minimum and the maximum value of a variable has been changed, and
- *all*– any change of a variable has occurred, e.g., removing a value from inside an interval.

The propagation loop depicted in Figure 2.8 that implements the consistency procedure for the constraint store cannot, in general, deliver a solution for the constraint problem. In fact, it is not able to provide information whether this solution exist. To be able to answer these questions the solver need to search for a solution or solutions. This search is usually implemented using an algorithm based on the constraint-and-generate method. It assigns a new value to a selected FDV and calls propagation loop. If at some point inconsistency is detected (by obtaining an empty FD) the algorithm need to backtrack. The backtracking annuls the results of the last decision by removing all values changed by this assignment. A new assignment is then performed. A simple depth-first-search algorithm is therefore well suited for this solving technique (in CLP community this method is called *labeling*).

To be able to implement a backtracking search we need to have a method that makes it possible to restore the previous state of the constraint store, i.e., the state which we need to backtrack. There exist three approaches to solve this problem [17]:

- trailing– the solver records only changes in the state and, if needed, undo this changes,
- copying– the solver always copy the whole state (basically the constraint store) and, if needed, removes the old state and returns to the old one, and
- recomputing– the solver always recomputes needed information from already made decisions.

By far dominating approach is trailing and therefore we will concentrate on this method.

The search for possible solutions organizes the search space as a *search tree*. In every node of this tree a value is assigned to a domain variable and a decision whether the node will be extended or the search will be cut in this node is made. We assign to each node of this tree a level that denotes the search level. When the tree is extended the level is incremented and when the tree is cut the level is decremented. Each FDV points to the list of domains. The first element on this list is the current domain at the current search level. The domain captures its actual domain, actual set of assigned constraints, and the stamp when the FDV has been assigned a new domain. When during the search a domain is pruned it is updated using a special procedure. This procedure takes into account the stamp and the current level. If the level and the stamp are the same, the domain is updated directly. While, if the level is higher than the stamp a new domain with the new stamp is put at the top of the list. In case of backtracking, all domains with the with stamp equal to current level are removed. In

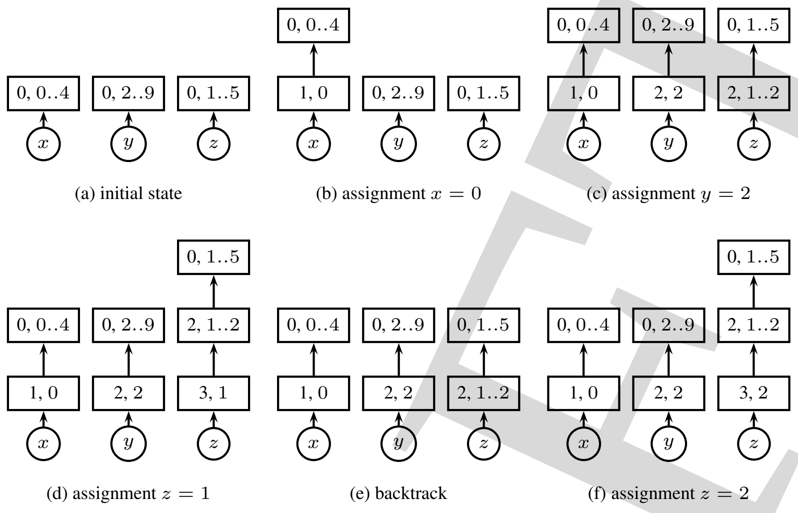


Figure 2.9: An example of trailing for three finite domain variables during search.

this way we only update variables that need to be updated. The access to the current domain of the variable is simply handled by a pointer to current domain of FDV.

Example 2.16: Consider a simple example of three FDVs $x :: 0..4, y :: 2..9, z :: 1..5$ and constraints $x < z, y \geq z$. Figure 2.9 depicts consecutive states of the constraints store during search. Each FDV is depicted as a circle and points to its domain represented as a box. In this box there is specified a stamp and a current domain of the variable. Figure 2.9(a) depicts an initial state. FDVs x, y and z have their initial domains and stamp=0. Next figures depict states during the search. Figure 2.9(b) represents the state after assigning 0 to x. In this case, a new value for x has been created with stamp=1 (since the search increased the search level to 1). The old value is still available with stamp=0. In the next step the search assigns y to 2. This triggers propagation and the propagation loop updates also domain of z to $1..2$. Both domains have stamp=2. Final assignment $z = 1$ leads to a solution and z is update to 1 with stamp=3. Assume now that our search procedure would like to find a next solution and therefore it backtracks the last assignment. This is depicted in Figure 2.9(e) that is identical to Figure 2.9(c). The new assignment to z can be done again and z gets value 2 which is produces a next solution.

The propagation loop that is used to prune domains of FDVs together with implementation of backtracking makes it possible to implement search methods. Search methods will be discussed in chapter 5 but we present here a simple search algorithm that can be used to find a solution to our constraint problem. The algorithm is presented in Figure 2.10. This algorithm is recursive. It selects FDV from the vector of FDVs and proceeds with consecutive variables. Each time the algorithm is entered the propagation loop is invoked. In case, the constraint store is consistent and there is no more variables to assign a value the solution is found. Otherwise the algorithm tries to assign a value v to a current variable and recursively calls itself. When the recursive call returns false (i.e., there is no solution for given constraints) the backtracking

```
boolean solver.search(level, $\mathcal{V}$)
  if solver.consistency()
    if $\mathcal{V} \neq \emptyset$
      pick one variable $var$ from $\mathcal{V}$
      for each $v \in D_{var}$
        impose $var = v$
        if solver.search(level+1, $\mathcal{V} \setminus var$)
          return true
        else        // backtrack
          remove FDVs on trail with stamp=level
      return false   // no more values to assign for $var$
    else
      return true    // assignment for all FDVs found
  else
    return false     // inconsistent constraints
```

Figure 2.10: The simple search algorithm.

procedure is initiated. All created values for FDVs are removed from the trail and the algorithm tries the next value from the domain.

In chapter 5 we discuss search algorithms in more details. We will assume that there exist implementation of trailing or similar methods and we are able to undo FDV assignment and backtrack to previous constraint store state.

Questions and Exercises

1. Consider the CSP of the constraint $z = y + 2x \wedge x \geq 2 \wedge z > y + 2$ on the domains $D_x = D_y = D_z = \{1..5\}$. Transform it to an equivalent CSP which is (a) node-consistent, (b) arc-consistent, (c) bounds consistent. Compare your computed results with the behavior of your favorite finite domain solver on the original CSP. Give an example CSP which demonstrates incompleteness of the your finite domain solver.

2. Use bounds consistency and domain consistency to prune the domains of FDVs involved in the constraint $x + y = z$, $x :: \{1..3, 12..20\}, y :: \{1..2, 7..8\}, z :: \{0..30\}$.

3. Sudoku is a placement puzzle with simple rules. The puzzle is given as a grid of size 9×9 built of 3×3 grids, called regions (see Figure 2.11). Each filed in the grid has to be assigned with a number between 1 and 9. The special rules define restrictions on these assignments. The rules require that each row, each column and each region has assigned numbers that are all different. The puzzle is given with some fields that already have assigned numbers. The goal is to find numbers that need to be assigned to the empty fields. Properly constructed sudoku has a single solution. Define the constraints that need to be fulfilled to find a solution for the sudoku depicted in Figure 2.11.

2	6		3					
5						7		
					1		4	
6			5			2		
		4			8			1
	5		9					
		7						3
					4		1	6

Figure 2.11: A sudoku puzzle example.

Three

Constraints

It does not exist generally accepted constraints taxonomy. Each finite domain solver provides a set of constraints that are defined based on different solvers needs and application areas for the solver. Solvers always provide arithmetic constraints and a number of combinatorial constraints. Propositional and some kind of conditional or reified constraints are sometimes defined as well. In this book, we will shortly review arithmetic constraints and then discuss propositional and reified constraints. Finally, we will concentrate on combinatorial constraints that provide efficient pruning methods for different classes of combinatorial problems.

3.1 Arithmetic Constraints

Inequality is the cause of all local movements.

Leonardo da Vinci (1452-1519)

Arithmetic constraints define relations between FDVs. This constrains have a form $Expr\ \mathcal{R}\ Expr$, where $\mathcal{R} \in \{=, \neq, <, \leq, >, \geq\}$ and $Expr$ is built using operators $\{+, -, \cdot, /\}$ with their usual meaning. Unlike constraints defined in linear programming, finite domain constraints can be defined over non-linear expressions. However, this constraints can yield less propagation then constraints over liner expressions.

A solver usually offers a limited number of primitive constraints and the user or a compiler need to translate complex expressions to these constraints. For example, if a solver offers only a constraint of a form $x + y = z$, constraint $c = a - b$ need to be translated into form $c + b = a$.

Example 3.1: Consider the following constraint $x \cdot y + 2 \cdot q < x$ and assume that our solver offers only constraints $x \cdot y = z, const \cdot x = z$ and $x < y$. Our constraint need to be translated into following constraints $x \cdot y = t_1, 2 \cdot q = t_2, t_1 + t_2 = t_3, t_3 < x$.

As indicated in section 2.4 each constraint is defined by its propagation rules. The constraint has also a procedure which tests whether the constraint is entailed (satisfied) or not. This is enough to define all arithmetic constraints. Definition of propositional, conditional and reified constraint (see sections 3.2 and 3.3) requires more information about constraints. We define additional propagation rule and a new entailment procedure. They are defined for negation of the original constraint. Consider constraint $x < y$ (see example 2.12). This constraint is defined by propagation rules and the entailment procedure but it also defines a propagation rule and an entailment procedure for negated constraint $x \geq y$. This new rule and the procedure is not used by the constraint directly but it is used by other constraints to either enforce a specific constraint or check constraint entailment or disentailment.

Example 3.2: Using arithmetic constraints we can implement, for example, knapsack problems. Consider a simple hardware/software partitioning that can be modeled using knapsack problem. Assume that we have a set of n modules that can be implemented in software or hardware. Each module i has assigned execution time in software ts_i and in hardware th_i. Additionally, hardware implementation of module i requires area a_i and the total available hardware area is A. We would like to find the fastest implementation that does not exceed hardware area constraint. The problem can be modeled using the following constraints.

$$\begin{aligned}
&\forall i : p_i :: \{0..1\} \qquad (3.1)\\
&T_h = max([p_1 \cdot th_1, p_2 \cdot th_2, \ldots, p_n \cdot th_n])\\
&T_s = \sum_{i=1}^{n}(1 - p_i) \cdot ts_i\\
&T = max([T_h, T_s])\\
&\sum_{i=1}^{n} p_i \cdot a_i \leq A
\end{aligned}$$

where $p_i = 0$ denotes software implementation and $p_i = 1$ denotes hardware implementation. Execution time in hardware is maximum of execution times of all units assigned to hardware (we assume parallel execution of all modules assigned to hardware) and execution time is software is the sum of execution times of all modules (we assume single processor implementation and sequential execution of modules). The total execution time T is maximum of execution times in software and hardware. Finally, an area constraint does not make possible to allocate more modules to hardware than area limit A allows.

Note that in the above formulation constraint max can be replace by inequality constraint "$\geq$" of a form $T \geq T_h \wedge T \geq T_s$. Our formulation in this case uses purely arithmetic constraints. This, however, requires additional assignment during search. Variable T need to be assigned a lowest value allowed in its domain after all other assignments to get a single value

To find a minimal execution time we need to minimize value T.

Some arithmetic constraints can be implemented as combinatorial constraints. For example, "Sum" and "Weighted Sum" are often used to define constraints and cost functions and a solver can benefit of this implementation.

3.2 Propositional Constraints

All imposed constraint are considered as a single new constraint that is a conjunction of all constraints. Sometimes we would like to escape from this definition and

```
void Disjunctive.consistency()
  numberEntailed ← 0, numberNotEntailed ← 0;
  for each $C_i$
    if satisfied($C_i$)
      numberEntailed ← numberEntailed + 1
      exit loop
    else
      if notSatisifed($C_i$)
        numberNotEntailed ← numberNotEntailed + 1
      else  // entailment of constraint $C_i$ is unknown
        j ← i
  if numberEntailed = 0
   if numberNotEntailed = n - 1
     consistency($C_j$)
   else
     if numberNotEntailed = n
       FAIL
```

Figure 3.1: An example propagation rule for disjunction of constraints.

would like to define other logical relations between constraints. Some solvers make it possible to define conjunction, disjunction, negation, equivalence and implication of constraints. This can be achieved using new propagation rules and entailment procedures introduced in the previous section. Note that these constraints will take as arguments another constraints.

Consider, for example, a disjunction of constraints of the form $C_1 \vee \cdots \vee C_n$. The propagation rule defined in Figure 3.1 represents an example implementation of this constraint. The constraint is entailed if any constraint C_i is entailed.

It can be noted that the propagation of this constraints is very weak. Basically the constraints watches all n constraints entailment and disentailment. It executes consistency method of constraint C_j if all other constraints $C_i, i \neq j$ are not entailed. It also fails when all n constraints are not satisfied. A related entailment procedure detects situations when one constraint C_i is entailed and the constraint is removed from further considerations by solvers propagation loop (see Figure 2.8). Similar propagation rules can be defined for other propositional constraints.

Example 3.3: Consider resource sharing constraints for scheduling and allocation. We assume that there exist a number of tasks which can be allocated and scheduled on a number of processors. Each task has three FDVs that model starting time (t_i), its execution time (d_i) and processor number allocated for its execution (p_i). Disjunctive constraint 3.2 states that computations of tasks i and j do not overlap (either i is before j or j is before i) or they are allocated to different processors.

$$t_i + d_i \leq t_j \vee t_j + d_j \leq t_i \vee p_i \neq p_j \tag{3.2}$$

Table 3.1: The consistency and satisfiability methods for reified constraints.

consistency	`if` $\min(B) = 1$ `then` consistency(*constraint*) `if` $\max(B) = 0$ `then` notConsistency(*constraint*) `if` satisfied(*constraint*) `then` B `in` $\{1..1\}$ `if` notSatisfied(*constraint*) `then` B `in` $\{0..0\}$
satisfiability	$(\min(B) = 1 \wedge$ satisfied(*constraint*)$) \vee$ $(\max(B) = 0 \wedge$ notSatisfied(*constraint*)$)$

3.3 Conditional and Reified Constraints

Propositional constraint gives the solver possibility to escape from pure conjunction of all imposed constraints. They offer a number of alternatives. Sometimes there exist need for some kind of conditional constraints, i.e., constraints hold if specific conditions are fulfilled. Most solvers offer different ways to express such constraints. There usually exist *reified constraints* and "`if-then-else`" type constraints.

The reified constraint reflects its satisfiability into a 0/1 variable B and it is of form

$$constraint \Leftrightarrow B \tag{3.3}$$

This constraint enforces a relationship between *constraint* and variable B. If the constraint holds $B = 1$ otherwise 0, and if $B = 1$ then the constraint must hold otherwise the negated constraint must hold. Table 3.1 defines consistency and satisfiability methods for reified constraints. Note that the constraint propagation is bidirectional. Any change in the constraint entailment is reflected in variable B and setting of variable B to 0 or 1 causes related consistency enforcement from the constraint.

Example 3.4: Consider the problem from Example 3.3. It can be specified using reified constraints as follows.

$$\begin{aligned} & t_i + d_i \leq t_j \Leftrightarrow B_0 \\ & t_j + d_j \leq t_i \Leftrightarrow B_1 \\ & p_i \neq p_j \Leftrightarrow B_2 \\ & B_0 + B_1 + B_2 \geq 1 \end{aligned} \tag{3.4}$$

All three constraints are reified to variables B_i and the last constraint enforces that at least one of these constraints must be fulfilled.

Reified constraints are bidirectional but sometimes one wants to achieve propagation in one direction only. In such cases, conditional constraints can be used. Conditional constraints correspond to "if-then-else" statements in programming languages and have the following forms.

$$\texttt{If}\ constraint_C\ \texttt{Then}\ constraint_T\ \texttt{Else}\ constraint_F \tag{3.5}$$

$$\texttt{If}\ constraint_C\ \texttt{Then}\ constraint_T \tag{3.6}$$

We use purposely capitalized words for this statement to distinguish it from related statements in programming languages. The semantics of this constraint is following. If $constraint_C$ is satisfied then consistency is enforced on $constraint_T$, if

$constraint_C$ is not satisfied the consistency is enforced on $constraint_F$, if present. Nothing happens if $constraint_C$ is in unknown state, i.e., the constraint is active but no consistency on $constraint_T$ or $constraint_F$ is enforced. Moreover, entailment of either $constraint_T$ or $constraint_F$ is not checked and it has no effect on this constraints and specially on $constraint_C$.

Example 3.5: Consider constraints 3.4 and their redefinition using the following conditional constraints.

$$\begin{aligned}
&\textbf{If } t_i + d_i \leq t_j \textbf{ Then } B_0 = 1 \textbf{ Else } B_0 = 0 \\
&\textbf{If } t_j + d_j \leq t_i \textbf{ Then } B_1 = 1 \textbf{ Else } B_1 = 0 \\
&\textbf{If } p_i \neq p_j \textbf{ Then } B_2 = 1 \textbf{ Else } B_2 = 0 \\
&B_0 + B_1 + B2 \geq 1
\end{aligned} \tag{3.7}$$

These constraints define basically the same conditions as disjunctive constraints 3.2 or reified 3.4 but their propagation is weaker. For example, if two last conditional constraints set $B_1 = 0$ and $B_2 = 0$ then the last constraint sets $B_0 = 1$ but this cannot propagate to enforce consistency of constraint $t_i + d_i \leq t_j$.

3.4 Combinatorial Constraints

3.4.1 Alldifferent

The `alldifferent` constraint ensures that all FDVs on a given list have different values. This constraint is specified as follows

$$\texttt{alldifferent}([X_1, \ldots, X_n]) \tag{3.8}$$

Functionally this constraint is equivalent to inequality constraints on all pairs of the FDVs present in the list. A simple generalized consistency method is presented in Example 2.15, however, there are better consistency methods.

Example 3.6: Consider a time tabling problem adopted from [72] for the purpose of this book. We have six tasks that need to be executed on a single processor. Each task need 1s for its execution. The tasks have additionally timing requirements that specify intervals when a task need to be executed. Table 3.6 specifies these requirements.

Table 3.2: Intervals for task execution.

Task	Execution interval
$task_1$	3..6
$task_2$	3..4
$task_3$	2..5
$task_4$	2..4
$task_5$	3..4
$task_6$	1..6

The problem can be formalized by introducing six FDVs t_i that define start time of task i and the following constraints on these variables.

$t_1 :: \{3..6\}, t_2 :: \{3..4\}, t_3 :: \{2..5\},$
$t_4 :: \{2..4\}, t_5 :: \{3..4\}, t_6 :: \{1..6\},$
`alldifferent`$([t_1, t_2, t_3, t_4, t_5, t_6])$

Consistency checking of the above constraints can give different results depending on the propagation algorithm used. The algorithm presented in Example 2.15 will not prune the domains at all before starting search. The algorithms, such as presented in [75] or [72] will produce the following result directly.

$t_1 = 6, t_2 :: \{3..4\}, t_3 = 5, t_4 = 2, t_5 :: \{3..4\}, t_6 = 1,$

Regardless of the propagation algorithm implemented in the solver the search finds two solutions to the imposed constraints:

$t_1 = 6, t_2 = 3, t_3 = 5, t_4 = 2, t_5 = 4, t_6 = 1,$ or
$t_1 = 6, t_2 = 4, t_3 = 5, t_4 = 2, t_5 = 3, t_6 = 1$

Obviously depending on the propagation ability to cut the search space it may require more or less search to find these solutions. In this example it is rather straightforward since the problem is very small.

Consistency Algorithms

The propagators for `alldifferent` constraint are well studied [95]. A simple propagator can be implemented using a method suggested in example 2.15. Pruning power of this implementation is equal to using $\frac{n\cdot(n-1)}{2}$ inequality constraints. More advanced implementations use either bounds consistency or hyper-arc consistency.

There exist several bounds consistency methods [72, 55] that have complexity $\mathcal{O}(n \log n)$, where n is the number of FDVs. There is an algorithm [58] that has complexity $\mathcal{O}(n)$ plus the time required for sorting the intervals endpoints. In particular, if the endpoints are from a range of size $\mathcal{O}(n^k)$ for some constant k, the algorithm runs in linear time. The constraint can also be implemented using hyper-arc consistency proposed by Régin in [75]. It is based on bi-partite graph matching and has complexity $\mathcal{O}(n^2 d^2)$, where n is the number of FDVs and d is the largest domain size.

Consider first bounds consistency methods and its ability to use global information existing in `alldifferent` constraint. The method builds on Hall's Theorem [40] that basically states a necessary and sufficient condition for the existence of a solution. The following Theorem is stated in terms of `alldifferent` constraint. Note, that $|K|$ denotes cardinality of a set.

Theorem 1 (Hall's Theorem)**.** *The constraint* `alldifferent`$([x_1, \ldots, x_n])$ *with respective variable domains* $D_1, \ldots, D_n$ *has a solution if and only if no subset* $K \subseteq \{x_1, \ldots, x_n\}$ *exist such that* $|K| > | \cup_{x_i \in K} D_i|$.

It basically says that we must have enough values in respective domains for assignment to FDVs. The following example illustrates this principle.

Example 3.7: Consider the following problem:

$x_1 :: \{1..5\}, x_2 :: \{2..3\}, x_3 :: \{2..3\}, x_4 :: \{2..3\},$
`alldifferent`$([x_1, x_2, x_3, x_4]).$

This problem is inconsistent since there are only two values available, namely 2 and 3, and there are three FDVs, x_2, x_3 and x_4. According to Hall's theorem $K = \{x_2, x_3, x_4\}$ and $|K| = 3$ but $| \cup_{x_i \in K} D_i| = |\{2..3\}| = 2$. Therefore the condition $3 > 2$ is fulfilled and it does not exist any solution for our problem.

Based on the Hall's theorem Puget introduced a bounds consistency algorithm for `alldifferent` constraint [72]. Since we are mainly interested in intervals we introduce first a definition of Hall interval .

Definition 1 (Hall interval)**.** *Given a constraint* `alldifferent`$([x_1, \ldots, x_n])$*, and an interval I, let K_I be the set of variables x_i such that $D_i \subseteq I$. We say that I is a Hall interval if and only if $|I| = |K_I|$.*

Example 3.8: Consider the following problem:

$x_1 :: \{1..5\}, x_2 :: \{1..2\}, x_3 :: \{1..2\},$
`alldifferent`$([x_1, x_2, x_3])$.

The Hall interval is $I = \{1..2\}$ and $K_I = \{x_2, x_3\}$.

Finally we introduce a proposition that can be used to build an algorithm for bounds consistency.

Proposition 1 (Puget [72])**.** *The constraint* `alldifferent`$([x_1, \ldots, x_n])$ *where no D_i is empty is bounds consistent if and only if*

1. *for each interval $I : |K_I| \leq |I|$,*
2. *for each Hall interval $I : \{min(D_i)..max(D_i)\} \cap I = \emptyset$ for all $x_i \notin K_I$.*

The conditions from Proposition 1 can be directly used to achieve bounds consistency. An algorithm can check the condition (1) and if is is not fulfilled will report inconsistency (fail). Condition (2) can be used for pruning of domains. Namely, for each FDV which domain does not belong to Hall interval we can remove values belonging to Hall interval.

Example 3.9: Consider the following problem:

$x_1 :: \{1..5\}, x_2 :: \{1..2\}, x_3 :: \{1..2\},$
`alldifferent`$([x_1, x_2, x_3])$.

The Hall interval is $I = \{1..2\}$ and $K_I = \{x_2, x_3\}$. Therefore we can prune domain of variable x_1 to $\{3..5\}$.

The algorithm for computing bounds consistency can be done in two phases [72]. One phase deduces new min bounds for FDVs and the second phase max bounds. Puget indicates that the second phase can use the algorithm proposed for the first phase if we replace intervals by their inverse. For example, and interval $\{2..3\}$ will be replaced by $\{-3..-2\}$. Therefore we will concentrate on the first phase for updating minimal values for intervals only.

The proposed algorithm sorts all variables in ascending order of max values in their domains. It then treats one variable $x[i]$ at each iteration. It maintains for each variable $x[j]$ for $j < i$ a number $u^i[j]$ defined as follows:

$$u^i[j] = min(x[i]) + |\{k : k < i, min(x[k]) \geq min(x[j])\}| - 1 \tag{3.9}$$

This number is computed incrementally using the following formula:

$$u^i[j] = u^{i-1}[j] + Bool(min(x[i]) \geq min(x[j])) \tag{3.10}$$

where $Bool(exp)$ is equal to 1 if exp is true and 0 otherwise.

The algorithm uses $u^i[j]$ to detect Hall intervals and then make related pruning of other domains. The following proposition suggest how to find Hall intervals using $u^i[j]$.

Proposition 2 (Puget [72]). *Using above definitions, if $u^i[j] = max(x[i])$ then the interval $I = \{min(x[j])..max([x[i])$ is a Hall interval.*

Proof. The width of I is $|I| = max(x[i]) - min(x[j]) + 1$. The number of variables included in I is $|\{k : max(x[k]) \leq max(x[i]), min(x[k]) \geq min(x[j])\}|$ which is greater or equal to $|\{k : k \leq i, min(x[k]) \geq min(x[j])\}|$ which is equal to $u[j] + 1 - min[j]$. If $u[j] = max(x[i])$ then the above number is equal to $|I|$ which concludes the proof. □

A version of the algorithm that implements the ideas presented above is depicted in Figure 3.2. It has $\mathcal{O}(n^2)$ time complexity. The improved version of this algorithm which uses balanced binary trees has complexity $\mathcal{O}(n \log n)$.

Example 3.10: Consider the problem introduced in Examples 3.8 and 3.9. The algorithm sorts FDVs in in ascending max order first and produces the following ordered vector $x_2 :: \{1..2\}, x_3 :: \{1..2\}, x_1 :: \{1..5\}$. Then it calls `Insert(i)` method three times as follows.

```
Insert(1)
  u[1] ← 1
Insert(2)
  u[2] ← 1
    min[1] ≥ min[2] hence u[2] ← 2
  since u[2] = max[2] we have found Hall interval {1..2}
  pruning: $x_1$ in {3..inf}
Insert(3)
  u[3] ← 1
    min[1] ≥ min[3] hence u[3] ← 2
    min[2] ≥ min[3] hence u[3] ← 3
```

The algorithm prunes therefore x_1 to $\{3..5\}$.

The algorithm presented in Figure 3.2 can be easily adapted to make domain consistency. Procedure `IncrMin` can be replaced by procedure `Remove`. This procedure removes all Hall intervals `{a..b}` from all FDVs that do not belong to this interval. In this way, it is possible to remove values from inside of intervals.

Another implementation of `alldifferent` constraint uses hyper-arc consistency algorithm. Such an algorithm proposed by Régin [75] will be discussed here. The

```
// x is an array containing the variables
// u, min and max are arrays of integers
sort(x)  // in ascending max order
for i = 1..n
  min[i] ← min(x[i])
  max[i] ← max(x[i])
for i = 1..n
  Insert(i)

Insert(i)
  u[i] ← min[i]
  bestMin ← inf
  for j = 1..i-1
    if min[j] < min[i]
      u[j] ← u[j] + 1
      if ( u[j] > max[i] ) FAIL
      if u[j] = max[i] ∧ min[j] < bestMin
        bestMin ← min[j]
    else
      u[i] ← u[i] + 1
  if ( u[i] > max[i] ) FAIL
  if u[i] = max[i] ∧ min[i] < bestMin
    bestMin ← min[i]

  if bestMin ≤ N + 1
    IncrMin(bestMin, max[i], i)

IncrMin(a, b, i)   // {a..b} is a Hall interval
  for j = i+1..n
    if  min[j] ≥ a
      x[j] in {b+1 .. inf}
```

Figure 3.2: One phase of bounds consistency algorithm for `alldifferent` constraint.

algorithm uses bipartite graphs to represent the constraint and then applies results known from matching theory to filter these values that are not consistent in the constraint. The bipartite graph used to model `alldifferent` constraint is called *value graph* and is defined below.

Definition 2 (Value graph)**.** *Given an `alldifferent` constraint C, the bipartite graph $GV(C) = (X_C, D_C, E)$, where $(x_i, d) \in E$ if and only if $d \in D_i$, is called the value graph of C.*

The value graph is used to achieve hyper-arc consistency and prune values that are not consistent. This consistency technique is built on *matching* . Basically, a matching in a graph is a subset of edges which have a property that no two edges have a node in common. A matching is of maximum cardinality is called *maximum matching*. A matching M covers a set of X if every node in X is an endpoint of an edge in M.

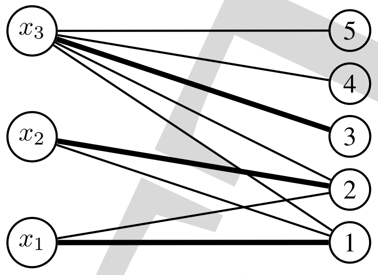


Figure 3.3: A value graph for `alldifferent` constraint from Example 3.11.

Based on this we can define a basic proposition that is used later to define consistency method for the constraint. The following example illustrates the idea of value graph.

Example 3.11: Consider constraint

$$\texttt{alldifferent}([x_1 :: \{1..2\}, x_2 :: \{1..2\}, x_3 :: \{1..5\}]).$$

Its value graph is depicted in Figure 3.3. A possible maximum matching is depicted by bold edges.

There exist usually several maximum matchings for a value graph. For example, another maximum matching for the value graph of Example 3.11 is defined by the following edges $\{(x_1, 2), (x_2, 1), (x_3, 4)\}$. Informally, edges that belong to any possible maximum matching are possible candidates for a solution of `alldifferent` constraint. In our example, edges $(x_3, 1)$ and $(x_3, 2)$ does not belong to any maximum matching. This comes directly from the definition of matching that says that no two edges have a node in common. If, for example, we allow edge $(x_3, 1)$ then neither an edge containing x_1 nor x_2 is possible and it is not possible to find a maximum matching. Similar reasoning can be applied to edge $(x_3, 2)$ and therefore both edges can be removed from the value graph and respective variables can be pruned accordingly. This can be formalized by the following property.

Proposition 3 (Régin [75])**.** *The `alldifferent` constraint C is hyper-arc consistent iff every edge in its value graph GV(C) belongs to a matching that covers X_C in GV(C).*

Proof. See [75]. □

The above proposition creates a basis for constructing an efficient consistency algorithm for `alldifferent` constraint. However, we do not want to find all possible matchings and therefore a more efficient method to find edges that do not belong to any maximum matching is needed. This algorithm can be constructed based on the following proposition.

Proposition 4 (Berge [9])**.** *An edge belongs to a maximum matching iff for some maximum matching, it belongs to either an even alternating path which begins at a free node, or to an even alternating cycle.*

```
build value graph $GV = (X_C, D_C, E)$
M(GV)  $\leftarrow$ ComputeMaximumMatching(GV)
if ($|M(GV)| < |X_C|$) FAIL
RemoveEdgesFromG(GV, M(GV))

RemoveEdgesFromG(GV, M(GV))
  • Mark all edges in $G$ as ‘‘unused’’.
  • Use breadth-first-search to find all even alternating paths
    starting at free vertex and mark their edges as ‘‘used’’.
  • Compute strongly connected components of $G$ and mark edges
    that join two vertices in the same strongly connected
    component as ‘‘used’’.
  for each edge $e$ marked as ‘‘unused’’
    if $e \in M(G)$
      mark $e$ as ‘‘vital’’
    else
      remove $e$ from $G$
```

Figure 3.4: A consistency algorithm for `alldifferent` constraint based on matching.

This proposition makes it possible to determine if an edge belongs to a maximum matching based on three step procedure. First, one maximum matching is computed and then, using alternating paths and cycles, it is decided which edges are necessary for the constraint and which can be removed. An alternating path or a cycle is a path or a cycle whose edges are alternately matching and free (i.e., does not belong to matching). A node is called free for a given matching if it is not incident to a matching edge. For example, nodes 4 and 5 are free for a matching depicted in Figure 3.3. Even alternating cycles can be found using an algorithm for finding strongly connected components (see, for example Tarjan's algorithm [81]).

Example 3.12: Consider a value graph and a matching as depicted in Figure 3.3. There are two even alternating paths which begin at a free node, $(5, x_3, 3)$ and $(4, x_3, 3)$. Therefore values 3, 4 and 5 cannot be removed from the domain of x_3. There is also one even alternating cycle, $(x_1, 1, x_2, 2, x_1)$. This makes related edges crucial for variables x_1 and x_2 and values 1 and 2 cannot be removed from their domains. On the other hand edges $(x_3, 1)$ and $(x_3, 2)$ do not belong to any even alternating path which begins at a free node or to an even alternating cycle and values 1 and 2 can be removed from the domain of x_3.

The algorithm that is built on this principle is depicted in Figure 3.4. It first computes a maximum matching. There exist several algorithms for this. The best one is proposed by Hopcroft and Karp [42] and has complexity $\mathcal{O}(\sqrt{n}m)$, where n is the number of nodes and m is the number of edges. After computing a maximum matching the algorithm checks whether number of variables is higher than number of edges in maximum matching and if it is the case the algorithm fails and reports that there is no solution for the constraint. Otherwise the algorithm removes edges based on Proposition 4 using alternating paths and cycles.

Constraint `alldifferent` restricts the assignment of values to FDVs in such a way that each value can be used only once. A generalization of this constraint makes it possible to specify minimum and maximum number of assignments for each value. This constraint is usually called *global cardinality constraint* (`gcc`) and is provided in some solvers. It assures that each specified value is used a specified number of times by constraint's FDVs. In [76] the generalized arc consistency algorithm based on flow theory has been proposed (the matching theory, used for `alldifferent` constraint, can be viewed as a specialization of flow theory). It has time complexity $\mathcal{O}(n^2d)$, where n is the number of FDVs and d is the number of values in the union of their domains. An approach, presented in [44], also uses matching and flows but it has been used to define the complete bound consistency algorithm. It has time complexity $\mathcal{O}(n+d)$ plus time to sort the bounds of the domains for the assignment FDVs, where n is the number of the assignment FDVs and d is the number of values in the union of their domains. The bounds consistency algorithm that uses Hall sets (generalization of Hall intervals to sets) has been presented for example in [74]. This algorithm runs in time $\mathcal{O}(t+n)$, where t is the time to sort the bounds of the domains of the variables and n is the number of FDVs.

3.4.2 Diff2

The `diff2` constraint takes as an argument a list of 2-dimensional rectangles and assures that for each pair i, j $(i \neq j)$ of such rectangles, there exists at least one dimension k where i is after j or j is after i– rectangles do not overlap. The rectangle is defined by a 4-tuple [O_1, O_2, L_1, L_2], where O_i and L_i are respectively called the origin and the length in i-th dimension. The `diff2` constraint is specified as follows.

$$\texttt{diff2}([[O_{11}, O_{12}, L_{11}, L_{12}], \ldots, [O_{n1}, O_{n2}, L_{n1}, L_{n2}]]) \tag{3.11}$$

Formally, it enforces the following constraint:

$$\forall i, j (i \neq j) \exists k \in \{1, 2\} \text{ such that } (O_{ik} + L_{ik} \leq O_{jk} \vee \\ O_{jk} + L_{jk} \leq O_{ik}) \tag{3.12}$$

It is therefore equivalent to imposing $\frac{n \cdot (n-1)}{2}$ constraints 3.12.

Example 3.13: Consider `diff2` constraint specification

$$\texttt{diff2}([[\{0..8\}, \{0..3\}, 2, 2], [0, 0, 1, 1], [0, 1, 1, 1], [0, 3, 2, 1], \\ [2, 1, 1, 1], [7, 1, 1, 4], [2, 0, 8, 1]])$$

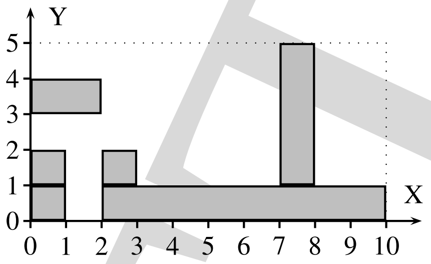
A graphical representation of this constraint is depicted in Figure 3.5(b). The pruning procedure produces rectangle R = [{2..5, 8}, {1..3}, 2, 2]. The rectangle can be placed in dark gray areas depicted in Figure 3.5(c).

The `diff2` constraint can be used to solve different packing problems. In assignment and scheduling it can define resource assignment constraints. In this case, a rectangle represents a task and O_{i1} is used to denote T_i, O_{i2} denotes R_i, L_{i1} denotes D_i and L_{i2} is 1.

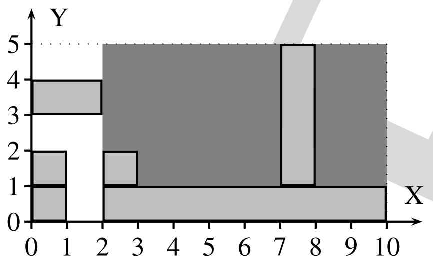
Example 3.14: Consider a problem of scheduling four tasks on two processors. Each task can be executed on either processor 1 or 2. Task one has to be executed before two and three, and task four has to be executed after task two. The

R =[{0..8}, {0..3}, 2, 2]
S =[[0, 0, 1, 1], [0, 1, 1, 1], [0, 3, 2, 1],
[2, 1, 1, 1], [7, 1, 1, 4], [2, 0, 8, 1]]

(a) definitions of rectangles R and S



(b) geometrical representation of S



(c) allowed area to place rectangle R (dark gray areas).

Figure 3.5: An example of `diff2` constraints pruning.

following model defines constraints for task assignment and scheduling. We use $t_i :: \{0..\inf\}$ to denote task start time, $p_i :: \{1..2\}$ assigned processor and d_i task duration.

$t_1 + d_1 \leq t_2$
$t_1 + d_1 \leq t_3$
$t_2 + d_2 \leq t_4$
`diff2`$([[t_1, p_1, d_1, 1], [t_2, p_2, d_2, 1], [t_3, p_3, d_3, 1], [t_4, p_4, d_4, 1])$

Consistency Algorithms

The propagators for `diff2` constraint are not widely published but they are implemented in several solvers. SICStus implements `disjoint2` constraint [83] and CHIP `diffn` constraint [23]. Both constraints have similar semantics but `diffn` constraint is more general. It imposes a constraint on n-dimensional rectangles instead of two-dimensional. The constraint can obviously use geometrical knowledge to calculate possible locations of rectangles. The knowledge on available space for placing rectangles as well as obligatory regions for rectangles can be efficiently used to prune variables of this constraint. In [79] in chapter 7 there is discussion on a method for constraint implementation on two dimension non-overlapping rectangles. Beldiceanu and Carlsson propose to use a method know from computational geometry [70], called *sweep* or *plane sweep* (in two dimensions), for propagator implementation [7]. The paper [51] discusses also a possible implementation of this constraint. The discussion in this chapter will based on this material.

This constraint can use several different propagators. For example, it is relatively easy to check whether there is enough space to place all rectangles in the limits defined by each rectangle FDVs domains. This quick check can directly generate fail if the

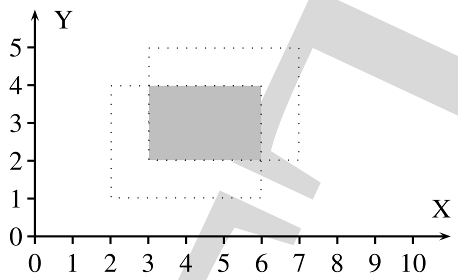


Figure 3.6: The obligatory region for the rectangle defined in Example 3.15.

condition is not fulfilled. If the condition is fulfilled we can start computing more computationally expensive propagators.

The `diff2` constraint propagator is based on the observation that even if a rectangle is not yet placed it can have a region that it will always occupy, called the *obligatory region*. This region is computed using the following formulae.

$$\begin{aligned}
&x_{min} = max(O_1) \\
&x_{max} = min(O_1) + min(L_1) \\
&y_{min} = max(O_2) \\
&y_{max} = min(O_2) + min(L_2) \\
&\textbf{if } x_{min} < x_{max} \wedge y_{min} < y_{max} \\
&\qquad obligatory\ region = [x_{min}, y_{min}, x_{max} - x_{min}, y_{max} - y_{min}]
\end{aligned} \tag{3.13}$$

Example 3.15: Consider rectangle $[x, y, d_x, d_y]$ with the following values $x :: \{2..3\}, y :: \{1..2\}, d_x :: \{4..5\}, d_y :: \{3..6\}$. The obligatory region for this rectangle is depicted in Figure 3.6. as the gray area. It is defined based on formulae 3.13 as $[3, 2, 3, 2]$.

Once the obligatory regions have been computed there exist information which regions are forbidden for placing rectangles. We call these regions *forbidden regions*. Figure 3.7 depicts the forbidden region, represented by gray and dark gray polygons, for selecting x and y for rectangle $[x :: \{0..10\}, y :: \{0..10\}, 2, 2]$. The forbidden regions are used directly to find possible pruning of origins and lengths of rectangles. This deduced information cannot always be represented using domains of variables. For example, the forbidden region in Figure 3.6 does not permit to place a new rectangle in such a way that it overlaps with this area. It basically means that selecting a pair of (x, y) having their values in this region is not possible since it will overlap with the obligatory region. However, if the rectangle can be placed in different positions under and over the forbidden region as well as on the left and right of this region then any value of x and y can be used. Only a specific combinations of x and y are forbidden but this is not possible to represent in the separate definitions of these variables. Consider another rectangle $[x :: \{0..10\}, y :: \{1..2\}, 2, 2]$. This rectangle cannot be place below and under the forbidden region and therefore it has to be placed either on the left or on the right of this region. Its variable x can be pruned to $\{0..1, 6..10\}$.

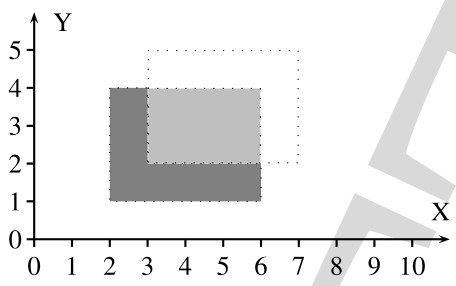


Figure 3.7: The forbidden region for the rectangle defined in Example 3.15.

This represents a situation that we will try to find. This situation is characterized by an existence of a "barrier" which restricts the placement of a rectangle. This barrier can be defined both in X and Y direction. It is used to restrict the origin and length of the rectangle. Here we will concentrate on pruning origins in X direction only. We will use sweep method to find values that can be pruned.

The idea of plane sweep method is quite straightforward [70]. There is a vertical line that sweeps (or scans) the plane from left to right halting at special points, called *event points*. The intersection of the sweep line with the problem data (in our case forbidden regions) contains all the relevant information for the continuation of the sweep. The algorithm uses the two following data structures:

- the *sweep line status* that contains information related to the current position of the sweep line, and
- *event point series* which holds the events to process, ordered from left to right in increasing order of abscissae.They define the halting positions of the sweep line and can be updated dynamically.

In case of `diff2` constraint we perform a sweep for each rectangle R_i and a set of forbidden regions $\mathcal{S}$ that overlap with the ranges of rectangle R_i. The sweep line moves from left to right and stops at the edges of the forbidden regions to check if we can remove the values from domain of x_i. The sweep line status is organized as a list that records for each Y position whether it is "safe" or "forbidden" position. The positions is "safe" if it does not overlap with the forbidden regions and it is "forbidden" otherwise. This is achieved by assigning a counter for each list position that counts number of forbidden regions that overlap with the current position. Value 0 of this counter indicates that it overlaps with 0 forbidden regions ("safe" state) and value greater than 0 indicates number of overlapping forbidden regions ("forbidden" state). The sweep line overlaps with a rectangle if the rectangle either starts at the current position or it is located on both sides of the line.

The pruning of x_i is decided based on the analysis of sweep line status lists. Basically if exist position n on this list with value 0 we know that rectangle R_i can be placed at the current position position of sweep line. Collecting all forbidden intervals on X axis gives us possibility to prune x_i.

Example 3.16: Consider the rectangle defined in Example 3.15 and its forbidden region defined in Figure 3.7. We check rectangle $[x :: \{0..10\}, y :: \{1..2\}, 2, 2]$

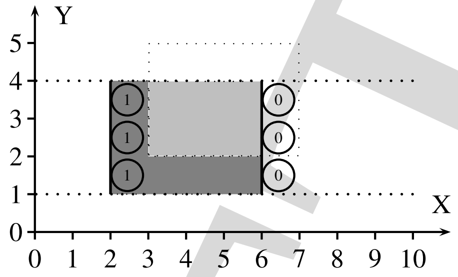


Figure 3.8: An example of the sweep algorithm for rectangles defined in Example 3.16.

using sweep method to identify and apply possible pruning of variable x. Two positions of the sweep line are depicted in Figure 3.8 that are event points for sweep. One position is at 2 and the other one at 6. We can note that the status of all y positions at sweep line location 2 is "forbidden" and the status for all y positions at location 6 is "safe". Therefore the forbidden interval is $\{2..5\}$ and it can be removed from domain of x that becomes $x :: \{0..1, 6..10\}$

The detail algorithm of the sweep method is beyond the scope of this book but an interested reader can find more details on the possible implementation of an algorithm in [7]. The complexity of this algorithm is $\mathcal{O}(n \log n + n \cdot f \log f))$ where n is the number of rectangles and f is the average number of rectangles that could overlap with the domain of placement of a given rectangle R_i. A method that uses similar principles has also been presented by the author in [51].

Example 3.17: Consider the following `diff2` constraint.

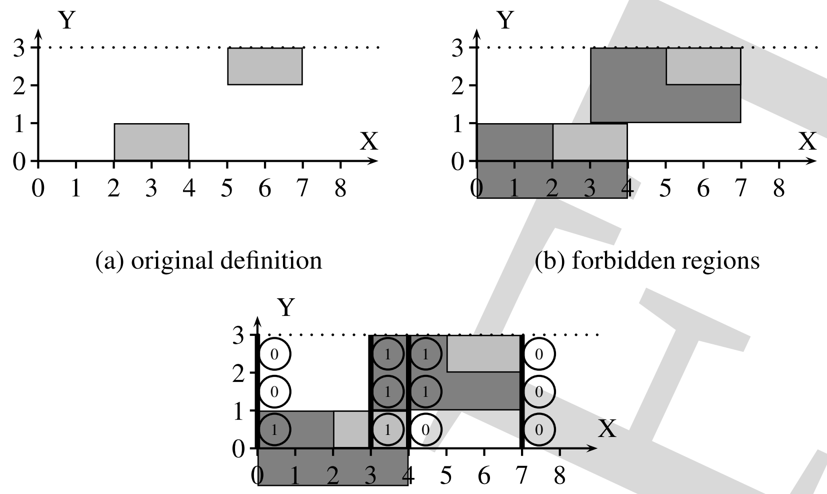
$$\texttt{diff2}([[x_1, y_1, 3, 2], [2, 0, 2, 1], [5, 2, 2, 1]])$$

with $x_1 :: \{0..10\}$ and $y_1 :: \{0..1\}$. Figure 3.9.a) depicts the graphical representation of rectangles 2 and 3. FDV x_1 of rectangle 1 will be pruned using the plane sweep method. Figure 3.9.b) depicts forbidden regions for rectangle 1 and Figure 3.9.c) the sweep lines and their status. x_1 is finally pruned to $\{0..2, 4..10\}$ based on the fact that the region $\{3..4\}$ is excluded from possible origins of the rectangle.

3.4.3 Cumulative

The `cumulative` constraint was originally introduced to specify requirements on tasks that need to be scheduled on a number of resources [1]. It expresses the fact that at any time instant the total use of these resources for the tasks does not exceed a given limit. It has usually four parameters: a list of tasks' starts T_i, a list of tasks' durations D_i, a list of amount of resources (resource capacity) C_i required by each task, and the upper limit of the amount of available resources $Limit$. All parameters can be either FDVs or integers. The constraint is specified as follows.

$$\texttt{cumulative}([T_1, \ldots, T_n], [D_1, \ldots, D_n], [C_1, \ldots, C_n], Limit) \tag{3.14}$$



(a) original definition

(b) forbidden regions

(c) sweep lines and their status

Figure 3.9: An example of the sweep method for rectangles defined in Example 3.17.

Formally, it enforces the following constraint:

$$\forall t \in [\min_{1\leq i\leq n}(T_i), \max_{1\leq i\leq n}(T_i + D_i)] : \sum_{k:T_k\leq t\leq T_k+D_k} C_k \leq Limit \qquad (3.15)$$

where min and max mean the minimum and the maximum value in the domain of an FDV respectively. The constraint ensures that at each time point, t, between the start of the first task (task selected by $min(T_i)$) and the end of the last task (task selected by $max(T_i + D_i)$), the cumulative resource use by all tasks, k, running at this time point is not greater than the available resource limit.

Example 3.18: Assume that we have three tasks that need to be scheduled on two processors. The tasks execution times are 3, 2 and 4 respectively. Tasks have to be started in specific intervals. Task 1 start interval is $\{2..4\}$, task 2 has to be started at interval $\{3..4\}$ and task 3 at time interval $\{4..6\}$. Obviously a task requires a processor to be executed and therefore each task has amount of required resource equal to 1.

$t_1 :: \{2..4\}, t_2 :: \{3..4\}, t_3 :: \{4..6\},$
$limit :: \{1..2\},$
`cumulative`$([t_1, t_2, t_3], [3, 2, 4], [1, 1, 1], limit)$

Note that $limit$ is in interval 1 and 2 that makes it possible to produce schedules which use either one or two processors. Figure 3.10 illustrates one possible generated schedule. Since `cumulative` constraint does not specify task assignment to processors, the placement of tasks 1 and 2 is arbitrary on the figure and does not mean that task 1 has to be executed on processor one and task 2 on processor 2. It

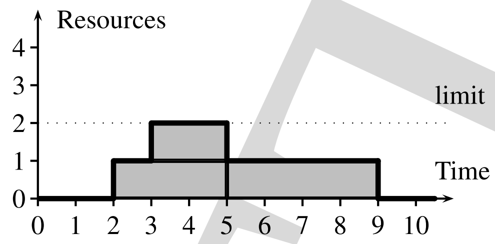


Figure 3.10: An example schedule generated for `cumulative` constraint from Example 3.18.

only reflects the situation that during time interval 3 to 5 we need two processors for execution of our tasks set. The bold line depicts the resource usage profile.

Consistency Algorithms

The `cumulative` constraint is well studied in constraint programming and operational research communities. This constraint offers basic pruning facilities for typical scheduling problems, such as job-shop scheduling. Researchers proposed a number of methods for achieving consistency of this constraint (see for example [2, 4, 18, 19]). The proposed consistency methods use diverse techniques to achieve best pruning and failure detection.

Obligatory parts for tasks, defined in a similar way as obligatory regions for rectangles, can be used to construct *resource histograms*. The histograms can directly be used for pruning start time and duration of a task in situations when number of resources does not allow task execution. *Energy-based reasoning* can also be used to deduce task ordering or achieve updating time bounds for a task [2, 4]. Finally, task intervals can be used to deduce possible start and completion times for tasks. The *edge-finding algorithm* has been proposed for this purpose [4], for example. We will discuss these methods in this section.

Obligatory parts, sometimes called also obligatory intervals, define a minimal interval which will always be occupied by a task. The obligatory part is defined for each task using *latest starting time* (LST) and *earliest completion time* (ECT) of a task. The obligatory part is then defined by the following formulae.

$$\begin{aligned} &LST_i = \max(T_i) \\ &ECT_i = \min(T_i) + \min(D_i) \\ &\texttt{if}\ LST_i < ECT_i\ \texttt{then}\ obligatory\,part = \{LST_i..ECT_i\} \end{aligned} \tag{3.16}$$

All obligatory parts together with their respective minimal resource use, $min(C_i)$, are composed to obtain a resource histogram for the tasks. This is achieved by adding minimal use of resources for all interleaving obligatory parts. The graphical interpretation of this procedure is depicted in Figure 3.11. The obtained resource histogram, depicted as the bold line in Figure 3.11, is then used to narrow other tasks' FDVs. In our example, task 3 requires two resources and has a minimum duration of three and therefore it cannot be placed in all parts because of lack of available resources. This lack of available resources is found when a part of a histogram has difference between resource limit and current utilization lower than the task's resource requirement ($(min(C_i))$). When such a part p_j is detected the pruning of the possible start times of the task is performed using the following formula.

$$t_i \textbf{ in } \{-\inf .. \min(p_j) - \min(d_i), \max(p_j) .. \inf\} \quad (3.17)$$

The following steps constraint the final domain of T_3.

1. in interval {1..3} there is only one resources available and therefore T_3 is constrained by T_3 **in** $\{-\inf ..1-\min(D_3), 3.. \inf\}$ that results in T_3 **in** $\{3..10\}$,
2. in interval {3..4} there is no resources available and the result is T_3 **in** $\{4..10\}$, and finally
3. in interval {4..5} there is only one resource available that results in the final interval for T_3 **in** $\{5..10\}$

The propagator based on obligatory parts and resource histogram is quite straightforward and intuitively obvious but it is quite powerful in its pruning power. It can achieve in some situations more pruning than the propagator for `diff2` constraint that is based on obligatory regions. It happens in situations when obligatory regions cannot yet be defined while obligatory parts already exist and overlap each other. In general, however, these propagators can achieve pruning in different situations and are difficult to compare. They often complement each other in achieving pruning. The following example illustrates the strength of resource histogram based pruning and `diff2` pruning.

cumulative($[T_1, T_2, T_3]$,$[D_1, D_2, D_3]$,[1,1,2],2)
where
$T_1 :: \{0..1\}, D_1 :: \{4..5\}, T_2 :: \{1..3\}, D_2 :: \{4..7\}$,
$T_3 :: \{0..10\}, D_3 :: \{3..4\}$
After consistency checking $T_3 :: \{5..10\}$

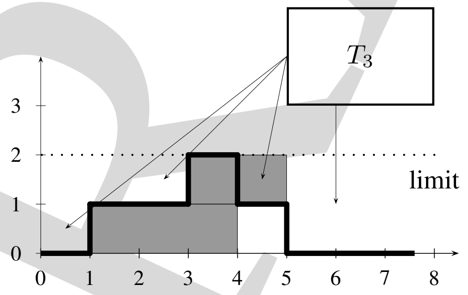
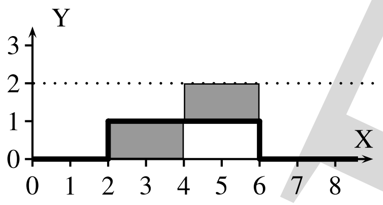
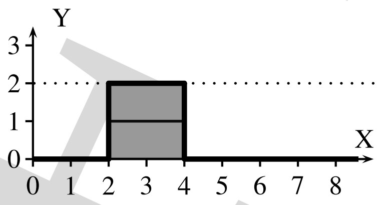


Figure 3.11: An example of the profile creation for the `cumulative` constraint.

Example 3.19: Consider a rectangle $r = [\{0..10\}, \{0..1\}, 2, 1]$ and two different situations presented in Figure 3.12. Figure 3.12 a) depicts two obligatory regions depicted as gray rectangles and the resource histogram depicted as a bold line. The resource histogram propagator will not be able to achieve any pruning since there is always one resource available for rectangle r but `diff2` propagator can achieve pruning and produces $r = [\{0..2, 4..10\}, \{0..1\}, 2, 1]$. Figure 3.12 b) presents two tasks (represented as rectangles) that have their started time 2 while their placement is not yet decided, i.e., it is represented as a domain $\{0..1\}$. In this case, `diff2` can not achieve any pruning since obligatory regions do not exist yet (the rectangles' positions in direction Y are still not decided). The resource histogram is depicted as the bold line which, when used for pruning, results in $r = [\{0, 4..10\}, \{0..1\}, 2, 1]$.



(a) obligatory regions pruning



(b) resource histogram pruning

Figure 3.12: Pruning examples for obligatory regions and resource histogram.

Energetic-based reasoning measures amount of "energy" required by tasks in a given interval and the amount of available energy. Energy is measured as product of number of required resources and execution time. The method compares the amount of resource energy required over time interval $[t_1, t_2)$ to the amount of energy that is available over the same interval. Let us first define the concept of *required energy consumption* $W(i, t_1, t_2)$ of a task i over an interval $[t_1, t_2)$ using the formula below.

$$W(i, t_1, t_2) = \min(C_i) \cdot \max(0, \min(D_i) - \max(0, t_1 - EST_i) - \max(0, LCT_i - t_2)) \tag{3.18}$$

where LCT is *latest completion time* defined as $LCT_i = \max(T_i) + \max(D_i)$ and EST is *earliest starting time* defined as $EST_i = \min(T_i)$.

Figure 3.13 illustrates situation of the required energy for task 1 ($T_1 = \{0..3\}$, $D_1 = 7$, $C_1 = 2$) over interval $[2, 7)$. The light gray rectangles represent the task execution span. Task 1 is depicted in the upper part of the figure. The lower part of the figure depicts the required energy consumption calculation. The left hand side gray rectangle represents part $\max(0, t_1 - T_i)$, i.e., $(2 - 0)$ and the right hand side part $\max(0, LCT_i - t_2)$, i.e., $(10 - 7)$. The final part of the task, depicted as middle gray area, has length 2 and hight 2 and therefore the required energy consumption is 4.

The first propagator examines whether task i can be run before j. If such a situation is not possible task j has to be executed before i and related pruning constraints can be added. To achieve this pruning the required energy consumption of all the tasks over the interval $[EST_i, LCT_j)$ is computed and compared to the energy available in the interval. The formula for this propagation rule is as follows.

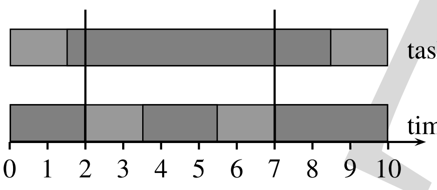


Figure 3.13: Required energy consumption of the task.

$$\max(\max(C_i), \max(C_j)) \cdot (LCT_j - EST_i) < \sum_{\substack{k \neq i \\ k \neq j}} W(k, EST_i, LCT_j) + \min(D_i) \cdot \min(C_i) + \min(D_j) \cdot \min(C_j) \quad (3.19)$$

If the formula 3.19 does not hold then there is not enough resources for execution tasks i before j and constraint $T_j + D_j \leq T_i$ is imposed.

There are proposed other propagators that use required energy consumption [2]. For example, one can choose a value x in the interval $[EST_i, LCT_i)$ and check whether task i can or cannot start at its earliest start time. If not the EST_i can be increased. Obviously a systematic strategy for selecting x is needed and there are proposals to do that [2].

The use of energetic rules for all pairs of $[EST_i, LCT_j)$ obviously requires either $\mathcal{O}(n^2)$ in space or $\mathcal{O}(n^3)$ in time [2].

Interval based propagation rules can find situations when task i has to be the last one or the first one in a given set $\mathcal{S}$ of tasks and update its T_i respectively. To simplify the discussion, we present the main idea of this algorithm for a case when $Limit = 1$. In the following, we extend our definitions of the last completion time and the earliest starting time for a set of tasks, $\mathcal{S}$, and denote it as $LCT_{\mathcal{S}}$ and $EST_{\mathcal{S}}$ respectively. Assume that we have a set of tasks $\mathcal{S}$, task i and $\mathcal{S}' = \mathcal{S} \setminus i$. Task i cannot be before all tasks in $\mathcal{S}'$ if

$$LCT_{\mathcal{S}'} - EST_i < \sum_{\mathcal{S}} \min(D_i) \quad (3.20)$$

and cannot be between all tasks in $\mathcal{S}'$ if

$$LCT_{\mathcal{S}'} - EST_{\mathcal{S}'} < \sum_{\mathcal{S}} \min(D_i) \quad (3.21)$$

Therefore, task i must be last and its start time can be narrowed according to T_i **in** $\{ECT_{\mathcal{S}'} .. \inf\}$.

Example 3.20: Consider three tasks with the following parameters: $T_1 :: \{0..11\}$, $D_1 = 6, C_1 = 1, T_2 :: \{1..7\}, D_2 = 4, C_2 = 1$ and $T_3 :: \{1..9\}, D_2 = 3, C_3 = 1$. The resource limit is 1. The illustration for the three tasks is depicted in Figure 3.14. Using rules 3.20 and 3.21 we can deduce that task 1 cannot be before tasks 2 and 3 because this would imply completion time for either task 2 or 3 equal 13 which is outside their allowed completion time. Therefore the earliest starting time for task 1 is adjusted to 8 (sum of durations of tasks 2 and 3). Using the similar reasoning on what tasks cannot be last the solver will also adjust the latest completion time for task 3 to 11 since the last task is task 1 and it start at latest at 11. The adjustments are represented as doted lines at respective intervals in Figure 3.14.

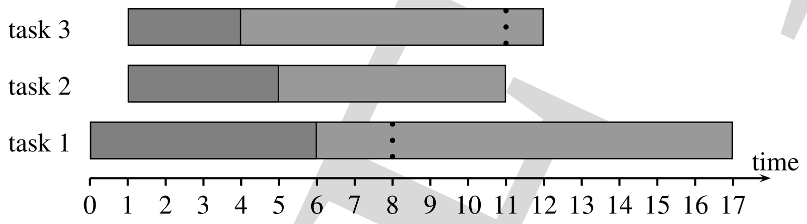


Figure 3.14: An illustration of the interval based propagation rules for three tasks and one resource.

A simple extension makes it possible to check whether task i cannot be before all tasks in $\mathcal{S}'$ if $Limit > 1$.

$$(LCT_{\mathcal{S}'} - EST_i) \cdot \max(limit) < \sum_{\mathcal{S}} \min(D_i) \cdot \min(R_i) \tag{3.22}$$

The technique that applies the discussed rules is known as *edge-finding* algorithm. This algorithm makes pruning of starting and completion times for each task. Notice that if n tasks require the resource, there are potentially $\mathcal{O}(n \cdot 2^n)$ pairs of $(i, \mathcal{S})$ to consider which obviously is not possible for practical implementation. Several researchers have proposed different algorithms that perform these adjustments in a reasonable time. The best algorithm is $\mathcal{O}(n \log n)$ but requires complex data structures [4, 16]. There exist several algorithms that has complexity $\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$ [4, 15, 57]. The algorithm with complexity $\mathcal{O}(n^3)$ has been proposed in [18]. It is based on *task intervals* and can be implemented in an incremental way. Therefore, it is difficult to determine its real efficiency in a solver.

The general idea of edge-finding algorithm with complexity $\mathcal{O}(n^2)$ has been suggested in [57]. This algorithm carries out two similar computations to find new EST and LCT for each task. We briefly discuss the first computation of this algorithm, which narrows EST of tasks. The second one, which narrows LCT's, is analogous. In this computation the algorithm finds all distinct LCT_k for all tasks and sorts them in decreasing order. Then, for each LCT_k the algorithm selects task i with $EST_i < LCT_k \wedge LCT_i > LCT_k$ and checks conditions (3.20) and (3.21). These conditions are checked for task i and set $\mathcal{S} = \{j | LCT_j \leq LCT_k\}$. Depending on the result of this checking a possible adjustment of task i EST is performed. It might decide that task i must be the last one or not the first one. Figure 3.15 presents this phase of the algorithm.

3.4.4 Element

The element constraint of the form `element` $(I,\ List,\ V)$ enforces a finite relation between I and V, $V = List[I]$. The vector of values, $List$, defines this finite relation.

```
for each unique $LCT_k \in \{lct|LCT(T_i)\}$ starting from max to min
  $\mathcal{S} = \{i|LCT_i \le LCT_k\}$,
  $\mathcal{R} = \{i|EST_i < LCT_k \wedge LCT_i > LCT_k\}$
while $\mathcal{R} \neq \emptyset \wedge S \neq \emptyset$
  select task $i$ with greatest $\min(D_i)$
  if before($i$, $\mathcal{S}$) $\wedge$ between($i$, $\mathcal{S}$)
    $\mathcal{S} = \mathcal{S} \setminus s$ with $EST_s = EST_\mathcal{S}$
  else
    $\mathcal{R} = \mathcal{R} \setminus i$
    if $\neg$ before($i$, $\mathcal{S}$) $\wedge$ $\neg$ between($i$, $\mathcal{S}$)
      // $i$ is after $\mathcal{S}$
      $T_i$ in $\{EST_\mathcal{S} + \sum_{s \in (S)} min(D_s)..\inf\}$
    else
      if $\neg$ before($i$, $\mathcal{S}$) $\wedge$ between($i$, $\mathcal{S}$)
        // $i$ cannot be before $\mathcal{S}$
        $T_i$ in $\{EST_\mathcal{S} + min(D_s)..\inf\}$
```

Figure 3.15: The general idea of the computation for finding new EST for tasks with one resource.

For example, the constraint

$$\texttt{element}(I, [3, 44, 10], V) \tag{3.23}$$

imposes the relation on index variable $I :: \{1..3\}$, and value variable $V :: \{3, 10, 44\}$. The change of one FDV propagates to another FDV. Imposing constraint $V < 44$ results in change of $I :: \{1, 3\}$. This constraint can be used, for example, to define discrete cost functions of one variable or a relation between task delay and its implementation resources.

The constraint can accept the list of integers only or can allow FDVs on this list. The second version of the constraint is more powerful but its implementation is more complex and introduces additional overhead.

Example 3.21: A task can be executed on one of five processors. The "cost" of execution on each processor for each task is different and is modeled by FDV c. Assume that FDV $p :: \{1..5\}$ defines the processor which is used for task execution. To assign the right "cost" of implementation to c depending on selected processor we define the following `element` constraint.

$$\texttt{element}(p, [3, 2, 2, 1, 4], c)$$

This constraint defines finite relation between selected processor p and its implementation cost c. Note if we decide to use processors $2, 3$ or 4 cost c will be pruned to $\{1..2\}$. On the other hand if we impose constraint on cost $c < 3$ then selection of p will be directly limited to $\{2..3\}$.

Consistency Algorithms

The constraint is simply implemented as a program which finds values allowed both for the first FDV and third FDV and updating them respectively. In the case when the list of FDVs is allowed the consistency algorithm has to take into account possible propagation to FDVs specified on the list of FDVs.

3.4.5 Distance

The distance constraint computes absolute value between two FDVs. The result is another FDV.

$$\texttt{distance}(X, Y, Dist) \tag{3.24}$$

Example 3.22: Consider that there are two memory operations accessing the same memory. These operations are executed in one clock cycle. The memory requires that all accesses follow certain timing specification. To be able to organize these accesses we have to schedule them in such a way that there are always two clock cycles between their accesses. The following constraint will solve this problem regardless of operations ordering.

$$\texttt{distance}(t_1, t_2, 3)$$

where t_1 and t_2 are memory access operations start times. Note, that the constraint requires three clock cycles distance between starting times of operations because we need to count current access which is one clock cycle.

Consistency Algorithms

Bounds consistency can be achieved by the following rules.

$$\begin{aligned}
&//Y - X = Z \\
&x_1 \leftarrow \min(Y) - \max(Z) .. \max(Y) - \min(Z) \\
&y_1 \leftarrow \min(X) + \min(Z) .. \max(X) + \max(Z) \\
&z_1 \leftarrow \min(Y) - \max(X) .. \max(Y) - \min(X) \\
&//X - Y = Z \\
&x_2 \leftarrow \min(Y) + \min(Z) .. \max(Y) + \max(Z) \\
&y_2 \leftarrow \min(X) - \max(Z) .. \max(X) - \min(Z) \\
&z_2 \leftarrow \min(X) - \max(Y) .. \max(X) - \min(Y)
\end{aligned} \tag{3.25}$$

$$\begin{aligned}
&X \ \textbf{in} \ \{x_1 \cup x_2\} \\
&Y \ \textbf{in} \ \{y_1 \cup y_2\} \\
&Z \ \textbf{in} \ \{z_1 \cup z_2\} \cap \{0 .. \inf\}
\end{aligned}$$

3.4.6 Circuit

The circuit constraint tries to enforce that FDVs which represent a directed graph will create a Hamiltonian circuit in this graph. The graph is represented by FDV domains in the following way. Nodes of the graph are numbered from 1 to N. A position in a list defines a node number. Each FDV domain represents direct successors of this node. For example, if FDV x at position 2 of the list has domain 1, 3, 4 then nodes 1, 3 and 4 are successors of node x. Finally, if i'th FDV of the list gets a value j then there is an arc from i to j.

$$\texttt{circuit}([X_1, \ldots, X_n]) \tag{3.26}$$

Example 3.23: Consider a directed graph depicted in Figure 3.16. It is defined by the following variables.

$$x_1 = 5, x_2 :: \{1, 5\}, x_3 = 2, x_4 :: \{2, 3\}, x_5 :: 4.$$

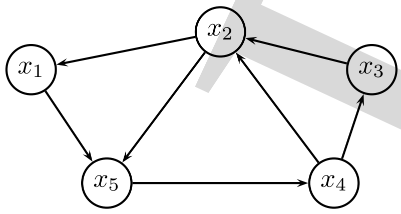


Figure 3.16: An example of `circuit` constraints.

Consistency Algorithms

The constraint can be considered as a specialization of `alldifferent` constraint. It requires that all FDVs must be different but, in addition, it needs an additional condition. This condition excludes circuits in the graph that do not contain all nodes. This is a so called subtour elimination or cut constraint known from ILP formulation of traveling salesperson problem (see section 4.2.7). Therefore `alldifferent` consistency rules can be applied in the first place. Then they need to be augmented with additional propagators.

In [20] authors propose simple rules that implement subtours elimination. The algorithm builds a list-like data structure that contains information on partial chains of nodes. It has information on first and last node as well as length of a subtour. The propagator examines last nodes in the chains and removes all next nodes candidates that can close a cycle and create a subtour which does not contain all nodes.

Example 3.24: Consider `circuit` constraint from Example 3.23 and assume that a partial tour has been built from node x_5 to x_2. This subtour is depicted in the figure with bold lines. To prevent construction of a shorter tour from node x_2 to node x_5 value 5 is removed by the propagation algorithm from variable x_2 (depicted as dotted line).

More advanced techniques can obviously make use of algorithms available for strongly connected components in graphs. Tarjan's algorithm [81] can find strongly connected components in $\mathcal{O}(|V| + |E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges. If the algorithm finds a largest strongly connected component of the length lower than number of nodes in the graph then there exist no solution to our constraint and the solver fails.

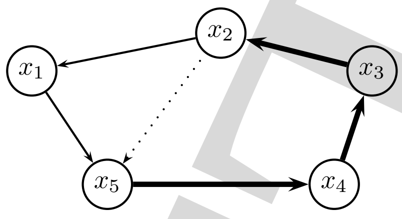


Figure 3.17: An example of subtour elimination.

3.4.7 Min and Max

These constraints enforce that a given FDV is minimal or maximal of all variables present on a defined list of FDVs.

$$\texttt{min}([X_0, \ldots, X_n], X_{min}) \tag{3.27}$$
$$\texttt{max}([X_0, \ldots, X_n], X_{max}) \tag{3.28}$$

Example 3.25: Consider that we schedule four tasks that have specified start times, t_i and completion times e_i by other constraints. Assume also that we would like to minimize the schedule length. It can be achieved by minimizing the latest completion time, *end*, of all tasks. This FDV which is used for minimization can be specified using `max` constraint as follows.

$\texttt{max}([e_1, e_2, e_3, e_4], end)$

Consistency Algorithms

A simple program can achieve consistency for this constraints. For example, `min` constraints can be implemented using the following consistency algorithm

```
for each $X_i$
  $X_i$ in {$\min(X_{min})$ .. inf}
$X_{min}$ in {$\min(\min(X_0), \ldots, \min(X_n))$ .. $\min(\max(X_0), \ldots, \max(X_n))$}
```

3.4.8 Sum and Weighted Sum

The sum constraints enforce that a sum of elements of FDVs' vector is equal to a given FDV `Sum`, that is $X_1 + X_2 + \cdots + X_n = Sum$.

$$\texttt{sum}([X_1, \ldots, X_n], Sum) \tag{3.29}$$

The weighted sum is provided by the constraint `SumWeight` and imposes the following constraint $W_1 \cdot X_1 + W_2 \cdot X_2 + \cdots + W_n \cdot X_n = Sum$.

$$\texttt{sumWeighted}([X_1, \ldots, X_n], [W_1, \ldots, W_n], Sum) \tag{3.30}$$

where X_i are FDVs and W_i integers representing weights.

These constraints are arithmetic constraints but they are included usually in many solvers to facilitate programming. The weighted sum constraint is used often to define weighted cost functions.

Consistency Algorithms

The consistency algorithms are trivial and basically implement bounds consistency as defined for arithmetical constraints.

3.4.9 Other Combinatorial Constraints

The constraints discussed in this chapter does not exhaust the list of constraints that are available in different constraints solvers. Usually, each solver offers arithmetic constraints and a number of specialized constraints that include combinatorial constraints. The list of available constraints differs however between different solvers. The implemented constraints depend on the application area and available resources for developing a particular constraint.

The list of constraints that might be useful for embedded system design but are not implemented in JaCoP solver can be made long. One can envision application of flow constraint that implements network flow problem [14]. Global cardinality (gcc) constraint has been already mention in subsection 3.4.1. Many problems that involve graph matching can be solved using graph pattern matching constraint [53]. One can also use specialized constraints that can be useful for particular problems encountered in the area of embedded system design. In [52] we have used, for example, a special version of `Diff2` constraints that makes it possible to define conditions when selected rectangles can overlap each other. This provided a way to model conditional resource sharing. Such solutions can be used easily adopted if the constraint solver is open and allows changes in its constraint implementations.

Questions and Exercises

1. Sudoku puzzle has been introduced in Questions and Exercises section in Chapter 2. Rewrite the constraints for this puzzle using different implementations of `alldifferent` constraints and compare the solving efficiency. The longer discussion on sudoku and possibility to solve it without search can be found in [82].

Four

Modeling with Constraints

Occam's Razor:
Do not assume more variables than necessary.

William of Occam (ca. 1285-1349)

4.1 Basic Modeling Techniques

Modeling with constraints is significantly different than other modeling techniques. We model the problem to be solved by defining constraints that need to be fulfilled in the final solution. We do not propose any algorithm for solving the problem. This is totally different than many other approaches offer. The modeling style is declarative and it is close to the modeling style of Integer Linear Programming (ILP). However, ILP is limited to inequalities over integers. It means that if exist non-linear constraints the model need to be linearized. Moreover, constraint programming offers a large set of constraints that can be used for modeling and the programmer has to select the right modeling style that models the problem in the best way. It is not an easy task since the problem can be modeled using different constraints.

To solve a problem using constraint programming one needs to define the following items that formalize the model and solving method:

1. *decision variables* and their domains,
2. *cost variable*, if optimization define,
3. *constraints* and possibly additional FDVs that represent the model, and
4. *search method*.

In this chapter we will discuss first three points and the search will be discussed in chapter 5.

Decision variables define important parameters or characteristics of the model that need to be determined. For example, for the graph coloring problem we define for each node a single FDV that defines the color of this node. The domain of this variable defines colors that are allowed for this particular node. When the model is solved each decision variable has assigned a single value that finally defines the selected color for the node. Selection of the domain for FDV is important since it restricts possible choices for the later steps that involve search.

Optimization problems require explicit definition of the optimization criteria. In constraint programming, it is represented as an additional FDV that defines the cost of a solution that need to be minimized. For example, if we want to minimize number of colors for graph coloring problem we can introduce a new FDV that is defined as a maximum of all FDVs defining the color of the nodes. Minimizing this variable will provide a solution that has a minimal number of colors used for coloring of a given graph.

Finally, the model need to define constraints that specify restrictions for the problem. In case of graph coloring it is quite simple. Two node connected by an edge cannot have the same color and therefore an inequality between two FDVs representing nodes connected by an edge are imposed. In this simple case we do not need additional variables. More complex models may require introduction of additional variables that are used to formalize relations between different parameters of the model.

The solver usually cannot find a solution to a given model using constraint consistency methods only. After imposing all constraints and enforcing constraint consistency of the model the search has to find a solution. Search methods are discussed in chapter 5.

4.2 Modeling of Selected Combinatorial Problems

In this section we define typical combinatorial problems using finite domain constraints. This illustrates principles of constraint programming based modeling and shows basic techniques to achieve standard models.

4.2.1 Bin Packing Problem

The bin packing problem is defined for a finite set $U = \{u_1, u_2, \ldots, u_n\}$ of items, a size $s(u) \in \mathbb{Z}^+$ for each $u \in U$, and a positive integer bin capacity B [32]. A solution to this problem is a partition of U into disjoint subsets $U_1, U_2, \ldots, U_k$ such that the sum of the items in each U_i is B or less. The objective is to pack the items from U in as few such bins as possible, i.e., minimize the number of disjoint subsets, k.

The problem can be modeled using several different constraints but the most natural and simple modeling uses `cumulative` constraint. In this formulation, T_i represents the bin number, D_i is the width of the bin which is always one, and C_i has value of the size of the item, i.e., $s(u_i)$. Minimization of the maximal T_i optimizes number of bins needed for the problem. The following constraints specify this modeling style.

$$\forall i : u_i :: 1..k$$
$$\texttt{cumulative}([u_1, u_2, \ldots, u_n], [1, 1, \ldots, 1], [s(u_1), s(u_2), \ldots, s(u_n)], B)$$
$$\texttt{max}([u_1, u_2, \ldots, u_n], Bin)$$

Minimization of FDV Bin provides minimal number of bins needed for packing the items. The other useful formulation is when we have a constant number of bins but we minimize the maximal size of a bin. This can be achieved by defining B as a FDV and minimizing it.

4.2.2 SAT Problem

The SAT problem is defined for a boolean formula containing n boolean variables. Given this formula find an assignment of some values to these variables so that the formula is true.

A logical formula can be represented in different ways. Here we consider a boolean formula in Conjunctive Normal Form (CNF), i.e., it is a conjunction of disjunctions. SAT problems having two boolean variables in each clause are called 2-SAT and problems with three variables are called 3-SAT problems. 2-SAT problems can be solved in polynomial time but 3-SAT problems does not have any known polynomial time algorithm. It has been historically found as the first NP-complete problem.

CNF formula φ over the variables $X = \{x_1, \ldots, x_n\}$ can be formally represented as follows.

$$\varphi = \bigwedge_{j=1}^{m} C_j = \bigwedge_{j=1}^{m} (\bigvee_{i \in P_j} x_i) \vee (\bigvee_{i \in N_j} \neg x_i)$$

where N_i and P_i are the sets of negated and unnegated variables in clause C_j respectively. A solution to SAT problem is a truth assignment to all x_i variables that satisfies all clauses C_j. We define this problem using the following constraints:

$$\begin{aligned} &\forall i : x_i :: 0..1, \\ &\forall j : \sum_{i \in P_j} x_i + \sum_{i \in N_j} (1 - x_i) \geq 1 \end{aligned} \tag{4.1}$$

An assignment to all x_i is a solution to SAT problem.

When all constraints cannot be satisfied, one might want to find a solution that has most constraints satisfied. This problem is called Max-SAT. Max-SAT problem finds an assignment to all x_i that satisfies a maximal number of clauses C_j. It can be defined using the following constraints:

$$\begin{aligned} &\forall i : x_i :: 0..1, \\ &\forall j : (\sum_{i \in P_j} x_i + \sum_{i \in N_j} (1 - x_i) \geq 1) \Leftrightarrow B_j \\ &cost = \sum_{j} B_j \end{aligned} \tag{4.2}$$

maximization of $cost$ will provide an optimal solution to the Max-SAT problem.

4.2.3 Graph Coloring

A vertex coloring is an assignment of colors to each vertex of a graph such that no edge connects two identically colored vertices. The most common type of vertex coloring seeks to minimize the number of colors for a given graph. The graph k-colorability decision problem for $k \geq 3$, i.e., the problem whether a graph can be colored with k colors, is NP-complete.

Given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, we make the following definitions to find the minimal graph coloring:

- each vertex v_i is assigned a FDV $x_i :: 0..N$, where N is maximum number of allowed colors

- $\forall(v_i, v_j) \in E : (x_i \neq x_j)$

The minimum number of colors can found by minimizing the following cost function: $cost = max(x_1, \ldots, x_n)$

4.2.4 Maximal clique

A clique of a graph is defined as a maximal completely connected subgraph, i.e., a subgraph which has every pair of vertices joined by an edge. The problem of finding the size of a clique for a given graph is an NP-complete problem.

Given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, we make the following definitions to find the clique of maximal cardinality:

- each vertex v_i is assigned a FDV $x_i :: 0..1$,
- $\forall(v_i, v_j) \notin E : (x_i \neq 1 \vee x_j \neq 1)$

The maximal clique can be found by maximizing the following cost function: $cost = \sum_i x_i$. Vertices which has $x_i = 1$ belong to maximal clique.

The above definition for maximal clique problem uses only primitive constraints and is not very efficient for large problems. Régin [77] presents more efficient methods for computing upper bounds, consistency algorithms for specialized constraints, and new search strategies for finding optimal solutions. He also presents experimental results that confirm efficiency of his algorithms.

4.2.5 Maximal independent set

Maximal independent set is a set of vertices in a graph such that for any pair of vertices, there is no edge between them and such that no more vertices can be added and it still be an independent set. It is NP-complete to decide if there is maximal independent set of size k in a graph.

Given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, we make the following definitions to find the maximal independent set of vertices (there are no edges between the vertices in this set):

- each vertex v_i is assigned a FDV $x_i :: 0..1$,
- $\forall(v_i, v_j) \in E : (x_i + x_j < 2)$

The maximal independent set can be found by maximizing the following cost function: $cost = \sum_i x_i$. Vertices which has $x_i = 1$ belong to maximal independent set.

4.2.6 Job-Shop Scheduling

The job-shop scheduling problem (JSSP) is a combinatorial optimization problem. It is defined by n jobs that need to be scheduled on m machines. Each job contains a sequence of m tasks that have defined their execution time and a machine for their execution. The tasks of each job are executed on different machines. It has been showed that non-preemptive JSSP is NP-complete problem, known to be among the most difficult ones [32, 4].

Given a set of n jobs, J, and m machines, each job i ($j_i \in J$) has m ordered tasks $task_{ik}, 1 \leq i \leq n, 1 \leq k \leq m$. Each task $task_{ik}$ has defined its duration d_{ik} and a machine for its execution r_{ik}. Moreover we define FDV t_{ik} that defines task starting time. The following constraints define JSSP:

- $\forall i : t_{ik} + d_{ik} \leq t_{ik+1}, 1 \leq k \leq m-1$
- $\forall k\ :\ cumulative([t_{1l_1}, t_{2l_2}, \dots, t_{nl_n}], [d_{1l_1}, d_{2l_2}, \dots, d_{nl_n},], [1, 1, \dots, 1], 1)$, $r_{il_i} = k$
- $max([t_{1m}, t_{2m}, \dots, t_{nm}], end)$

The first set of constraints defines precedence constraints between tasks of each job and m cumulative constraints enforce condition that executions of tasks on a given machine do not overlap. Finally the last constraint defines the makespan of the schedule. Minimization of variable end produces a schedule with the minimal execution time.

4.2.7 Traveling Salesperson Problem

Traveling salesperson problem is defined on weighted graphs that has weights assigned to edges. To solve the problem we need to find a path through this weighted graph which starts and ends at the same vertex, includes every other vertex exactly once, and minimizes the total cost of edges. The defined path through all vertices is called a Hamiltonian cycle or circuit, however a Hamiltonian cycle need not be optimal. While the problem of finding Hamiltonian circuit is NP-complete, the traveling salesperson problem is NP-hard.

Given a graph $G = (V, E)$ with $V = \{v_1, \dots, v_n\}$ and weights w_{ij} assigned to each edge $e_{ij} = (v_i, v_j)$, we make the following definitions to find the minimal route for traveling salesperson (typical CLP formulation):

- each vertex v_i is assigned a FD variable $x_i :: 0..N$, where N is the number of vertices,
- $circuit(x_1, \dots, x_N)$
- $\forall i : element(x_i, [w_{i1}, \dots, w_{i,N}], l_i)$

The minimum trip is found by minimizing the following cost function $cost = \sum_i l_i$.

ILP inspired formulation:

- for each edge e_{ij} we define $x_{ij} :: 0..1$
- in-degree constraints $\forall v_j \in V : \sum_{v_i \in V} x_{ij} = 1$
- out-degree constraints $\forall v_i \in V : \sum_{v_j \in V} x_{ij} = 1$
- strong connectivity constraints (often called *subtour elimination constraints* or *cut constraints*)

$$\forall S \subset V, S \neq \emptyset : \sum_{v_i \in S} \sum_{v_j \in V \setminus S} x_{ij} \geq 1 \tag{4.3}$$

The minimum trip is found by minimizing the following cost function.

$$cost = \sum_{v_i} \sum_{v_j} w_{ij} \cdot x_{ij} \tag{4.4}$$

The ILP formulation generates roughly $2^{50} > 10^{15}$ constraints for 50 cities.

4.3 Assignment and Scheduling

Most successful applications of constraint programming are in planning and scheduling. Since embedded systems design problems consider task (processes, jobs or operations) assignment and scheduling it is natural to use the developed methods for this purpose. We will use the term task in our discussions. It represents an atomic computation that need to be assigned to processing unit and scheduled. The task can be a simple operation, such as addition or multiplication, or a software processe that need to be executed on a processor or implemented by a hardware unit. In this section we are going to discuss a number of typical applications and modeling styles for solving these problems. The discussion is based on previous research in this area [48, 49, 38, 39, 87, 86, 50, 88, 52, 89, 51, 85].

We structure this presentation as follows. First we will discuss multiprocessor task assignment and scheduling for tasks without dependencies. Then we will concentrate on assignment and scheduling of *task graphs* represented as *data flow graphs* (DFG). These methods are applicable in high-level synthesis and system-level synthesis and we will use examples and more detail problems from these areas.

4.3.1 Scheduling For Tasks Without Dependency

Assume that n tasks need to be executed on a $k, k > 0$ execution resources, e.g., processors. Each task execution resource is denoted by $r_i, 1 \leq i \leq n$. The tasks are independent. They do not communicate and do not need to synchronize. We assume also that each task has execution time, d_i, that is the same on all execution units. This simplified assumption can later be removed and we will show how to solve a similar problem for tasks with different execution times. The problem is how to assign tasks to execution units that the completion time, cp, of all tasks will be minimized. This problem can be easily modeled as a version of the bin packing problem as defined in section 4.2.1.

The bin packing formulation is defined as follows. Each bin models a processing unit and each task is assigned unit r_i. The task is modeled as a rectangle of width 1 and hight d_i. All constraints for our problem are defined by one cumulative constraint of the form.

$$\texttt{cumulative}([r_1, r_2, \ldots, r_n], [1, 1, \ldots, 1], [d_1, d_2, \ldots, d_n], cp) \tag{4.5}$$

Minimization of FDV cp provides a solution with minimal completion time of all tasks. In practice, it is useful to compute a lower bound for such a solution since it can be used to stop search after finding a solution that has cp equal to the computed lower bound. Fortunately, the lower bound can easily be computed using the following formula.

$$lb = \Big\lceil \sum_{i=1}^{n} d_i / k \Big\rceil \tag{4.6}$$

Example 4.1: Assume that we want to assign ten independent tasks into three identical processors. The execution times for these tasks are defined in Table 4.1. The problem can be encoded using the cumulative constraint as defined in formula

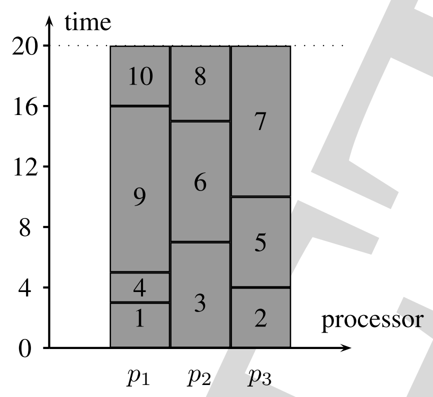


Figure 4.1: Assignment of tasks to three processors for Example 4.1

.

4.5. One of possible optimal solutions is depicted in Figure 4.1. For each processor, the order of task placement in the diagram is not relevant. The tasks are assigned to the processors but their execution order is not determined, i.e., they are not scheduled. Note that in this example the lower bound is equal the optimal solution.

Table 4.1: Execution times for tasks from Example 4.1

Task	1	2	3	4	5	6	7	8	9	10
Duration	3	4	7	2	6	8	10	5	11	4

We have assumed that each task has the same execution time on all execution units. This is not always the case in practice. Different processors can be used and their speed can vary quite much and therefore it is should exist a method to define different execution time for tasks depending on selected processor for it execution. The relation between task duration, d_i, and processor for it execution, r_i can be easily defined using `element` constraint, defined in section 3.4.4, in the following way:

$$\texttt{element}(r_i, [v_0, v_1, \ldots, v_n], d_i) \tag{4.7}$$

where $v_k, 0 \leq k \leq n$ is an integer value specifying duration of task i on processor k.

Example 4.2: Assume that nine tasks need to be assigned to three different processors. The tasks have often different execution times on different processors. Table 4.2 specifies execution time for each task on each processor. Note, that some tasks cannot be executed on some processors. For example, task eight cannot be executed on processor p_1 which is indicated by symbol "-" in the table.

The constraints model the task assignment problem for data specified in Table 4.2.

Table 4.2: Execution times for tasks on different processors for Example 4.2

Task	1	2	3	4	5	6	7	8	9
Duration p_1	2	2	1	1	1	1	3	-	1
Duration p_2	3	1	1	3	1	2	1	2	1
Duration p_3	1	1	2	-	3	1	4	1	4

```
$r_1 :: \{1..3\}$, $r_2 :: \{1..3\}$, $r_3 :: \{1..3\}$, $r_4 :: \{1..2\}$, $r_5 :: \{1..3\}$,
$r_6 :: \{1..3\}$, $r_7 :: \{1..3\}$, $r_8 :: \{2..3\}$, $r_9 :: \{1..3\}$,

impose element($r_1$, [2, 3, 1], $d_1$),
impose element($r_2$, [2, 1, 1], $d_2$),
impose element($r_3$, [1, 1, 2], $d_3$),
impose element($r_4$, [1, 3], $d_4$),
impose element($r_5$, [1, 1, 3], $d_5$),
impose element($r_6$, [1, 2, 1], $d_6$),
impose element($r_7$, [3, 1, 4], $d_7$),
impose element($r_8$, [0, 2, 1], $d_8$),
impose element($r_9$, [1, 1, 4], $d_1$),

$cp :: \{1..100\}$
impose cumulative([$r_1, r_2, \ldots, r_9$], [$1, 1, \ldots, 1$], [$d_1, d_2, \ldots, d_n$], $cp$)
```

Note that domains of variables r_4 and r_8 does not contain all values at positions $1, 2$ and 3 since only two processors can be used for their execution. Since `element` constraint requires specification of values starting from index 1 the related execution time value for task 8 is set to 0 but it will never be assigned because of index ranges from 2 to 3. The optimal solution for this example has the total execution time equal three and all tasks are assigned to processors in such a way that their execution time equals to one. One solution assigns tasks 3, 4 and 9 to processor 1, tasks 2, 5 and 7 to processor 2, and tasks 1, 6 and 8 to processor 3.

4.3.2 Scheduling For Task Graphs

The system is represented as a *task graph*. An example of such graph is depicted in Figure 4.2. Each node in this graph corresponds to a task and an edge connects two nodes if and only if there exist at least one direct communication between the corresponding tasks. The task activates communication to another task, at the end of its execution, if there is an edge in the task graph connecting a sender task node with the receiver task node. For example, task p_1 communicates with task p_3 since there is the edge between nodes p_1 and p_3.

A task, represented by a node, is activated when a communication on all its active inputs, modeled by edges, took place. The task graph is acyclic but an implicit loop between output nodes and input nodes is assumed. In the task graph, depicted in Figure 4.2, for example, after execution of tasks p_6 and p_7, tasks p_1 and p_2 are activated again. The execution can thus be repeated infinitely.

Each task has a deterministic execution time but it can be different on different execution units. A communication time is also deterministic and possibly different

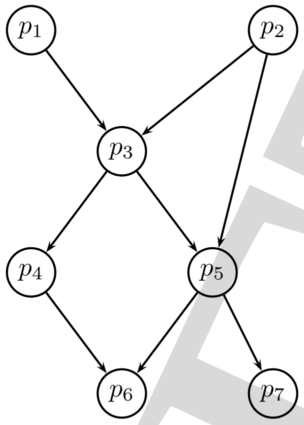


Figure 4.2: Task graph example.

for different communication devices. In particular, we assume that the communication between two tasks assigned to the same resource, either a processor or or an ASIC, does not require any time.

Communication and synchronization between tasks is allowed only at the beginning or at the end of the task execution, as indicated by the task graph. Many embedded systems fit into this model. For example, DSP program specifications based on synchronous data-flow model have these properties. If this restriction is too strong and it cannot be fulfilled by the system specification, the specification has to be partitioned into a number of smaller tasks. After this pre-partitioning communication channels need to be established to provide communication facilities between newly created tasks.

The goal of the embedded system synthesis is to find an assignment of tasks into processors/ASICs and communication channels into communication resources (e.g., buses, point-to-point links) which fulfills given timing constraints and minimizes system resources.

We will concentrate on modeling issues for a given task graph and do not consider in this discussion techniques for program or graph transformations, such us loop unfolding.

To model task assignment and scheduling we need to define a number of constraints that can be grouped in different categories. Each task graph defines a number of task precedence constraints that define order of execution of tasks. There exist constraint on resource usage. For example, two tasks cannot be executed simultaneously on one processor. In this discussion we will first define task precedence constraints and then discuss resource constraints.

In this section we continue to use the same notation for FDVs, i.e., execution resource is denoted by r_i and task duration by d_i. In addition, we assume that each task has defined its starting time t_i. Constraints between these FDVs will define basic requirements for task assignment and scheduling.

Task Precedence

The precedence relation imposed by a partial order of task graph nodes is modeled by inequality constraints. If there exists an arc from task i to task j then $t_i + d_i \leq t_j$ inequality constraint is imposed. This inequality defines *precedence constraints*. In general, we have the following code for imposing these constraints.

```
for $(i, j) \in$ Dependencies                                    (4.8)
  impose $t_i + d_i \leq t_j$
```

A task delay d_i, as indicated in the previous section, can be an integer or FDV.

Example 4.3: Consider the task graph depicted in Figure 4.2. The following precedence constraints need to be defined.

$t_1 + d_1 \leq t_3$, $t_2 + d_2 \leq t_3$, $t_2 + d_2 \leq t_5$, $t_3 + d_3 \leq t_4$,
$t_3 + d_3 \leq t_5$, $t_4 + d_4 \leq t_6$, $t_5 + d_5 \leq t_6$, $t_5 + d_5 \leq t_7$.

Assume a definition of all $t_i :: \{0..10\}$ and all $d_i = 1$. For this definition the solver automatically updates all t_i to the following values.

$t_1 :: \{0..7\}$, $t_2 :: \{0..7\}$, $t_3 :: \{1..8\}$, $t_4 :: \{2..9\}$,
$t_5 :: \{2..9\}$, $t_6 :: \{3..10\}$, $t_7 :: \{3..10\}$.

Equation 4.8 defines perfectly all precedence constraints specified in the task graph. In some situations, however, one can define additional constraints that improve efficiency of the solver. This is typically information that cannot be discovered by the solver form imposed constraints and has to be explicitly added to the model. Consider, for example, that the task graph depicted in Figure 4.2 is supposed to be assigned and scheduled on a single computational unit, for example processor. In this case task p_6 has to be schedule after execution of both tasks p_4 and p_5. It does not depend of the order of execution of both tasks but they need to be executed before task p_6 and therefore its precedence constraint can extended to constraint $t_3 + 3 \leq t_6$, i.e., earliest starting time for task p_6 is the time when we have executed task p_3 and both tasks p_4 and p_5 ($d_3 + d_4 + d_5 = 3$). The similar reasoning can be applied to p_2 and p_6. This kind of constraints can be added to the model based on the analysis of the problem. They can be treated as redundant constraints but needed for the solver. CHIP system [23] developed for such cases a general combinatorial constraint `precedence` that handles such situations.

Example 4.4: Consider Example 4.3 and assume that the task graph is executed on a single resource. Constraints $t_3+3 \leq t_6$ will improve efficiency of propagation and we will get the following domains of FDVs after consistency.

$t_1 :: \{0..6\}$, $t_2 :: \{0..6\}$, $t_3 :: \{1..7\}$, $t_4 :: \{2..9\}$,
$t_5 :: \{2..9\}$, $t_6 :: \{4..10\}$, $t_7 :: \{3..10\}$.

In the new formulation the start time of task p_6 has been updated to start earliest at time 4 that is correct for one resource execution.

The additional precedence constraints has been used to improve the efficiency of solvers for their models. Authors of [92], for example, use ordinary precedence constraints together with these kind of constraints that they call *distance constraints*. They also use `allfifferent` constraints, based on bounds consistency, to impose resource constraints. They obtained very good results for scheduling instructions for single-issue processors. They applied their method to SPEC95 floating point benchmarks. All 7402 of the benchmarks' basic-blocks were optimally scheduled, including basic-blocks with up to 1000 instructions. The compile time increased by 0.6% only. They also observed that for a number of "difficult" scheduling problems it was possible to obtain optimal results only when distance constraints have been used.

Resource Sharing

Resource constraints are used to define basic restrictions on the use of resources. We usually consider computational resources, such as processors or computational units, but one can consider other resources such as power. The resource constraints define limitation on simultaneous use of these resource. For example, the application cannot use more processors than is available. The most common resource constraint is the constraint on *resource sharing* that defines rules for reusing the resource.

The domain of variable r_i specifies possible implementation resources for a given task. Tasks can usually share a resource if their executions do not overlap which introduces additional constraints on resources sharing. These constraints, called *resource constraints*, prohibit simultaneous use of resources and can be specified using disjunctive constraints defined as follows.

```
for $(i, j)$ where $i \neq j$                                    (4.9)
  impose $t_i + d_i \leq t_j \vee t_j + d_j \leq t_i \vee r_i \neq r_j$
```

These constraints can also be defined using reified constraints as follows.

```
for $(i, j)$ where $i \neq j$                                    (4.10)
  impose $t_i + d_i \leq t_j \Leftrightarrow b_1 \wedge t_j + d_j \leq t_i \Leftrightarrow b_2 \wedge r_i \neq r_j \Leftrightarrow b_3$
  impose $b_1 + b_2 + b_3 > 0$
```

Finally, the combinatorial constraint `diff2` can also be used for this purpose. In this case, we specify all tasks that might share the same resource in one `diff2` constraint as defined below.

$$\texttt{diff2}([[t_1, r_1, d_1, 1], [t_2, r_2, d_2, 1], \ldots, [t_n, r_n, d_n, 1]]) \tag{4.11}$$

Efficiency of propagation can often be improved by adding *implied constraints*, i.e., constraints that do not impose new requirements but help a solver to find situations where more efficient pruning of some FDVs can be done. Though adding an implied constraint to the specification of a problem does not change the set of solutions, it can reduce the amount of search the solver has to do. The `cumulative` constraint can be used for this purpose since it has global view on all involved tasks and uses different consistency methods than `diff2` constraint. This constraint is usually defined as follows.

$$\texttt{cumulative}([t_1, \ldots, t_n], [d_1, \ldots, d_n], [1, \ldots, 1], ResourceLimit) \tag{4.12}$$

The modeling style has significant impact on the solver performance. One can expect that use of combinatorial constraints improves the performance since their related pruning algorithms have more look-ahead principles. A simple comparison of the assignment and scheduling results using different modeling styles has been presented in [51]. This study have explored four modeling styles of the fifth order elliptic filter (well-known benchmark for high-level synthesis). For the simple case of two adders and two multipliers the optimal solution has been achieved for all modeling styles. However, the solver did not need any backtracking to generate the optimal solutions and prove their optimality for the cases, when combinatorial constraints (`diff2` and `cumulative`), has been used. In other cases, number of backtracks was between 14,000 and 20,000.

Example 4.5: Consider a data-flow graph of differential equation solver used often as an example in HLS and depicted in Figure 4.3. It defines a computation that has 11 operations. They can be implemented on two types of functional units, adders and multipliers. Assume that addition has duration of one clock cycle and multiplication two clock cycles. We assume that two adders, numbered 1 and 2, and two multipliers, numbered 1 and 2, are used in this example. This information is used to define domains for FDVs r_i. The following constraints specify both precedence and resource constraints.

```
for $i = 1..11$
  $t_i :: \{0..100\}$

int n=2, m=2;  // number of adders and multipliers
$r_5 :: \{1..n\}$, $r_8 :: \{1..n\}$, $r_9 :: \{1..n\}$, $r_{10} :: \{1..n\}$, $r_{11} :: \{1..n\}$,
$r_1 :: \{1..m\}$, $r_2 :: \{1..m\}$, $r_3 :: \{1..m\}$, $r_4 :: \{1..m\}$, $r_6 :: \{1..m\}$,
$r_7 :: \{1..m\}$,

// precedence constraints
impose $t_1 + 2 \leq t_6$, impose $t_2 + 2 \leq t_6$, impose $t_3 + 2 \leq t_7$,
impose $t_4 + 2 \leq t_8$, impose $t_5 + 1 \leq t_9$, impose $t_6 + 2 \leq t_{10}$,
impose $t_7 + 2 \leq t_{11}$, impose $t_{10} + 1 \leq t_{11}$,

// resource constraints for adders
impose diff2($[[t_5, r_5, 1, 1], [t_8, r_8, 1, 1], [t_9, r_9, 1, 1],$
             $[t_{10}, r_{10}, 1, 1], [t_{11}, r_{11}, 1, 1]]$),
// resource constraints for multipliers
impose diff2($[[t_1, r_1, 2, 1], [t_2, r_2, 2, 1], [t_3, r_3, 2, 1],$
             $[t_4, r_4, 2, 1], [t_6, r_6, 2, 1], [t_7, r_7, 2, 1]]$).
```

Resource constraints can be defined using a single `diff2` constraint if we use conseuitive values for resource numbers, for example adders $\{1..2\}$ and multipliers $\{3..4\}$.

The solver enforces consistency of the imposed constraints and adjusts the domains of defined FDVs. The domains of variables r_i does not change in this stage but start time varibales are adjusted as follows.

$t_1 :: \{0..94\}, t_2 :: \{0..94\}, t_3 :: \{0..95\}, t_4 :: \{0..97\}, t_5 :: \{0..98\}, t_6 :: \{2..96\},$
$t_7 :: \{2..97\}, t_8 :: \{2..99\}, t_9 :: \{1..99\}, t_{10} :: \{4..98\}, t_{11} :: \{5..99\}$.

Note, that the effect of consistency checking is similar to computing ASAP and ALAP schedules and determining operations mobility.

The above formulation with precedence and resource constraints can be used to define different types of optimization problems that define different types of schedul-

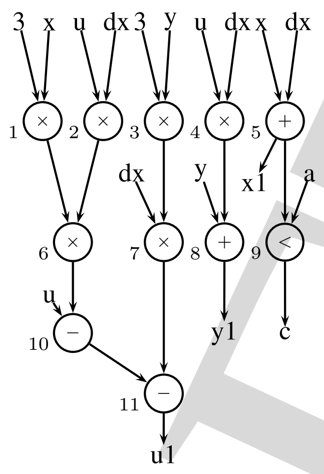


Figure 4.3: An example of data-flow graph for differential equation solver.

ing. Two types of scheduling methods are usually distinguished. *Resource-constraint scheduling* takes the specification of all constraints, assumes the limit on the number of resources (e.g., adders, multipliers) and optimizes the completion time of the whole application. *Time-constraint scheduling*, on the other hand, assumes the same constraints but optimizes the number of resources given a limit on the execution time of the application. Traditionally, both scheduling techniques cannot be solved using the same algorithms and specific methods for each type of scheduling need to be developed. Resource-constrained scheduling can be solved, for example, using *list scheduling* [28] while resource-constrained using *force-directed scheduling* [67]. Constraint programming formulation can be used for both types of scheduling policies.

Example 4.6: Consider constraints for assignment and scheduling of differential equation solver as defined in Example 4.5. First, we will define optimization method to solve resource-constrained scheduling. To do that we need to define a cost function and an optimization method. Our cost function is defined as the completion time of the whole application, i.e., the maximum time when the last operation finishes its execution. The whole method can be defined as follows (assuming minimization procedure defined in section 5.13).

impose $e_8 = t_8 + 1$, **impose** $e_9 = t_9 + 1$, **impose** $e_{11} = t_{11} + 1$,
$v \leftarrow [[t_0, r_0], [t_1, r_1], \ldots, [t_{11}, r11]]$,
impose max$(completionTime, [e_8, e_9, e_{11}])$,
minimize$(v, completionTime)$

The solver finds an optimal assignment and schedule for two adders and two multipliers that is seven clock cycles long.

Now, using the same constraints we define time-constraint scheduling. Our cost function is changed and it is simply defined as a number of adders and multipliers used by our application. The constraints below define time-constraint scheduling.

impose max$(numberAdders, [r_5, r_8, r_9, r_{10}, r_{11}])$,

```
impose max($numberMuls, [r_1, r_2, r_3, r_4, r_6, r_7]$),
impose $numberResources = numberAdders + numberMuls$,
minimize($v, numberResources$).
```

In this case, if we constraint the completion time to 13 clock cycles, the solver finds an optimal solution that has cost function equal two which represents the assignment and schedule with one adder and one multiplier.

Modeling assignment and scheduling problems with constraint has big advantage because one can easily add additional constraints to enforce particular conditions or requirements.

Relative timing constraints are used in high-level synthesis to impose relations on start times between pairs of nodes [47]. One used to define minimal and maximal time separation between starting times of operations. They can easily be defined in constraint programming framework by adding additional inequality constraints. No additional extension to task graphs or data-flow graphs and related algorithms is required. Following the definitions of [47] we can define the minimum and maximum timing constraints between operations i and j as follows.

- a minimun timing constraint $l_{ij} \geq 0$ requires that:
  $t_j \geq t_i + l_{ij}$
- a maximum timing constraint $u_{ij} \geq 0$ requires that:
  $t_j \leq t_i + u_{ij}$

Example 4.7: Example 4.5 can be extended with timing constraints. For example, one can require that $t_9 \geq t_5 + 4$. All assignments and schedules generated for these extended constraints have to obey the new requirement. For example, a valid solution will have $t_5 = 0$ and $t_9 = 4$.

Functional Pipelining

Pipelining a task graph is an efficient way of accelerating a design. In computer architecture, two-dimensional reservation tables are usually used for pipeline analysis [43]. Our approach for modeling resource sharing requirements with `diff2` constraint is similar to this approach but for pipeline modeling we extend it with several copies of rectangles representing subsequent pipeline computations. For an n stage pipeline of an initiation rate of k time units we add n copies of rectangles representing the tasks. The subsequent copies of rectangles (subsequent computations of tasks) start at positions k, $2 \cdot k$, $3 \cdot k$, and so on. This prevents placing tasks in forbidden locations, which are to be used by subsequent pipeline computations. The copies of rectangles do not define final task positions but these positions will be adjusted during assignments of values to original FDVs representing tasks' start time.

The following pseudo-code defines, for example, a two-stage pipeline with initiation rate k:

```
for each $i$
  impose $t1_i = t_i + k$
  impose diff2($[[t_1, r_1, d_1, 1], \ldots, [t_n, r_n, d_n, 1],$
               $[t1_1, r_1, d_1, 1], \ldots, [t1_n, r_n, d_n, 1]]$)
```

We also assume a lower bound on the initiation rate for the pipeline. We compute it separately for each resource and take the maximum over all resources, as defined below.

$$\begin{aligned} &\forall i : lb_i = \lceil (n_i \cdot d_i)/m_i \rceil \\ &lb = \max_i (lb_i) \end{aligned} \tag{4.13}$$

where n_i denotes number of operations of type i and m_i denotes number of resources of type i. d_i is operation's delay as defined before.

Example 4.8: In this example, we will define pipeline constraints for the differential equation solver from Example 4.5. Both definition of resource FDVs r_i and start timer FDVs t_i are the same. We add new start time FDVs for consecutive executions of the application and call them $t1_i$ and $t2_i$. Related relations between t_i and these variables are also established using FDV k that defines pipeline initiation rate. Additional resource constraints are also added in `diff2` constraints both for adders and multipliers.

```
$k :: \{0..100\}$
for $i = 1..11$
  $t_i :: \{0..100\}$
  $t1_i :: \{0..100\}$
  impose $t1_i = t_i + k$
  $t2_i :: \{0..100\}$
  impose $t2_i = t_i + 2 \cdot k$

// $r_i$ FDVs defined as previously
// precedence constraints as previously

// resource constraints for adders
impose diff2($[[t_5, r_5, 1, 1], [t_8, r_8, 1, 1], [t_9, r_9, 1, 1],$
             $[t_{10}, r_{10}, 1, 1], [t_{11}, r_{11}, 1, 1],$

             $[t1_5, r_5, 1, 1], [t1_8, r_8, 1, 1], [t1_9, r_9, 1, 1],$
             $[t1_{10}, r_{10}, 1, 1], [t1_{11}, r_{11}, 1, 1],$

             $[t2_5, r_5, 1, 1], [t2_8, r_8, 1, 1], [t2_9, r_9, 1, 1],$
             $[t2_{10}, r_{10}, 1, 1], [t2_{11}, r_{11}, 1, 1]]$),

// resource constraints for multipliers
impose diff2($[[t_1, r_1, 2, 1], [t_2, r_2, 2, 1], [t_3, r_3, 2, 1],$
             $[t_4, r_4, 2, 1], [t_6, r_6, 2, 1], [t_7, r_7, 2, 1],$

             $[t1_1, r_1, 2, 1], [t1_2, r_2, 2, 1], [t1_3, r_3, 2, 1],$
             $[t1_4, r_4, 2, 1], [t1_6, r_6, 2, 1], [t1_7, r_7, 2, 1],$

             $[t2_1, r_1, 2, 1], [t2_2, r_2, 2, 1], [t2_3, r_3, 2, 1],$
             $[t2_4, r_4, 2, 1], [t2_6, r_6, 2, 1], [t2_7, r_7, 2, 1]]$).
```

The lower bound for the pipeline initiation rate is six and the solver finds an implementation with this initiation rate. This is improvement by one clock cycle comparing to non-pipeline implementation.

Task Communication

Tasks connected by an arc in a task graph communicate by sending messages. This communication uses a communication resource (e.g., a point-to-point link or a bus) and is modeled as another communication task. The precedence and resource constraints are defined in the same way as discussed for computational tasks. Task communications require however additional constraints to consider local communications. The local communications between tasks is the communication that takes place between tasks assigned to the same processor. These tasks communicate through a shared memory and do not need to use communication devices. The communication task "disappears" and related communication time is usually assumed to be zero. In this case, $d_i = 0$ that influences both `diff2` and `cumulative` constraints since one of the rectangle or task durations become zero. This is not allowed in all solvers and additional measures need to be applied. For solvers that allow zero sizes in `diff2` and `cumulative` constraints this is not needed.

Consider an example where task i communicates with task j. The following constraints are defined for this case.

$$\begin{aligned} &\textbf{impose } r_i = r_j \Leftrightarrow d_{i,j} = 0 \\ &\textbf{impose } r_i \neq r_j \Leftrightarrow d_{i,j} = v_{i,j} \end{aligned} \tag{4.14}$$

where r_i and r_j have the same meaning as before, $d_{i,j}$ is FDV for communication time between tasks i and j and $v_{i,j}$ specifies communication time between tasks when assigned to different communication resources. Note that $v_{i,j}$ can be another FDV produced by `element` constraint since the communication time is different on different communication resources.

Example 4.9: Consider a task graph example depicted in Figure 4.4 from [69]. This task graph can be executed on a number of processors and execution times for each task on each processor are specified in Table 4.2. The processors have different cost. Processor of type 1 costs 4, type 2 costs 5 and type 3 costs 2. We assume also a bus communication structure. All non-local communications are assigned and scheduled on this bus. The communication time is assumed to take one time unit. In our model we assume that we can run our application on 6 processors; two of each type. Our model makes it possible to run both time-constrained and cost-constrained optimizations.

The following pseudo-code gives a sketch of the constraints defined to describe the problem.

```
// precedence constraints
impose $t_1 + d_1 \leq t_{1,3}$
impose $t_{1,3} + d_{1,3} \leq t_3$
...
// computational resources constraints
impose diff2($[[t_1, r_1, d_1, 1], \ldots, [t_9, r_9, d_9, 1]]$)
// bus constraints
```

```
impose diff2($[[t_{1,3}, r_{1,3}, d_{1,3}, 1], \ldots, [t_{6,9}, r_{6,9}, d_{6,9}, 1]]$)
// communication time constraints
impose $r_1 = r_3 \Leftrightarrow d_{1,3} = 0$
impose $r_1 \neq r_3 \Leftrightarrow d_{1,3} = 1$
...
// processor cost constraints
impose $(r_1 = 1 \vee r_2 = 1 \vee \cdots \vee r_9 = 1) \Leftrightarrow C_1$
...
impose $(r_1 = 6 \vee r_2 = 6 \vee \cdots \vee r_9 = 6) \Leftrightarrow C_6$
impose $Cost = 4 \cdot C_1 + 4 \cdot C_2 + 5 \cdot C_3 + 5 \cdot C_4 + 2 \cdot C_5 + 2 \cdot C_6$
// completion time constraints
impose $e_6 = t_6 + d_6$
impose $e_7 = t_7 + d_7$
impose $e_8 = t_8 + d_8$
impose $e_9 = t_9 + d_9$
impose $max(end, [e_6, e_7, e_8, e_9])$
```

The model has been used to generate different task assignments and schedules. The results are summarized in Table 4.3. Runtime has been measured at 2 GHz IBM laptop with 0.5 GB RAM using Java 1.5.

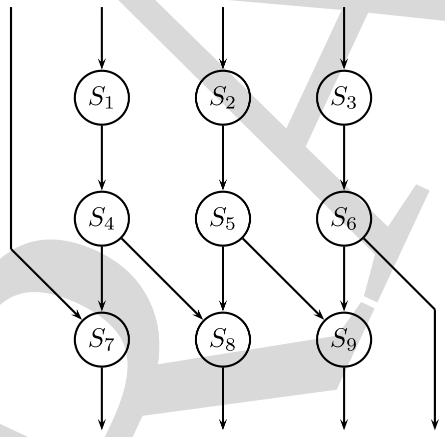


Figure 4.4: The task graph for Example 4.9.

Table 4.3: Scheduling Results for the System from Example 4.9

			Performance optimization		Cost optimization	
Architecture	Cost	Performance (time units)	Back-tracks	Runtime (s)	Backt-tracks	Runtime (s)
	10	6	117	0.39	285	0.22
bus	6	7	696	0.81	717	0.72
	5	15	61	0.11	66	0.12

Register Assignment

In HLS, a register needs to be assigned to all input/output and temporary variables. Since registers can be shared, a careful lifetime analysis of variables is needed to determine register assignment. This analysis determines a time interval when the variable is valid and needs to occupy a register. Register assignment is determined based on this analysis.

The time intervals for variables can be defined using constraints. Each time interval of a variable starts at time $t_i + d_i$, i.e., when operation i computed the value and written it into a register. The variable is valid until the last operation j used the value from the register at time $t_j + d_j$. Note, that if several operations uses a value then the last operation j defines this value. This can be easily defined in constraint programming using basic arithmetic and `max` constraints. Furthermore the lifetimes of variables can be modeled using rectangles that span on time axes during defined by constraints define-use time of variables. Defining the lifetimes of variables as rectangles provides a natural way to use both `diff2` and `cumulative` constraints.

The idea is depicted in Figure 4.5. Operation o_i produces data consumed by both operation o_j and o_k. The date are therefore valid from the time when operation o_i finishes its execution until it is consumed by operations o_j and o_k. Since the last operation is o_k the rectangle starts at $t_i + d_i$ and finishes at $t_k + d_k$.

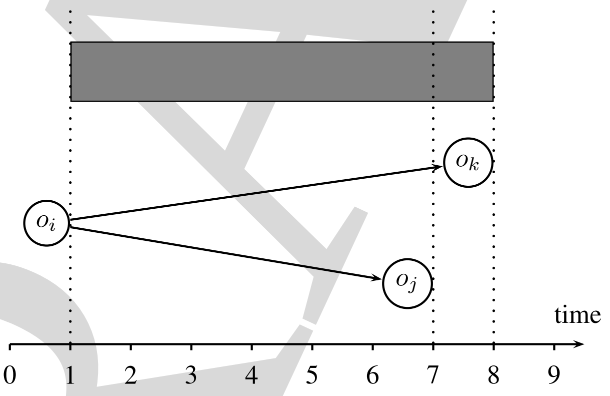


Figure 4.5: The idea of modeling define-use time for register assignment using constraints.

Example 4.10: Consider again Example 4.5. The registers has to be assigned to each arc in the data flow graph that defines data transfers. The registers need to be assigned for input and output data as well as intermediate data. The following additional constraints define rectangles for `diff2` constraint for register assignment.

```
for $i = 1..11$
  $e_i :: \{0..100\}$
  impose $e_i = t_i + d_i$

impose max($end_{1,3}, [e_1, e_3]$)
impose max($end_{1,5}, [e_1, e_5]$)
impose max($end_{2,4,10}, [e_2, e_4, e_{10}]$))
impose max($end_{2,4,5,7}, [e_2, e_4, e_5, e_7]$))
impose max($end_{3,8}, [e_3, e_8]$))

                  // input registers
rectangles = [$[0, m_3, end_{1,3}, 1], [0, m_x, end_{1,5}, 1]$,
              $[0, m_u, end_{2,4,10}, 1], [0, m_{dx}, end_{2,4,5,7}, 1]$,
              $[0, m_y, end_{3,8}, 1], [0, m_a, e_9, 1]$,
              // output registers
              $[e_5, m_{x1}, 100, 1], [e_8, m_{y1}, 100, 1], [e_9, m_c, 100, 1]$,
              $[e_{11}, m_{u1}, 100, 1]$,
              // temporal registers
              $[e_1, m_{1,6}, e_6 - e_1, 1], [e_2, m_{2,6}, e_6 - e_2, 1]$,
              $[e_3, m_{3,7}, e_7 - e_3, 1], [e_4, m_{4,8}, e_8 - e_4, 1]$,
              $[e_5, m_{5,9}, e_9 - e_5, 1], [e_6, m_{6,10}, e_{10} - e_6, 1]$,
              $[e_7, m_{7,11}, e_{11} - e_7, 1], [e_{10}, m_{10,11}, e_{11} - e_{10}, 1]]$,
```

Additional variable m_i has been defined to denote assigned register number. The subscript indicates the source or the destination of data for input and output registers. Intermediate registers has two subscripts indicating the source operation and the destination operation. For example, m_x denotes a register number assigned for input data x and $m_{1,6}$ defines a register number for intermediate data produced by operation 1 and consumed by operation 6. We also assumed that the schedule length is maximally 100 cycles and therefore output data must be valid until 100 cycles. When register assignment is applied after assignment and scheduling of operations the optimal number of assigned registers is seven.

Memory Constraints

The problems of assigning tasks to processors and communication to communication resources and their scheduling has been discussed in section 4.3.2. There exist, however, other constraints that may influence task assignment and scheduling. One of the most important issues, that is extensively studied, is memory. There are different types of problems related to memory. Data allocation is one of the most important issue since it influences final system performance. Different types of memories might also require special considerations. SDRAM, for example, need to be considered very carefully since the access time depends very much of location of data.

In this section, we will discuss an extended case of task assignment and scheduling with additional memory constraints. It is suppose to illustrate opportunity of imposing additional constraints that need to be consider. Since tasks need memory to store their code and data we will discuss code memory and data memory constraints. We also assume that our system contains a number of processors that are equipped with local memory. Tasks use this memory for computation and communication and there exist no global memory. These assumptions are used in this section to concentrate on typical solutions but more general assumptions can also be encapsulated in our model. More discussion on SDRAM memories and their formalization with constraints can be, for example, found in [85, 84].

Code memory is used to store programs implementing tasks. The amount of code memory needed to implement a task depends on the processor type, but it is fixed during the execution of the whole task graph. Assigning a task to a processor we need to be sure that there exist enough code memory at this processor. This requirement can be formalized using reified constraints as follows.

```
for each task $i$                                                        (4.15)
  for each resource $j$
    $p_{ij} :: \{0..1\}$
    impose $r_i = j \Leftrightarrow p_{ij}$
for each resource $i$
  impose $CodeMemory_i \geq m_1 \cdot p_{1i} + m_2 \cdot p_{2i} + \cdots + m_n \cdot p_{ni}$
```

where n is the number of tasks, k is the number of processors, r_i is the execution resource of task i, m_i is the code memory size required by task i and p_{ij} is a 0/1 variable indicating, if 1, whether task j is run on processor i. The reified constraints collect information to which processor a task is assigned. Inequalities impose the constraint on code memory.

Data memory constraints are more complex since data memory usage changes during execution of tasks. These constraints can be divided into two classes. The first class of constraints defines requirements for allocation of data used during task execution. We call this data *computational data*. The second class encompasses data that are transfered between tasks and we call it *communication data*. We assume, without lost of generality, that computational data of size s_i is allocated by a task during its execution, i.e., during the interval $\{t_i..t_i + d_i\}$. Obviously, it is possible to generalize this assumption and model a situation when tasks allocate different amount of data during different phases of their execution. This data requirement can be modeled using `diff2` constraint. Each data for this constraint is modeled as a rectangle that starts at time t_i, at address a_i, has duration d_i and size s_i. Note, that if we have a number of local memories we need to divide address space for `diff2` constraint in such a way that it represents different memories in one constraint. Basically, we shift address spaces for consecutive memories and rectangles can move between different memories before eventually they are allocated to a given local memory. The constraints for modeling data memory are presented below.

```
for each task $i$                                                        (4.16)
  $a_i :: \{0..S\}$
  for each resource $j$
    impose If $r_i = j$ Then $j \cdot size \leq a_i < j \cdot size + size$
impose diff2 $([[t_1, a_1, d_1, s_1], [t_2, a_2, d_2, s_2], \ldots, [t_m, a_m, d_m, s_m]])$
```

where we assume that all memories are of size $size$, $S = n \cdot size$, and there is n processors (resources) and m tasks. Figure 4.6 illustrates `diff2` constraint for data memory allocation for three tasks and two processors. Tasks 0 and 1 are assigned to processor 0 and their data are allocated to memory of processor 0. Data for task 2 assigned to processor 1 are allocated to memory of this processor. `Cumulative` constraint can be additionally used as implied constraint to improve propagation (see [86] for details).

Communication data define buffers for data transfers between tasks. These buffers need to be allocated for both producer and consumer tasks and have to be reserved during the whole transmission. If tasks are allocated to the same processor the buffers can

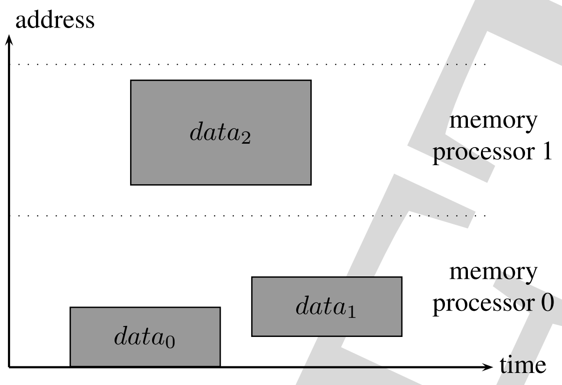


Figure 4.6: Modelling of data memory allocation with `diff2` constraint for two processors and three tasks.

be implemented as a shared memory and if tasks are assigned to different processors both producer and consumer tasks need to have such buffers in their local memories.

Consider two tasks, producer task i and consumer task j and communication ij between these tasks. The following pseudo-code defines constraints and rectangles that model allocation of communication data into memories. The defined rectangles are added to `diff2` constraint from 4.16.

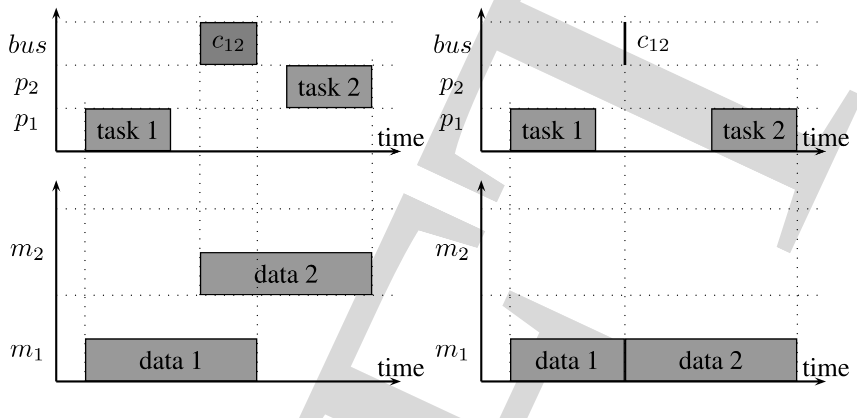
$$\begin{aligned} &rectangle_{producer} = [t_i, a_i, t_{ij} + d_{ij} - t_i, s_{ij}] \\ &rectangle_{consumer} = [t_{ij}, a_j, t_j - t_{ij}, s_{ij}] \\ &\texttt{impose}\ r_i = r_j \Leftrightarrow a_i = a_j \end{aligned} \tag{4.17}$$

The graphical representation of this rules is depicted in Figure 4.7. In Figure 4.7 a) the situation when tasks are assigned to different processors is depicted.The data buffers “data 1” and “data 2” are allocated in the local memories for the respective two processors that execute tasks 1 and 2. They overlap during communication since data are transfered during this time. Figure 4.7 b) represents situation when tasks communicate through the shared memory. The buffers are allocated in the same memory and they use the same area as enforced by a equality constraint specified in 4.17.

Power Consumption Constraints

Other constraints can also be defined for particular problems. For example, power consumption of a task on a given resource can be modeled using a concept similar to modeling task execution time on different processors (4.7). This is possible if power can be modeled using discrete values. It is usually not a problem since we can scale power values and obtain a required precision.

To model power consumption, we introduce for each task an additional FDV, p_i, that defines task power consumption. A relation between a resource and its power consumption using `element` constraint is then added to combine power consumption of a particular task on a specific resource. The cumulative constraint can then be used to model the restriction on power consumption as specified below



(a) tasks assigned to two different processors (b) tasks assigned to the same processor

Figure 4.7: Modelling of communication data with `diff2` constraint.

```
for each $i$
  impose element($r_i, [v_1, \ldots, v_n], p_i$)
impose cumulative($[t_1, \ldots, t_n], [d_1, \ldots, d_n], [p_1, \ldots, p_n], PowerLimit$)
```

This solution does not influence efficiency of a solver very much, regardless of a discretization granularity, since power variables, p_i, are usually not explicitly assigned during search. Their values are usually assigned by propagation of decisions on selected resources, assignment to FDVs r_i.

4.4 Partitioning

The purpose of partitioning is to group objects in such a way that a given cost function is optimized and design constraints are fulfilled [28]. The main goal of partitioning is to optimize basic design parameters, such as design cost, performance and power consumption. Partitioning helps to divide design into smaller blocks that can be more efficiently handled by designers and related design tools.

Formally, a *k-way partitioning* is defined as follows. For a given set of n objects $V = \{v_1, v_2, \ldots, v_n\}$ find partitioning to k groups $\{G_1, G_2, \ldots, G_k\}$ such that $V = G_1 \cup G_2 \cup \cdots \cup G_k$, and $G_i \cap G_j = \emptyset$ for all $i, j, i \neq j$. This definition states that each object must be assigned to one group.

The *partitioning problem* is to find a k-way partitioning of a set of n objects, so that the cost of the partitioning is minimal and a set of constraints is satisfied.

Partitioning is performed at various abstraction levels as well as at different stages of the design process. To represent these different partitioning problems in a unified way a common representation based on graphs is usually used. A node of this graph represents an object and it has usually assigned attributes that define different parameters of this node. For example, if a node represents a task it can have assigned execution time, code memory size, data memory requirements and power consumption. An arc between two nodes typically represents relation between the nodes and has assigned attributes as well. It can, for example, represent communication between tasks and the related attribute defines number of bytes transfered between these tasks. An example of a partitioning graph is depicted in Figure 4.8. It has assigned weights to both nodes and edges. Their interpretation can vary but for the purpose of this section we assume that edge weight represent number of bytes that need to be communicated between two nodes. The node weight represents a number of a particular resource needed for task execution. It can represent, for example, size of a code memory or power consumption of a task.

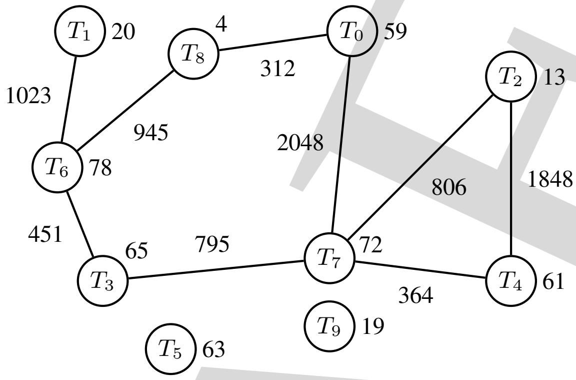


Figure 4.8: An example of a partitioning graph.

Assume that we would like to make a k-way partitioning of a partitioning graph. The partitioning problem states that cost function has to be minimized while constraints has to be fulfilled. This definition fits perfectly to constraint programming framework. We define a cost function and a set of constraints and then perform minimization of our cost function. Typically we would like to minimize communication between partitions while fulfilling other system related constraints, such as size of each partition. The cost of communication between partitions can be computed using the following constraints.

$$
\begin{aligned}
&p_0 :: \{0..k\}, p_1 :: \{0..k\}, \ldots, p_n :: \{0..k\} \\
&\textbf{for each } \texttt{edge } (T_i, T_j) \\
&\quad p_i \neq p_j \Leftrightarrow c_{ij} \\
&Cost = \sum_{ij} w_{ij} \cdot c_{ij}
\end{aligned}
\tag{4.18}
$$

where p_i denotes FDV for node i which value indicate the partition number selected for this node. The reified constraints compute 0/1 value indicating whether tasks i and j belong to the same partition (value 0) or different partitions (value1). Finally the cost is computed as a weighted sum of weights assigned to edges. Since edges between nodes belonging to the same partition has value zero the cost adds only weights for edges crossing partitions boundaries.

The constraints that must be fulfilled by the solution differe very much between different applications and therefore it is diffciult to define typical constraints. They can represent different classes of constraints. for example, one would like to constraint partition size based on several factors, code size, power consumption,

Example 4.11: Consider the partitioning graph depicted in Figure 4.8. Assume that we would like to partition this graph to two partitions of roughly the same size but minimized the sum of weights of edges going between partitions. By roughly the same size we mean that each partition must have the size of at least 45% of the sum of all weights assigned to the nodes. The following constraints and search method define a method for finding a solution to this problem.

```
// partitioning FDVs
for i = 0..9
  $p_i = \{0..1\}$
  $np_i = \{0..1\}$
  $np_i + p_i = 1$
// communication FDVs $c_{ij} = 1$ means that
// $i$ and $j$ are in different patritions
impose $p_0 \neq p_7 \Leftrightarrow c_{07}$
impose $p_0 \neq p_8 \Leftrightarrow c_{08}$
impose $p_1 \neq p_6 \Leftrightarrow c_{16}$
impose $p_2 \neq p_4 \Leftrightarrow c_{24}$
impose $p_2 \neq p_7 \Leftrightarrow c_{27}$
impose $p_3 \neq p_6 \Leftrightarrow c_{36}$
impose $p_3 \neq p_7 \Leftrightarrow c_{37}$
impose $p_4 \neq p_7 \Leftrightarrow c_{47}$
impose $p_6 \neq p_8 \Leftrightarrow c_{68}$

impose $Size_1 = 59 \cdot p_0 + 20 \cdot p_1 + \cdots + 19 \cdot p_9$
impose $Size_2 = 59 \cdot np_0 + 20 \cdot np_1 + \cdots + 19 \cdot np_9$
// sum of all node weights is 454; 0.45 * 454 = 204.3
impose $Size_1 \geq 205$
impose $Size_2 \geq 205$

impose $Cost = 2048 \cdot c_{07} + 312 \cdot c_{08} + \cdots + 945 \cdot c_{68}$

minimize($[p_0, p_1, \ldots, p_9], Cost$)
```

The solver generated solution with two groups. The first partition constains p_0, p_2, p_4, p_7 and p_9 and has size 230 and the second one contains p_1, p_3, p_5, p_6 and p_8 and has size 224. The communication cost between partitions is 1107.

4.5 Network on Chip

Developments in technology provide opportunities to put more and more components on a single chip. This led to development in system design that provides a full system on a chip that is called System on Chip (SoC). Since different parts of the system need to communicate there exist clear need to provide an efficient communication facility on the chip. Classical bus oriented architectures can be applied to simple systems with low communication requirement but many applications require higher communication bandwidth. There exist proposals for specific communication networks to solve this problem. Communication architectures are called Network on Chip, NoC. In this book, we will use an example of a specific network structure called *Nostrum* Mesh Architecture [64].

Nostrum architecture is a regular mesh structure with computational resources located in mesh cells and switches on intersections of communication lines. Each computational resource is connected to a single switch. The network is packet switched. Communication between two tasks assigned to computational resources is handled by the network by routing a message through several switches. The basic architecture is depicted in Figure 4.9. Compuational resources are denited by R and switches by S.

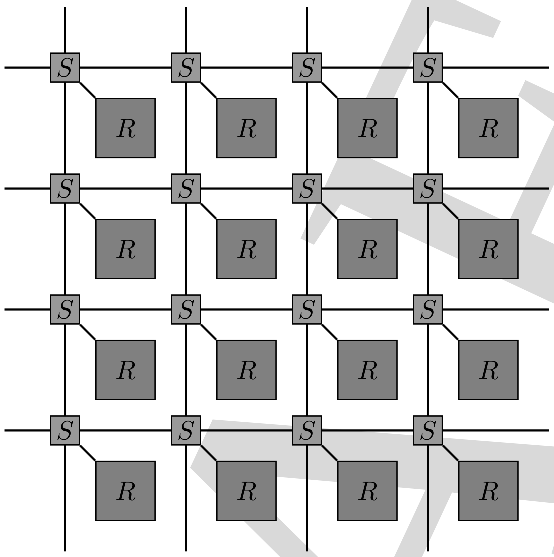


Figure 4.9: Nostrum Mesh Architecture.

In this discussion we assume that the mapping of tasks into computational resources has been already done. The main task which remains to be done is to find a mapping of computational resources to mesh cells. This is not a trivial tasks since there are number of requirements and optimization goals which has to be fulfilled.

We assume that each computational resource is given as an IP block (intellectual property block), either hard or soft. That is it has either a fixed geometrical dimensions (hard IP block) or the dimensions can be controlled during synthesis of a soft block. Therefore we assume that we might be able to use blocks of different geometry and they can be rotated. In addition, we assume that the dimensions of the grid are determined by the maximum sizes of blocks. The size of column i is determined by the maximum width of a block in dimension x. Similar property applies to rows.

The placement of blocks has to obey given distance constraints between different blocks. The distance between blocks is defined as a number of “hopps” between switches and it is basically equal to the Manhattan distance between these blocks. We would like to optimize the lost area which is defined as an difference between grid area and the area of all placed blocks. In addition, it is also important to minimize the used bandwidth.

We define first FDVs relevant for our problem. We assume a 2-D mesh topology with blocks of size 1×1. Each block has defined its position by two FDVs x_i and y_i For example, assuming 10×10 grid we define x_i :: $0..9$ and y_i :: $0..9$. The blocks cannot be placed at the same position and therefore we use a combinatorial constraint diff2. We use rectangles defined by the following 4-tuples [x_i, y_i, 1, 1] and diff2 constraint is specified by a list of all such rectangles.

To specify distance constraints between blocks we use distance constraints that define absolute value between two FDVs. The definition of constrains on a number of N "hoops" between two blocks is defined as follows.

```
$Xdistance :: 0..N, Ydistance :: 0..N,$
$distance_{i,j} :: 0..2 \cdot N$
impose distance($x_i$, $x_j, Xdistance$)
impose distance($y_i$, $y_j, Ydistance$)
impose $Xdistance + Ydistance = distance_{i,j}$
```

Using these constraints we also measure a distance between two blocks without limiting their placement.

The constraints discussed so far does not take into account different sizes of our blocks and how they influence sizes of rows and columns of our 2-D grid. In our model, we assume that each block has a specification of possible sizes in X and Y direction defined as a vector. For example, a block can be defined by the following vector [[1,3],[3,1]] that specifies the size of this block to be either 1×3 or 3×1. The selection of the size for a block is defined by FDV g_i that specifies the block geometry. For the block defined above we define g_i :: $0..1$, indicating that either geometry 0 or 1 can be selected. The specific geometry for a block is selected using element constraint. The constraints for selection between two geometries ([1,3] and [3,1]) are specified below. The first element constraint selects length of the cell in X direction while the second one in Y direction.

```
impose element($g_i, [1, 3], Lx_i$)
impose element($g_i, [3, 1], Ly_i$)
```

Finally, we define rows and column sizes using conditional constraints and maximum constraints. Basically we assign a size to each row and column based on values assigned to x_i and y_i and then find a maximum size of a column or a row. This is later used to compute a total area of a grid and find out the lost area. A piece of code presented below generates constraints for finding column sizes for all grid cells and defines constraints for the size of column i. Similar constraints are defined for rows. Note that If-Then-Else is a conditional constraints and Max is the maximum constraint.

```
for(int i=0; i < numberOfBlocks; i++)
  for (int j=0; j < numberOfColumns; j++)
    impose If $x_i = j$ Then
               $blockXlength_{i,j} = Lx_i$
             Else
               $blockXlength_{i,j} = 0$

for (int i=0; i < numberOfColumns; i++)
  for (int j=0; j < numberOfBlocks; j++)
    $LxI_j \leftarrow blockXlength_{j,i}$
  impose max($LengthX_i, LxI$)
```

The cost function is defined as a weighted sum of two optimizations criteria: the lost area and the required bandwidth (see eq. (4.19)). The bandwidth part is defined as a product of the required bandwidth between two blocks and their distance. This helps to minimize the distance between blocks which need high communication bandwidth.

$$Cost = w_1 \cdot LostArea + w_2 \cdot \sum_{i,j} distance_{i,j} \cdot bandwidth_{i,j} \tag{4.19}$$

The search uses partial search methods and branch-and-bound principle. The selected partial search method is a credit search as discussed in section 5.2.3. During search there is a number of decisions which have to be made. The algorithm has to select a next block for assignment and define its position. Our algorithm selects a block from a list of blocks defined by 3-tuples $[x_i, y_i, g_i]$. The selection order is defined using the following principle. First we select blocks with lowest mobility (i.e., the sum of their domain sizes for x_i and y_i is smallest). Then if there are several blocks with the same mobility we select a block with the highest number of constraints assigned to variables x_i and y_i. This selection is dynamic, which means that we follow domain pruning for our FDVs. Once the block is selected we try to assign FDVs x_i, y_i, g_i in this order. The value which is assigned to the FDV is selected as a middle value from variable's domain. In case of backtracking we select values from the left and right part of the domain alternately.

Example 4.12: Consider NoC mapping example with 25 blocks that need to be mapped into a mesh network of size 5×5. There are three types of blocks. The blocks of type one have possible sizes 2×3 or 3×2, blocks of type two can have size 1×4, 4×1 or 2×2 and finally type 3 blocks are always of size 3×3. Blocks numbered 0-9 are of type 1, 10-19 type 2 and 20-24 of type 3. Additional constraints are imposed on distances between blocks. Blocks 0-8 must be located at distance 2 since they form a pipeline. In addition to minimization of the lost area we would like to minimize the bandwidth required for communication between block 10 and blocks 20-24. The required bandwidth for this communication is 3.

One possible solution that has been obtained is depicted in Figure 4.10. Note several interesting features of this solution. The lost are has been minimized and it is 12.1% of the total area (20 of 165). Blocks 0-8 has been placed in a direct neighborhood to each other as distance constraints required. Block 10 is placed in the middle and it has a short distance to blocks 20-24 since this was imposed by the optimization function (the bandwidth cost is 33). The solution has been found using an incomplete search method and therefore it is not known whether it is optimal.

Questions and Exercises

1. The data-flow graph for the 5th-order elliptic wave filter is depicted on Figure 4.11. It consists of 8 multiplications and 26 additions. These operations need to be scheduled on multipliers and adders. Write a program that will optimize the schedule length for different amount of resources (e.g., 1 multiplier and 1 adder or 2 multipliers and 2 adders). Assume that multiplication is two clock cycles and addition one long. The program should be written in such a way that you can easily specify other values. Make your program data independent, so if new operation or new operation type is added the program does not have to be changed but only the database of facts.

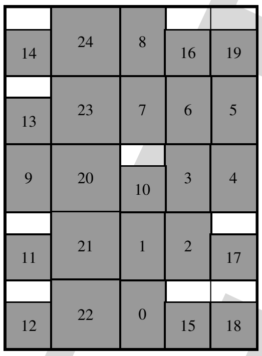


Figure 4.10: An example of a solution for NoC mapping problem.

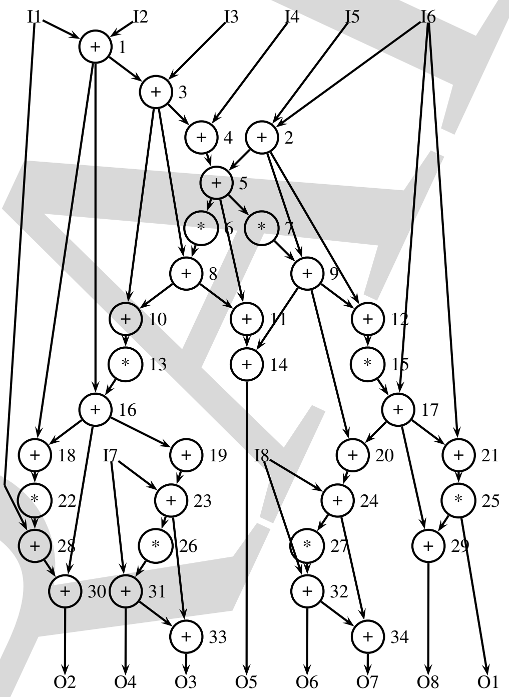


Figure 4.11: DFG for fifth order elliptic filter.

2. The video coding algorithm H.261 derived from [8] is depicted in Figure 4.12. The task graph contains 12 tasks and 14 interconnections between them. The tasks can be implemented on different execution units with different execution time as defined in Table 4.4. Tasks IN, FB1, FB2 model external memories and tasks Q and IQ model external environment. Their execution time is zero. The communications are implemented on a shared bus and have transmission time assigned to arcs specifying these transfers in Figure 4.12. The execution and communication times are specified in clock cycles. Write constraints and

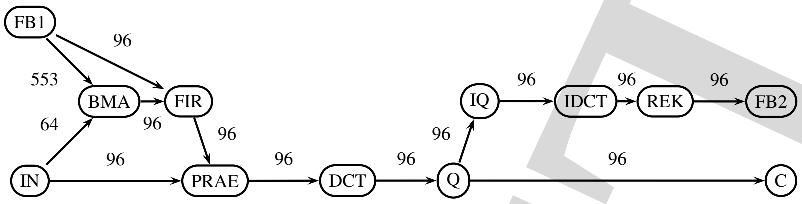


Figure 4.12: H.261 task grpah.

search for finding optimal task assignment and scheduling. Try to implement pipelining of the task graph to improve performance.

The author of [8] does not allow in his ILP model for pipelining to start a new task graph computation on a given resource before all previous computations of this task graph did not finish their executions. Has your CP model the same limitation?

Table 4.4: Execution times for H.261 tasks on different units.

Task	Universal	BMA array	PAR1 PAR1	DCT array	FIR array	BMA pipe	FIR seq	FIR pipe	DCT seq	DCT pipe
BMA	7234	484	-	-	-	3617	-	-	-	-
FIR	7234	-	-	-	510	-	3461	1170	-	-
PRAE	1280	-	128	-	-	-	-	-	-	-
DCT	12312	-	-	132	-	-	-	-	6156	474
IDCT	12312	-	-	132	-	-	-	-	6156	474
REK	1536	-	256	-	-	-	-	-	-	-
C	132	-	-	-	-	-	-	-	-	-

3. The NASA/JPL Mars Pathfinder [54] is a rover designed to roam on the surface of Mars to several target locations for an extended period of time. Its tasks include performing experiments and taking snapshots and transmitting them wirelessly back to the spaceship. It operates on very limited power supply. The power is given by solar panels. The power obtained from solar panels was measured at different temperatures and the results were the following: 14.9W at -40 °C, 12.0W at -60°C and 9.0W at -80°C. There is a battery power source too, which gives maximal 10.0W. The Mars rover has 6 driving and 4 steering motors, which need to be warmed up before respective driving and steering can be performed.

   The Mars rover main task is to explore Mars and therefore it must move from one place to another. There are two types of constraints in this problem. Timing and power constraints are represented in Tables 4.5 and 4.6. Timing constraints are always applied to the start time of an operation.

   For all temperatures, schedule the specified operations in such a way that the Pathfinder will finish one full phase of moving, as shown in Figure 4.13, in shortest time without extending power limit. How can you make sure that shortest schedule uses minimal amount of power?

Since data for power is not given as integers, there is a need to multiply all power data by 10 to obtain data which is representable by integers therefore it is possible to use the FD solver.

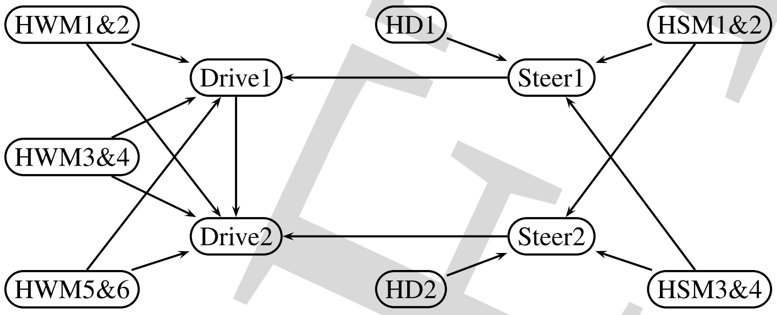


Figure 4.13: Mars pathfinder operations for one full phase.

Table 4.5: Timing constraints for Mars Pathfinder rover.

Operation	Duration (s)	Timing constraints
Heating steering motors	5	At least 5s, at most 50s before steering
Heating wheel motors	5	At least 5s, at most 50s before driving
Hazard detection	10	At least 10s before steering
Steering	5	At least 5s before driving
Driving	10	At least 10s before next hazard detection

Table 4.6: Power constraints for Mars Pathfinder rover.

Tasks	Duration	Power (W)		
	(s)	-40°C	-60°C	-80° C
Heat two motors	5	7.6	9.5	11.3
Drive	10	7.5	10.9	13.8
Steer	5	4.3	6.2	8.1
Hazard Detection	10	5.1	6.1	7.3
CPU	Constant	2.5	3.1	3.7

Five

Search

He who can properly define and
divide is to be considered a god.

Plato (ca. 429-347 BC)

Test everything. Hold on to the good.
Avoid every kind of evil.

(1 Thessalonians 5:21-22)

5.1 Complete Search Methods

All imposed constraints are checked for consistency using the propagation loop that applies constraints' consistency methods. This procedure prunes some FDVs and enforces constraint consistency. It is, however, not a complete procedure and our set of constraints is not solved yet. In fact, it might not exist a solution to the imposed constraints. To find a solution we have to find an assignment that satisfies all constraints. If we are not able to find such an assignment we have a proof that our set of constraints is not consistent.

An assignment that satisfies all constraints is found using a method based on the *constraint-and-generate* approach (see section 2.5). This method assigns a new value to a selected FDV and performs consistency checking. If at some point inconsistency is detected (by obtaining an empty FD) the method backtracks. The backtracking annuls the last decision by removing all values changed by this assignment. A new assignment is then performed.

The standard algorithm used in constraint programming for implementation of constraint-and-generate method is depth-first-search (DFS) algorithm. In CLP community this is also called *labeling*. The DFS algorithm selects an FDV, from a given vector of FDVs, and a value from its domain. This value is then assigned to the selected FDV by imposing the equality constraint. The solution is found if all FDVs from a given vector have a value assigned.

An implementation of DFS method is often called chronological backtracking and is discussed in the next subsection. The method can be backtrack-free when specific

```
label DFS($\mathcal{V}$)
  if solver.consistency()
    if $\mathcal{V} \neq \emptyset$
      $var \leftarrow$ select FDV from $\mathcal{V}$
      $\mathcal{V}' \leftarrow \mathcal{V} \setminus var$
      $v \leftarrow$ select value from $D_{var}$
      impose $var = v$
      $result \leftarrow$ DFS($\mathcal{V}'$)
      if $result \neq$ nil
        return $result \cup (var, v)$
      else
        undo all decisions // backtrack
        impose $var \neq v$
        return DFS($\mathcal{V}$)
    else
      return {}    // solution found, return empty result
  else
    return nil      // inconsistent constraints
```

Figure 5.1: Depth first search algorithm.

conditions hold. This is discussed in subsection 5.1.2. Finally different improvements to the basic idea of chronological backtracking are presented in subsequent subsections.

5.1.1 Chronological Backtracking

The chronological backtracking is implemented using DFS algorithm. The simple DFS based search has already been introduced in Figure 2.10. In this section, we extend it and present in Figure 5.1 a generic DFS algorithm. This algorithm first checks consistency of the store. If all constraint are consistent it selects a FDV from a vector of variables, $\mathcal{V}$. Then it selects a value for this variable, imposes constraint $var = v$ and recursively calls itself. If this call fails and returns `nil` the DFS algorithm undoes all decisions and continues search with the same vector of FDVs and the new imposed constraint $var \neq v$. We will use this algorithm to discussed different options used in DFS search. We will also extend it later to address more advanced issues.

There are several decision that need to be made when implementing a DFS algorithm. First, we need to determine the method for selecting FDV from the vector of FDVs that can be given as a parameter to DFS method. This selection takes place in line 4. The second decision is the selection of a value from the domain of selected variable `var` (line 6).

There is many options to select a FDV from a vector of variables and solvers usually provide a way to explicitly define this option. The typical options are as follows.

- *input order* option selects the first variable from the current input vector,
- *first fail* option selects a variable with the smallest number of elements in its domain,

- *most constrained* option selects a variable that appears in most constraints,
- *smallest* option selects a variable that has the smallest minimal value in its domain,
- *largest* option selects a variable that has the greatest minimal value in its domain,
- *smallest maximal* option selects a variable that has the smallest maximal value in its domain, and
- *max regret* option selects a variable that has the largest difference between the smallest and the second smallest value in its domain.

The selection process is usually dynamic. The next variable for assignment is selected based on the current search status. This means that pruning of FDVs and its "strength" might influences the selection of variables and search strategy.

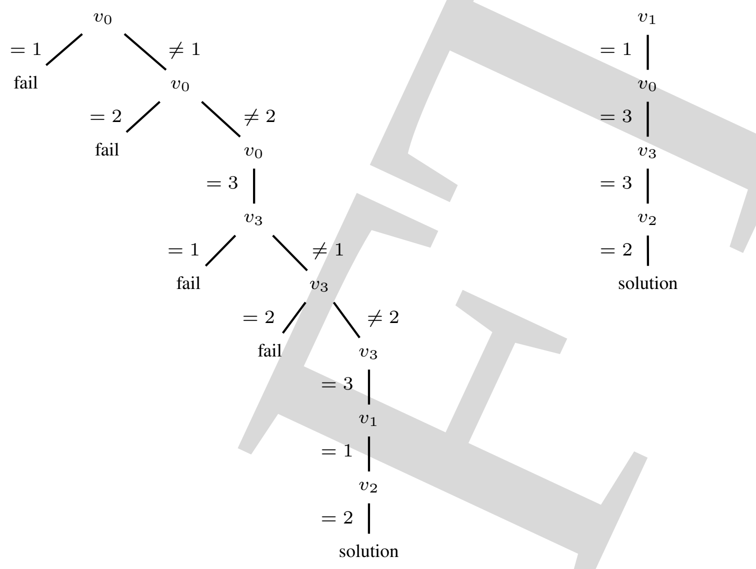
The other decision that DFS algorithm makes is the selection of a value form the domain of the selected FDV. The typical options that are used to achieve this are as follows.

- *minimal* value from the domain is selected,
- *maximal* value from the domain is selected,
- first the *middle* value from the domain is selected and then left and right values are tried, and
- *random* value from the domain is selected.

Example 5.1: Consider the graph coloring problem represented in Figure 1.1 with FDVs defined as follows $v_0 :: \{1..3\}$, $v_1 :: \{1..2\}$, $v_2 :: \{1..2\}$ and $v_3 :: \{1..3\}$. Assume that our vector of variables is $[v_0, v_3, v_1, v_2]$. The search tree for DFS algorithm with input order variables election option and minimal value assignment is depicted in Figure 5.2 a). It has four backtracks. The same example has a search tree with no backtracks if first fail option for variable selection is used instead, as depicted in Figure 5.2 b). Both search methods finds the same result.

Example 5.1 illustrates how important is to select right options for search. First fail option is specially well suited for many constraint programming examples and it has good intuitive motivation. Selection of small domains at the beginning of the search is motivated by the fact that small domains might provide higher opportunity for the solver to fail with the assignment. We simply try first assignments where it is most likely to detect failure. This strategy will remove parts of the search space early in the search. If the assignment produces a failure it is better to backtrack there since later fail means need to backtrack to early decision located often close to the root of the search tree. This means time consuming process that might never finish in practice.

Other variable selection options have also good motivations. Most constrained option tries to enforce as much propagation as possible at the beginning of the search. It selects a FDV that is present in most constraints and therefore it might wake up large number of constraints and achieve good search tree pruning. Smallest and smallest maximal options are justified for scheduling applications. They are usually applied to FDV that defines starting time of a task. It means that search tries to schedule tasks



(a) input order option (b) first fail option

Figure 5.2: Search trees for graph coloring example.

based on their urgency. It reminds a little bit priorities used in list scheduling methods [28]. Finally, max regret option is usually used when cost function minimization is performed and we would like to check first a small number in variable domain before going to the next one which is much larger. This is why we try a small value first and, if this value fails later in the search tree, the next value is not much larger since the largest next values where tried at the beginning of the search.

There are usually several FDVs with the same value for the variable selection option. In such cases we can use a second option to break ties between variables. For example, first fail option can be combined with most constrained option. If there is several variables with the same size of their domains then the most constrained variable among them is selected. This is often implemented in the solver implicitly.

In the discussion so far we have assumed that we have a vector of FDVs that are selected for assignment. This has a hidden assumption that all problem decision variables are comparable in some sense and our selection options can be uniformly applied to them. This is not always true. There are problems that require several different types of FDVs that model different aspects of the problem. For example, if we model scheduling and assignment we have two types of FDVs. Task i starting time, t_i, belongs the first type of variables but a resource assigned for task execution, r_i, is a different type of variable. They are not comparable in respect to selection options and placing them in the same vector of FDVs will not create a good search strategy. In such a case, we need to take this difference into account when organizing our search.

One method that is often used in constraint programming is based on grouping variables and organize search in such a way that it considers differences in groups. We can built a vector that contains subvectors of grouped variables. For example, subvector $[t_i, r_i]$ groups start time and assigned resource for task i. Selection can be made based on FDV t_i but assignment will try to assign first t_i and then r_i. In this way decisions on task i are grouped together. Another way to group variables is to build two vectors, one containing all t_i and the other one containing all r_i. Search, in this case, can assign first all t_i and then all r_i or vice versa. To preserve opportunity for obtaining optimal results backtracking need to be possible between both subvectors.

Example 5.2: Consider data-flow graph depicted in Figure 4.3. Two different search methods that are based on the discussion above can be defined for scheduling and assignment of this data-flow graph.

In the first method we can create vector

$$[[t_1, r_1], [t_2, r_2], \ldots, [t_{11}, r_{11}]]$$

and select two FDVs t_i and r_i for value assignment based on the first variable using, for example, smallest selection option.

In the second approach we can create the following vector

$$[[t_1, t_2, \ldots, t_{11}], [r_1, r_2, \ldots, r_{11}]]$$

and assign values first to the first vector and then to the second one. We can also apply different variable selection criteria for the first and the second vector.

Obviously there are other possibilities as well.

So far we have discussed DFS strategies that try to organize search space based on variable assignment. The are several options to implement this search strategy. One can assign consecutive values to a variable (see Figure 2.10) or make a decision that a variable is equal value v or it is not equal v (see Figure 5.1). Constraint programming offers opportunity to implement much more advanced strategies that are based on *domain splitting*. Domain splitting reduces the domain of each variable, by splitting the current domain into several possible subdomains. Since the solver can call consistency methods after each decision we can trigger consistency checking of involved constraints even the final value is not yet determined. Many constraints can prune domains of involved variables with assigned subdomains since their consistency methods are based on bound consistency and do not require final assignment to a variable. This offers opportunity to discover infeasible parts of the search tree for subintervals instead of values that possibly can lead to more efficient search strategies.

The most popular domain splitting method splits the domain into two parts, the lower and upper halves. In this way, we make a "weak commitment" to lower or upper halve of the domain but the solver is usually able to prune domains and, in some cases, detect unfeasibility. The algorithm is depicted in Figure 5.3. It is similar to the standard DFS algorithm but instead of imposing assignment constraints it imposes constraints that split the domain, i.e., $var \leq mid$ or $var > mid$. The algorithm also checks if the variable have got a value assigned and, if this is the case, it continues the search for remaining variables. Otherwise it still includes the variable on the list of variables for assignment. This may lead to larger search trees.

5.1.2 Backtrack-Free Search

In the previous section, we have noted that different order can lead to different number of backtracks. An interesting question is whether there exist a class of constraint

```
label DFS_split($\mathcal{V}$)
  if solver.consistency()
    if $\mathcal{V} \neq \emptyset$
      $var \leftarrow$ select FDV from $\mathcal{V}$
      $\mathcal{V}' \leftarrow \mathcal{V} \setminus var$
      if $min(var) = max(var)$
        $result \leftarrow$ DFS_split($\mathcal{V}'$)
        if $result \neq$ nil
          return $result \cup (var, min(var))$
        else
          return nil
      else
        $mid \leftarrow min(var) + (min(var) + max(var))/2$
        impose $var \leq mid$
        $result \leftarrow$ DFS_split($\mathcal{V}$)
        if $result \neq$ nil
          return $result$
        else
          undo all decisions // backtrack
          impose $var > mid$
          return DFS_split($\mathcal{V}$)
    else
      return {}    // solution found, return empty result
  else
    return nil      // inconsistent constraints
```

Figure 5.3: Depth first search algorithm with domain splitting.

programming problems for which we can find an ordering that search does not need to backtrack. The problem has been extensively studied and there is several results [25]. In this section, we will concentrate on constraint problems that have this property because they have a special structure of their constraint graph. That is this property does not depend on actual constraints. More discussion on special properties of constraint relations that lead to backtrack-free search can be found in [25].

Formally, a given CP problem is backtrack-free for a given order of variables $(x_1, \ldots, x_n)$ if for every $i < n$, every partial solution $(x_1, \ldots, x_i)$ can be consistently extended to include x_{i+1}. To be able to discuss backtrack-free search we need to define several properties for constraint graphs.

An *ordered constraint graph* is a constraint graph whose nodes have been ordered linearly. The width of a node in an ordered constraint graph is the number of parent nodes. The width of the ordered constraint graph is the maximum width of any of its nodes. Finally, the width of the constraint graph is the minimum width of all the orderings of that graph. In general the width of a constraint graph depends upon its structure.

Example 5.3: Consider a simple graph coloring problem with three nodes, A, B

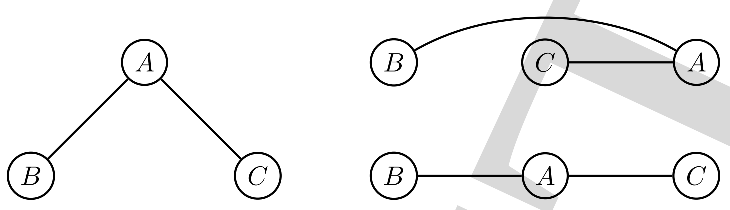


Figure 5.4: The constraint graph and two possible orderings of FDVs for Example 5.3 problem.

and C, and the following constraints.

$$A :: \{1..2\}, B :: \{1..2\}, C :: \{1..2\}$$
$$A \neq B$$
$$A \neq C$$

The constraint graph and two possible orderings of FDVs for this problem are depicted in Figure 5.4. We see that the ordering depicted at the upper part of Figure 5.4 b) has width 2 while the other ordering has width 1. The width of the constraint graph is therefore 1.

Fredure [31] proposed the following theorem that connects together ordered constraint graph width, consistency method and backtracking-free search.

Theorem 2 (Freuder [31]). *Given a constraint satisfaction problem:*

1. *A search order is backtrack-free if the level of strong consistency is greater than the width of the corresponding ordered constraint graph.*
2. *There exists a backtrack-free search order for the problem if the level of strong consistency is greater than the width of the constraint graph.*

Proof. In instantiating any variable v we must check consistency requirements involving at most j other variables, where j is the width of the node. If the consistency level k is at least $j + 1$, then given prior choices for the j variables, consistent among themselves, there exists a value for v consistent with the prior choices. □

Based on the presented theorem, we can conclude that tree-structured binary constraint graphs have a special property. Any such arc consistent graph has backtrack-free search for variety of orderings since there exist many orderings with width 1.

Furthermore arc, path and k-consistency have been extended to handle search when a specific order of variables is defined. Directional arc consistency, directional path-consistency and directional k-consistency has been developed to restrict consistency checking to a given order of variables [25].

5.1.3 Back Jumping

Chronological backtracking always backtrack to the previous decision which, in constraint programming context, usually means the previous FDV and its assignment.

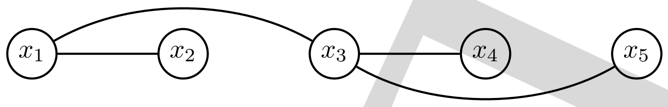


Figure 5.5: An example constraint graph for Example 5.4 problem.

This is done to resolve the conflict that occurred during the current assignment. DFS algorithm simply tries to move one level up and change the previous decision by making a different assignment to the directly preceding variable. This strategy, while simple and correct for most problems, is not the most efficient one. The reason is that the conflict detected during current assignment might not be directly resolved using different assignment to the preceding variable. The assignment conflict might exist between current variable and and a variable assigned much earlier in the search tree, i.e., several levels up from current search tree node. In such situations backtracking to directly preceding variable has no much sense. The algorithm need to backtrack to the variable assignment that caused this conflict. The technique that implements this kind of intelligent backtracking is called backjumping.

Example 5.4: Consider a constraint graph depicted in Figure 5.5 and the ordering of FDVs $[x_1, x_2, x_3, x_4, x_5]$. Assume also that the search algorithm need to backtrack at state denoted by variable x_3. Chronological backtracking would backtrack to state that assigns variable x_2 but this variable is not connected by any constraint with x_3 and therefore changing its value will not resolve the conflict. Intelligent backtracking needs therefore backtrack to the state that has a variable that can resolve this conflict. In our example, this is variable x_1 and backjumping backtracks to this variable.

To discuss backjumping methods we need to look more carefully when backtracking is initiated. It is basically initiated always when all values for a given FDV has been in conflict with the current (partial) assignment. We call such a state *dead-end state* and the variable a *dead-end variable*. This is defined more formally later in this section. Further we distinguish between two situations of dead-end states. The first dead-end state takes place in the situation when all values form FDV domain has been tried and non consistent assignment exist. In such case, since there is no more values to try we need to initiate backtracking. This FDV is called *leaf dead-end variable* and the state *leaf dead-end state*. The second situation takes place when backtracking returns to the state which has no more values availble to be assigned to the variable and therefore the backtracking is again initiated. Such a state is called *internal dead-end state* and the variable *internal dead-end variable*.

Example 5.5: Consider again the constraint graph depicted in Figure 5.5 and the ordering of FDVs $[x_1, x_2, x_3, x_4, x_5]$. If the search fails to find a legal assignment to FDV x_5, this state becomes the leaf dead-end state and the search backjumps to x_3. If all values for x_3 has been already assigned this state is internal dead-end state and the search can backjump to x_1.

Different backjumping algorithms provide different ways of coping with these two situations. Gaschnig's backjumping algorithm, for example, collects information directly during search but it is able to make safe and maximal backjumping only for leaf dead-end states of the search tree. Graph-based backjumping extends the concept to internal dead-end states but it needs a constraint graph. Furthermore, since it does not use conflict information during the search, it makes quite conservative assumption for backjumping. Finally, conflict-directed backjumping (CBJ) algorithm combines methods proposed by the previous two algorithms into a single framework [25]. We will discuss here conflict-directed backjumping.

The CBJ algorithm does not use constraint graphs but maintains special data structures during search. The necessary information is gathered in the so called *jumpback set* that is used to select the latest variable in this set for backjumping. To define this set we need to introduce several definitions first. We will follow the formalism proposed in [25].

In our discussion, we first define formally the dead-end that will be used later for identifying backtrack states.

Definition 3 (dead-end)**.** *A dead-end state at level i indicates that a current partial assignment $\vec{a}_i = (a_1, \dots, a_i)$ conflicts with every possible value of x_{i+1}. $(a_1, \dots, a_i)$ is called a* dead-end state*, and x_{i+1} is called a* dead-end variable

At the dead-end a DFS algorithm tried to assign all possible values to x_{i+1} but failed. Therefore it does not exist any consistent assignment for x_{i+1} in the current partial context and DFS needs to backtrack. In chronological backtracking, it would backtrack to assign a new value to variable x_i. In CBJ we try to find a conflict set of FDVs and deduce the backtrack point. We define first the *earlier constraint* based on the constraint scope, $scope(R)$. The constraint scope is the set of variables present in the constraint.

Definition 4 (erlier constraint)**.** *Given an ordering of the varibales in a constraint system, we say that constraint R is* earlier *than constraint Q if the latest varibale in $scope(R) - scope(Q)$ proceeds the latest varibale in $scope(Q) - scope(R)$*

For example, under FDV ordering $(x_1, x_2, \dots)$ and scope of constraints R_1 is (x_3, x_5, x_8, x_9) and scope of R_2 is (x_2, x_6, x_8, x_9), then R_1 is earlier than R_2 because x_5 proceeds x_6. With a given order of FDVs the relation from definition 4 defines a total order on the constraints.

Definition 5 (erliest minimal conflict set)**.** *Given an ordering of FDVs, let $\vec{a}_i$ be a tuple whose potential dead-end varibale is x_{i+1}. The* erliest minimal conflict set *of $\vec{a}_i$ (or of x_{i+1}) denoted $emc(\vec{a}_i)$ can be generated as follows. Consider the constraints $C = \{R_1, \dots, R_C\}$ with scopes $\{S_1, \dots, S_C\}$ ordered as defined in definition 4. For $j = 1..c$, if there exist $b \in D_{i+1}$ such that R_j is violeted by $(\vec{a}_i, x_{i+1} = b)$, but no constraint earlier than R_j is violated by this assignment, then $var\_emc(\vec{a}_i) \leftarrow var\_emc(\vec{a}_i) \cup S_j$. Finally, $emc(\vec{a}_i)$ is the subtuple of $\vec{a}_i$ projected over $var\_emc(\vec{a}_i)$.*

Finally, we can define *jumpback set* that is used for selecting a place where DFS algorithm will backtrack.

Definition 6 (jumpback set)**.** *The* jumpback set *J_{i+1} of a leaf dead-end varibale x_{i+1} is its $var\_emc(\vec{a}_i)$. The jump-back set of an internal state $\vec{a}_i$ (or varibale x_{i+1}) includes all the $var\_emc(\vec{a}_j)$ of all relevant dead-ends $\vec{a}_j$, $j \geq i$, that occured in the current session of x_i. Formally, $J_i = \bigcup \{var\_emc(\vec{a}_j) | \vec{a}_j$ is a relevant dead-end in x_i's session$\}$.*

```
vector CBJ($\mathcal{V}$)
  $i \leftarrow 1$    // variable selection pointer
  $D'_i \leftarrow D_i$ // domain copy
  $J_i \leftarrow \emptyset$ // conflict set initialization
  for $i = 1..n$
    $x_i \leftarrow$ Select_Value_CBJ
    if $x_i$ has no assignment
      $i_{prev} \leftarrow i$
      $i \leftarrow$ index of last variable in $J_i$ // value for backjumping
      $J_i \leftarrow J_i \cup J_{i_{prev}} \setminus \{x_i\}$ // merge conflict sets
    else
      $i \leftarrow i + 1$
      $D'_i \leftarrow D_i$ // reset domain
      $J_i \leftarrow \emptyset$ // reset conflict set
  if i = 0
    return null   // no solution
  else
    return values of $\{x_1, \ldots, x_n\}$

int Select_Value_CBJ()
  while $D'_i$ is not empty
    $v \leftarrow$ select value from $D'_i$
    remove $v$ from $D'_i$
    $consistent \leftarrow$ true
    $k \leftarrow 1$
    while $k < i \wedge consistent$
      if consistency($\vec{a}_k, x_i = v$)
        $k \leftarrow k + 1$
      else
        $R_S \leftarrow$ the earliest constraint causing the conflict
        add the variables in $R_S$'s scope $S$ excluding $x_i$ to $J_i$
        $consistent \leftarrow$ false
    if $consistent$
      return $v$
  return 'no assignment'
```

Figure 5.6: The conflict-directed backjumping algorithm.

For a given FDV x_{i+1} the algorithm backjumps to the latest variable in its set J_i. Note, that for internal dead-ends the set J_i contains FDVs that are in the earliest minimal conflict set for this variable as well as all FDVs assigned after this variable during search. In this respect, the algorithm extends Gaschnig's backjumping algorithm that can make safe and maximal jumps only at leaf dead-end states. It can be proved that CBJ backjumping to the latest variable in set J_i is safe, i.e., no solutions will be lost using this backjumping method.

The algorithm for conflict-directed backtracking is depicted in Figure 5.6. It dynamically creates conflict sets J_i and, in case of failure, backtracks to the last variable in its conflict set. This is implemented as an assignment of the index of the last variable in J_i to variable i. J_i is, on the other hand, created using current scope of conflicting variables as well as $J_{i_{prev}}$ that defines conflict sets for variables that were assigned after variable i. In this way, the internal dead-end states are handled correctly. Note, that the consistency method used in procedure `Select_Value_CBJ()` is different than the our consistency method that we have used so far. It checks consistency of an assignment $x_i = v$ in a partial assignment specified by vector of assignments $\vec{a}_i$. In this way the procedure can identify dynamically jumpback sets J_i.

5.2 Incomplete Search Methods

In this section, we will discuss search methods that does not necessarily need to be complete, i.e., check all possible assignments. That means that they might not search the whole search space and skip some parts purposely. As an effect of that they might miss solutions that exist in the parts of the search space that they skipped. These methods, however, depending on given parameters, can usually search the full search space as well.

5.2.1 Iterative Broadening

Chronological backtracking searches all possible assignments and has possibility to find all solutions to imposed constraints. In practice, it is often necessary to find only a single solution that satisfies all constraints. While chronological backtracking is complete algorithm it may sometimes stack at parts of the search tree and is not able to find any solution in a reasonable time. Ginsberg and Harvey [36] proposed a solution that still offers a complete algorithm but that restricts number of examined branches in the search tree. This algorithm is presented in Figure 5.7.

The algorithm restricts the number of assigned values to each of selected variables by specifying new parameter b. The search starts with $b = 1$ and therefore in the first iteration DFS algorithm artificially backtracks after trying only a single assignment to each variable. If the solution is found the whole search stops and the found solution is reported. In case of failure the algorithm continues and tries to find a solution with two possible assignments for each variable. Again if this fails it continues until all assignments for all variables has been tried and therefore it will do the full DFS search.

The analysis of iterative broadening using probability theory has been done by the authors [36]. It showed that iterative broadening can provide computational speed-up over chronological backtracking if the number of possible solutions is large., i.e., the total number of solutions is greater than $e^{b/2}$, where b is the branching factor of the search tree. Experimental studies confirmed obtained results.

Iterative broadening has several limitations. First, it can provide performance improvement for problems that require only a single solution. Second, the algorithm can still be trapped in some part of the search tree in a similar way as classical DFS. Finally, the analysis has been done based on the assumption that the search tree has a uniform branching factor. This assumption is often not met by real search trees which can often have different shapes of different subtrees.

5.2.2 Discrepancy Search

Several search methods that are based on the notion of *discrepancy* has been proposed in the literature [41, 46, 97]. All these methods try to limit the search tree size by

```
label IB($\mathcal{V}$)
  $b \leftarrow$ 1
  do
    result $\leftarrow$ breadth_bounded_DFS($b$, $\mathcal{V}$)
    $b \leftarrow b + 1$
  while result $=$ nil $\wedge\, b < \max_i(|D_i|)$
  return result

label breadth_bounded_DFS($b$, $\mathcal{V}$)
  if solver.consistency()
    if $\mathcal{V} \neq \emptyset$
      select one variable $var$ from $\mathcal{V}$
      do
        select value $v$ from $D_{var}$
        delete $v$ from $D_{var}$
        impose $var[i] = v$
        $result \leftarrow$ breadth_bounded_DFS($b$, $\mathcal{V} \setminus var$)
        if $result \neq$ nil
          return $result \cup (var, v)$
        else         // backtrack
          undo all decisions // backtrack
      while $D_{var} \neq \{\} \wedge$ less than $b$ values of $D_{var}$ has been tried
      return nil     // no more values to assign for $var$
    else
      return {}      // assignment for all FDVs found
  else
    return nil      // inconsistent constraints
```

Figure 5.7: The iterative broadening search algorithm.

examining only a number of nodes that get assigned values different than the first value selected by variable and value selection options.

Limited discrepancy search (LDS) has been originally proposed in [41] for binary search trees. The intuition behind this search method is that often our variable ordering method is good enough to order variable in such a way that a solution can be found without backtracks. However, in situations when a wrong variable assignment is made we need to backtrack and change the decisions on variable assignment. These decisions that are wrong and need to changed are called *discrepancies* and therefore the method is called limited discrepancy search since we try to limit the number of discrepancies.

The original definition of LDS has been done for binary search trees, i.e., finite domain variables have domain size two. This means that in each node of a search tree we have possibility to select at most one of two values for assignment to FDV.

LDS uses an idea of iterative search, in a similar way as iterative broadening. The first iteration, with limit of zero discrepancies, searches for a solution when no wrong decisions are accepted. The next iteration searches all possibilities with at most one discrepancy, and so on. The algorithm is shown in Figure 5.8. it is changed a little bit comparing to [41] to fit better to the formalism used in this book.

```
label LDS()
  for $i = 0..$maximum dept
    $result \leftarrow$ LDS_probe($\mathcal{V}, i$)
    if $result \neq$ nil
      return $result$
  return nil

label LDS_proble($\mathcal{V}, k$)
  if $\mathcal{V} = \emptyset$ return {}
  if $\neg$ solver.consistency() return nil
  $v \leftarrow$ select variable from $\mathcal{V}$
  $\mathcal{V}' \leftarrow \mathcal{V} \setminus \{v\}$
  if $k = 0$
    impose $v = first\_value(v)$
    $result \leftarrow$ LDS_proble($\mathcal{V}', 0$)
    if $result \neq$ nil
      return $result \cup (v, first\_value(v)$
    else
      return nil
  else   // k > 0
    impose $v = second\_value(v)$
    $result \leftarrow$ LDS_proble($\mathcal{V}', k - 1$)
    if $result \neq$ nil
      return $result \cup (v, second\_value(v)$
    undo decisions
    impose $v = first\_value(v)$
    $result \leftarrow$ LDS_proble($\mathcal{V}', k$)
    if $result \neq$ nil
      return $result \cup (v, first\_value(v)$
    else
      return nil
```

Figure 5.8: The limited discrepancy search algorithm.

The algorithm tries to search iteratively and makes assumptions on an order of assigning values to FDVs. Note that the iteration with zero discrepancies tries always the lower value in the domain (constraint $v = first\_value(v)$). In the iterations with one or more discrepancies the algorithm changes the order and tries first the higher value in the domain (constraint $v = second\_value(v)$). This leads to the search method that tries discrepancies first at the upper part of the search tree.

Figure 5.9 depicts the search tree organization of LDS search for four variables. The bold lines represent the visited paths in each iteration. The tree is fully searched after fourth iteration. If one stops search earlier, parts of the tree are not visited.

In general, if the maximal depth of the search tree is d then the number of expanded nodes with a discrepancy limit i is bounded by d^{i+1}. For iteration i, there are at most d^i fringe nodes, with each path to a fringe node expanding at most d nodes. If d is large, the cost of any single iteration dominates the summed costs of preceding ones.

The proposed LDS algorithm [41] has been originally defined for binary domains

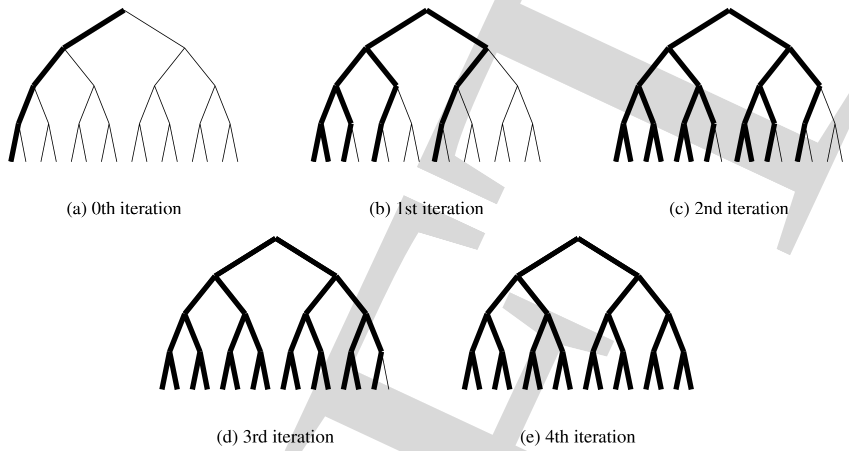

(a) 0th iteration (b) 1st iteration (c) 2nd iteration (d) 3rd iteration (e) 4th iteration

Figure 5.9: The limited discrepancy search on a binary tree of depth four.

only. Authors have pointed out that the algorithm can be extended to domains of sizes larger than two but careful considerations has to be applied. It is, for example, not clear whether one-discrepancy search should include every alternative value of the variable that violates the first assignment or only the single next most attractive value. If the number of variables is large the second option might be more realistic. LDS has been provided in CHIP solver [23] for arbitrary sizes of domains, for example.

The main drawback of LDS is that it visits some nodes several times. The iteration of i discrepancies visits all nodes of the search tree with i or less discrepancies. Several improvements to LDS has been proposed. In [46], for example, the author has proposed *Improved Limited Discrepancy Search* (ILDS).

ILDS tries to visit only those paths that have *exactly* i discrepancies. This is achieved by keeping track of the remaining depth of the search tree. If this depth is less than or equal the number of discrepancies, only the higher values in the domain are assigned (constraint $v = second\_value(v)$). Figure 5.10 depicts graphical representation of visited nodes by ILDS for balanced binary search tree.

Depth-bounded discrepancy search (DDS) is another extension of LDS method. It biases the search to discrepancies located high in the search tree. It uses a depth bound parameter to iteratively increase the depth of the discrepancy search. On the iteration $i + 1$, DDS explores those branches on which discrepancies occur at a depth i or less. Similarly to ILDS this search tries to avoid visiting nodes visited in previous iterations. This is achieved by enforcing to take only right branches at depth i on the $i + 1$ iteration. Left branches are avoided since they would lead to previously visited nodes. An example binary search tree for four variables is depicted in Figure 5.11.

5.2.3 Credit Search

Credit search is an incomplete depth first search method where search is controlled by two parameters: the value of a credit and the number of backtracks [6]. First the search uses the credit points and distributes them to different branches of the DFS tree. The credit is always represented by integer numbers and it has to be greater or equal zero. When the credit is used (its value is zero), only the number of backtracks,

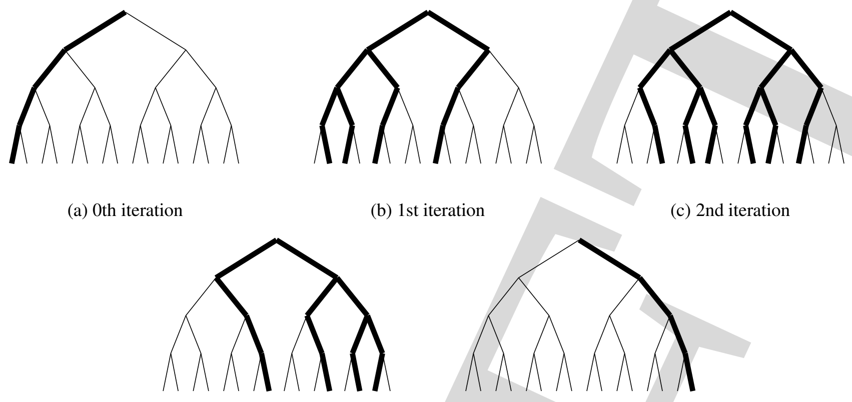

(a) 0th iteration (b) 1st iteration (c) 2nd iteration (d) 3rd iteration (e) 4th iteration

Figure 5.10: The improved limited discrepancy search on a binary tree of depth four.

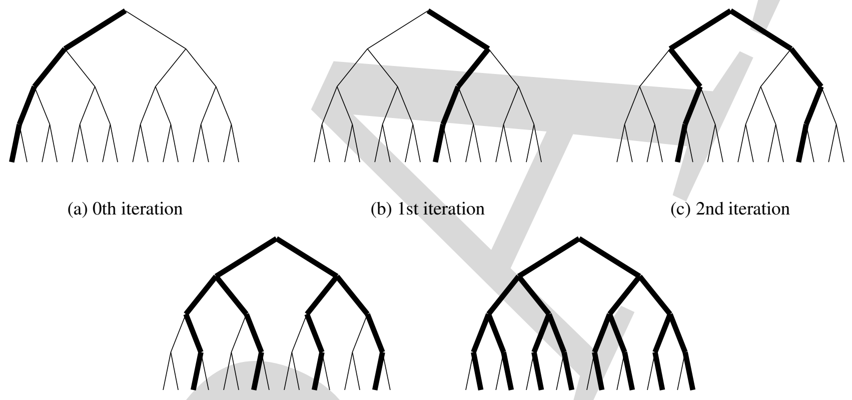

(a) 0th iteration (b) 1st iteration (c) 2nd iteration (d) 3rd iteration (e) 4th iteration

Figure 5.11: The depth-bounded discrepancy search on a binary tree of depth four.

specified by the second parameter, is allowed. This obviously artificially cuts the search space but makes it possible to explore the parts of the search tree that are impossible in practice otherwise (because of long execution times). Of course, in general, this method cannot be used to prove that a solution is optimal since it does not check all possible assignments.

An example of the credit search tree is depicted in Figure 5.12. The search has initially 8 credits and the distribution is specified by 1/2 indicating that half of the credits are distributed to the selected choice. The number of possible backtracks is three. The first part of the search is based on the credits and makes it possible to investigate many possible assignments to domain variables while the other part is supposed to lead to a solution. Since we control the search it is possible to partially explore the whole tree and avoid situations when the search is stuck at one part of the tree which is a common problem of DFS algorithm.

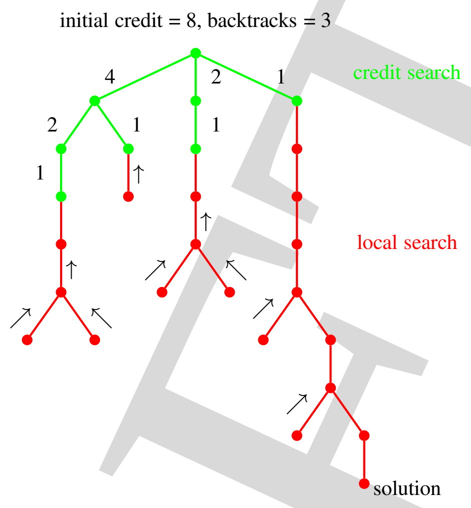


Figure 5.12: Credit search example.

The credit search algorithm represents a typical incomplete search method that is based on heuristic assumptions. It is based on an intuition that most important decision are done at the beginning of the search tree and most search is needed there. Therefore the search based on credit distribution is done first. Later only a small number of backtracks is allowed. It is difficult to decide what is the best way to assign credit, its distribution and number of backtracks. There are, however, some indications that can help to make a good enough decision.

In an unconstrained search tree and with the number of backtracks equal zero, the credit is equivalent to the number of leaf nodes that will be reached. The number of leaf nodes in a search tree grows exponentially with the number of variables. Since we would like to have polynomial runtimes we would like to enforce a credit number that will restrict number of choices to be polynomial of the number of variables. Therefore the good rule of thumb is to use the credit value that is equal the number of variables squared or cubed, thus enforcing polynomial runtime.

5.3 Optimization

An important part of constraint programming is optimization. In this section we will discuss methods to do minimization of a given cost function but maximization can be done in a similar way. We assume that there exist in our model a single FDV that represents the cost for minimization. It can be, for example, the completion time of our application that need to be minimized to obtain the shortest schedule. Constraint programming traditionally does optimization by using branch and bound (B&B) algorithm.

The B&B algorithm reduces the search space by cutting solutions that cannot provide a better cost value. This is done based on cost estimation that comes automatically from the solver when performing consistency enforcement. In practice, the B&B algorithm extends DFS algorithm by adding the following rule. When a solution is found, a new constraint is added indicating that the optimal cost must be strictly less than the current solution cost (constraint of a form $cost < currentCost$). Therefore the DFS algorithm will automatically fail and will backtrack when a solution with the possible higher cost is to be generated. Notice that the partial solution can have already cost that is strictly higher than the previous one and can generate fail.

The search can either be restarted from the beginning with this additional constraint or it can continue from the point where a solution has been found. In both cases if no other solution is found, then the best so far cost is the optimal one. A search can incorporate different DFS strategies, both complete and incomplete. This provides opportunity to find good solutions in many cases but if complete search cannot be achieved the proof of optimality is not available any longer.

```
label minimize($\mathcal{V}$, $cost$)
$result \leftarrow$ nil
$solution \leftarrow$ nil
do
  $result \leftarrow$ DFS($\mathcal{V}$)
  if $result \neq$ nil
    $solution \leftarrow result$
    impose $cost < valueOfCost(result)$
while $result \neq$ nil
return $solution$
```

Figure 5.13: The optimization algorithm with restarts and DFS algorithm.

Figure 5.13 shows a simple algorithm that implements DFS based minimization algorithm with restarts. The algorithm iteratively calls DFS method. Each new iteration, except the first one, is run with the new constraint on cost FDV. In this way we obtain always solutions with the lower value of the cost variable. When the last iteration fails to find a solution the previously found solution is optimal. If does not exist any previous solutions then the algorithm return `nil` which means that no solutions exist for given set of constraints.

The minimization algorithm presented in Figure 5.13 uses DFS method to find consecutive solutions. This method can be easily replaced by any of incomplete search methods, such as LDS or credit search. In such cases, we can still minimize the cost function but we do not have guarantee that the final solution is optimal.

One can further extend the minimization method by adding additional parameters and related actions. Typically, minimization methods can have specified a lower bound and a solution improvement factor. If the lower bound value is specified the minimization method will exit the `do-while` loop as soon as the current value of the cost variable is equal the specified lower bound. In this way, we avoid additional search that proves optimality. This search is often very time consuming. The solution improvement factor specifies that any new solution must be at least a given percent better than the previous one. This parameter can help in avoiding solutions with only minor changes in the cost value.

5.4 Multi-objective Optimization

During design process one often wants to explore different design alternatives and select the one that suits given requirements best. Design space exploration can therefore be defined as a process of finding different solutions that provide trade-offs between design parameters. It is usually achieved by doing a multi-objective optimization, i.e., the optimization that simultaneously optimizes more than one cost function. In such cases it does not exist a single optimal solution but instead we introduce a notion of *Pareto optimality*.

To define multi-objective optimization more formally we assume that there exist several optimization criteria or cost functions denoted by $f_i(x), i = 1, \ldots, n$. We define Pareto optimal solution as vector x, if all other vectors y have a higher value for at least one of the cost functions, i.e., $\forall_y \exists_i f_i(y) > f_i(x)$, or have the same value for all cost functions. Each vector x that is Pareto optimal is also called *Pareto point*. Pareto points form a trade-off curve or surface called *Pareto curve* or *Pareto surface*. Pareto curve and Pareto surface indicate the nature of the trade-off between different optimization objectives. Pareto point is also called *non-dominated* or *non-inferior* point since it does not exist any other point that is better in respect to at least one criteria.

The graphical representation for Pareto points is depicted in Figure 5.14. It shows the situation when we have two optimization criteria, cost and performance. Four points representing four different solutions are depicted. Points A and B are Pareto points since they are not dominated by any other point. Point D is not a Pareto point since it has both cost and time greater than any other point. Finally, point C is not a Pareto point since it is dominated by point B that has better cost for the same time value.

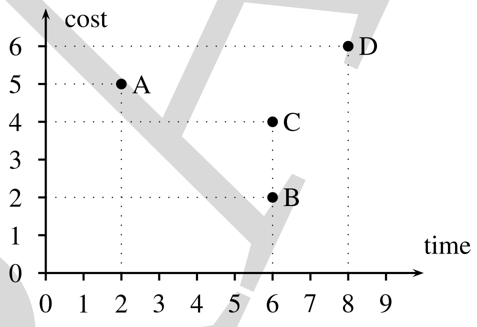


Figure 5.14: Pareto points example.

The generation of Pareto points can be nicely formalized using constraints. The idea is to start search with most relaxed constraints. When a solution is found additional constraints are imposed. These constraints cut the part of the search space that can only have dominated points. Consider two dimensional space depicted in Figure 5.14 and assume that the solver found a solution with cost c_i and execution time et_i. The new constraint forbids to find solutions with greater cost or greater execution time and therefore constraint $cost < c_i \lor time < et_i$ is imposed. This prohibits to find dominated points in future but does not assure that the points that has been found before are not dominated by newly found point. Therefore it is necessary to remove from the list of candidate Pareto points such dominated points. The whole procedure is depicted in Figure 5.15. It iteratively executes depth first search method. After each execution the found point is added to the set of Pareto points and a constraint that cuts dominated points is imposed. Dominated points are also removed from the set of created Pareto points, if such exist.

```
vector pareto($\mathcal{V}$, $criteria$)
$paretoPoints \leftarrow$ nil
while DFS($\mathcal{V}$) $\neq$ nil
  $paretoPoints \leftarrow paretoPoints \cup (val_0, \ldots, val_n)$
  impose $criteria_0 < val_0 \vee \cdots \vee criteria_n < val_n$
  remove points dominated by $(val_0, \ldots, val_n)$ from $paretoPoints$
return $paretoPoints$
```

Figure 5.15: The Pareto points generation algorithm.

Consider, as an example, search for Pareto points depicted in Figure 5.14. Figure 5.16 shows consecutive steps during search for Pareto points. First, point C is found and the imposed constraint restricts search space in such a way (gray zone) that point D is cut from reachable points. Next point A is found and in this stage points A and C are current Pareto points. Note that point C is not a Pareto point and in the next step, when point B is found, it is removed from Pareto points set.

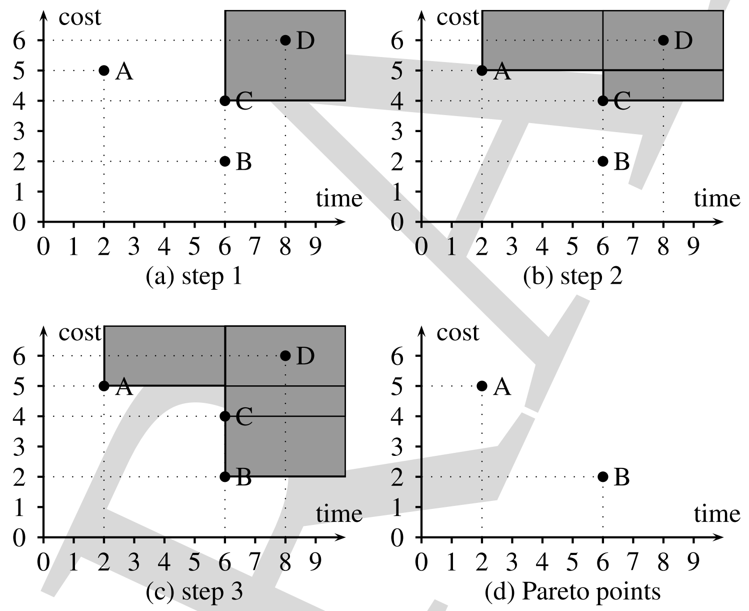


Figure 5.16: The Pareto search steps for example 5.14.

Example 5.6: Consider Example 4.5 of differential equation solver. We optimize three cost functions: number of adders, number of multipliers and number of execution cycles. The algorithm proposed in Figure 5.15 generates the Pareto points depicted in Table 5.1.

For realistically large examples the algorithm proposed in Figure 5.15 has problems to complete the computation in a reasonable time and therefore heuristic methods must be applied to explore only a sub-set of the design space to derive an acceptable

Table 5.1: Pareto points for differential equation solver.

Number adders	Number multipliers	Execution cycles
1	4	6
2	3	6
1	3	7
2	2	7
1	2	8
1	1	13

solution in terms of search time and accuracy. Incomplete search methods can be used for this purpose and method DFS can be replaced by a selected incomplete search method, such as credit search or discrepancy search.

5.5 Advanced Search Methods

In this section we will discuss advanced search methods that are often developed for specific applications. The search methods discussed so far in this chapter were build on the principle that the decisions are made which value is assigned to a FDV. Many problems do not need to assign the values to FDVs but instead they can be solved by defining specific constraints. For example, for job-shop scheduling discussed in section 4.2.6 it is enough to define the order of tasks assigned to each machine. Knowing this order one can find a valid assignment, without backtracking.

In this section we will use job-shop scheduling and build a special search method that can be used to solve difficult job-shop scheduling problems. The constraints that define job-shop scheduling problem have been introduced in section 4.2.6 and comprise precedence constraints for each job and cumulative constraints for each machine. We will also assume that a maximum constraint provides us with the cost variable that defines the schedule makespan.

Th search algorithm for job-shop scheduling is based on the method presented in [4] and it is depicted in Figure 5.17. It basically selects first a machine that has not ordered tasks and makes an ordering decision. This machine can be selected based on different criteria. In the presented algorithm we use criteria, suggested in [4], that selects machine with the minimal slack, i.e. machine that has the smallest difference between so called supply and demand. The supply is the period when the tasks can be scheduled and it is measured as the earliest release time for all tasks and the latest completion time. The demand is the sum of tasks durations. Once the machine and its tasks are selected the algorithm sorts tasks using their earliest release time and then using this order makes decisions which task need to be executed first. Basically the first task is ordered as the first among tasks on the list and related precedence constraints are imposed. On backtracking the algorithm selects the next task to be executed first and so on. If all tasks on all machines are ordered the algorithm found a total ordering of tasks and basically solved the problem. The exact time for starting a particular task is not yet defined but it can be determined by additional DFS search. This search finds a solution without backtracking.

It can be noted that the strength of the search algorithm presented in this section is supported by the consistency methods implemented in `cumulative` constraints.

```
// $\mathcal{M}$ is a vector of vectors representing tasks assigned
// to a machine. Each task is specified by its starting
// time  (FDV $t$) and task duration ($d$)
label Jobshop_Search($\mathcal{M}$)
  if solver.consistency()
    if $\mathcal{M} \neq \emptyset$
      $m \leftarrow$ selectCriticalMachine($\mathcal{M}$)
      sort tasks in $m$ in ascending values of $t.min()$
      for each $i = 1, \ldots, n$
        for each $j = 1, \ldots, n$
          if ($i \neq j$)
             impose $m_i.t + m_i.d \leq m_j.t$
        $\mathcal{M}'$ $\leftarrow$ $\mathcal{M} \setminus m_i$
        if Jobshop_Search($\mathcal{M}'$)
          return true
        else
          return false
      return false
    else
      report solution
      return true
  else
    return false

vector selectCriticalMachine($\mathcal{M}$)
  for each $m_i \in \mathcal{M}$
    min $\leftarrow \min(\min(m_i.t_0), \min(m_i.t_1), \ldots, \min(m_i.t_n))$
    max $\leftarrow \max(\max(m_i.t_0 + m_i.d_0), \max(m_i.t_1 + m_i.d_1), \ldots,$
           $\max(m_i.t_n + m_i.d_n))$
    supply $\leftarrow$ max - min
    demand $\leftarrow$ $\sum m_i.d_i$
    critical $\leftarrow$ supply - demand
  return machine $m_i$ with the lowest critical value
```

Figure 5.17: Search algorithm for job-shop scheduling.

The edge-finding algorithm propagates information when a decision that a task is scheduled before all other tasks is made. This can in many cases detect inconsistency as well as narrow domains of FDVs efficiently.

Example 5.7: Consider a simple job-shop scheduling benchmark known as Fisher and Thompson 6x6 instance or MT06. This problem has 6 jobs that are scheduled on 6 machines and is specified in Table 5.2. Tasks are specified in the table by defining machine for its execution (m_i) and tasks execution time (d_i). The optimal schedule obtained with the proposed scheduling algorithm is depicted in Figure 5.18. It has length 55 and it has been proven to be optimal.

The problem used in Example 5.7 is relatively simple and can be solved using

Table 5.2: Fisher and Thompson 6x6 job-shop instance (MT06).

Job	m_1	d_1	m_2	d_2	m_3	d_3	m_4	d_4	m_5	d_5	m_6	d_6
1	2	1	0	3	1	6	3	7	5	3	4	6
2	1	8	2	5	4	10	5	10	0	10	3	4
3	2	5	3	4	5	8	0	9	1	1	4	7
4	1	5	0	5	2	5	3	3	4	8	5	9
5	2	9	1	3	4	5	5	4	0	3	3	1
6	1	3	3	3	5	9	0	10	4	4	2	1

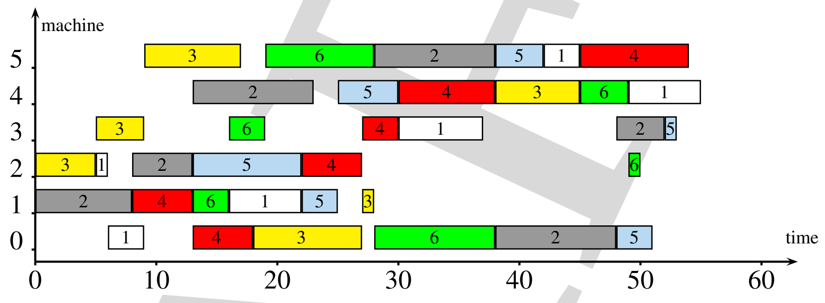


Figure 5.18: Gantt diagram for Example 5.7

standard DFS methods discussed in previous sections. However, there exist many job-shop examples that cannot be solved with simple DFS search with domain enumeration. One of the well known example is the instance known as MT10 (10x10 job-shop problem). This instance was proposed in 1964 and it was open until 1989. It can be solved with the proposed algorithm in a number of minutes on current computers. This is possible since the search algorithm gets strong support from propagations produced by cumulative constraint that implements edge-finding algorithm.

Figure 5.19 depicts how the search for MT10 benchmark progresses. It is interesting to indicate that the algorithm finds good quality solution quite quickly and then it searches very much to improve it. Finally, when the optimal solution is found after 243 s the algorithm needs another 208 s to prove that this solution is optimal. This observation is typical for job-shop that has been also pointed out by other authors. Therefore heuristics have been proposed to improve execution time for the job-shop search algorithms [3].

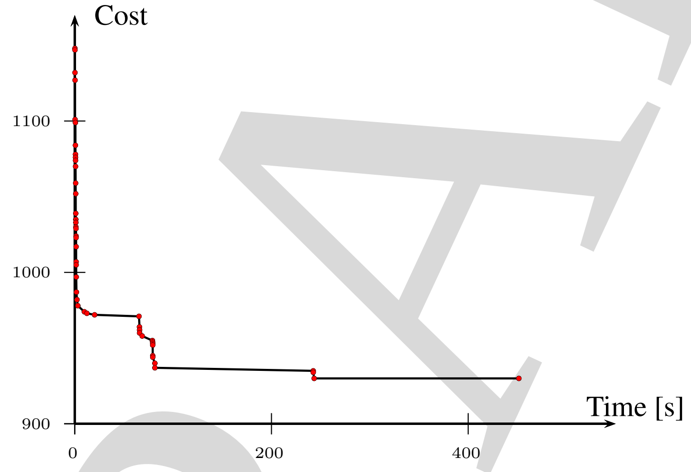


Figure 5.19: Found solutions for the MT10 problem in function of time.

Six

Advanced Techniques

In this chapter we will shortly discuss some of issues of constraint programming that are very important for practical applications but are still heavily researched. We will concentrate first on soft constraint, i.e., constraints that preferably should be fulfilled in the final solution but their fulfillment is not mandatory. Then we will discuss symmetry breaking since it provides opportunity to drastically reduce the search tree and make constraint programming more effective. Then we will also discuss how to combine integer linear programming methods and local search methods with constraint programming to get more effective tools for solving complex problems.

6.1 Soft Constraints

Constraint that are defined for many practical problems can often be classified as *hard* and *soft* constraints. The hard constraints must be satisfied in all solutions. Soft constrained, on the other hand, may be violated in admissible solution. A constraint programming solver is well suited to handle hard constraints. All constraints added to the constraint store are treated in the same way, i.e., the consistency of a conjunction of all constraints is enforced. This does not fit soft constraints since some of them might be fulfilled and some not. Moreover it could be good to have a way to "maximize" the satisfaction of soft constraints.

In practice, soft constraints are usually handled using a kind of test of the constraint entailment (e.g., reified constraint) and optimization of satisfaction criteria for soft constraints. This is achieved by construction of the cost function in a such way that it reflects the goodness of satisfaction of soft constraints. This method works pretty well but in general it does not use the power of consistency enforcement and propagation of results. Soft constraints are passively watched whether they are satisfied or not and the optimization tries to direct the search into solutions that satisfy soft constraints best. An example of a possible formulation of soft constraints is presented below.

Example 6.1: Consider task assignment problem presented in Example 4.1. This example has been formulated to assign tasks to processors while minimizing the execution time for the whole set of tasks. Assume that we would like to have the optimal execution time of 3 but we would like to assign some tasks to the same processor. This might be preferable because of reliability issues, for example. To enforce the assignment of tasks to the same processor we can add hard constraints but let us consider that we add three soft constraints implying our preferences that tasks pairs $(1, 2)$, $(7, 9)$, and $(2, 5)$ should be if possible assign to the same processor. To achieve this we add the following three reified constraints.

impose $(r_1 \neq r_2) \Leftrightarrow b1$
impose $(r_7 \neq r_9) \Leftrightarrow b2$
impose $(r_2 \neq r_5) \Leftrightarrow b3$

Therefore if $b_i = 1$ it means the the related tasks are assign to different processors. We also add the constraint $cp = 3$ to enforce all assignments with execution time equal 3. To get the solution with the maximal number of satisfied constraints we minimize the cost function that is the sum of all b_i (i.e., $b_1 + b_2 + b_3$ in our case). It represents the number of violated constraints. The solver finds a solution which has $b_1 = 0, b_2 = 0$, and $b_3 = 1$ that means that tasks pairs $(1, 2)$ and $(7, 9)$ are assigned to the same processor while tasks 2 and 5 are assigned to different processors. In this way we have minimize the number of not satisfied soft constraints.

Further refinements of this approach to soft constraint definition are possible. For example, one can prioritize constraints by specifying a weighted sum of b_i. Therefore satisfaction of constraints with higher weight has larger effect on optimized cost. This however does not remove the main problem of the method that is the limited propagation between soft constraints.

The general idea of this approach was to assign violation cost that represent the measure of the quality of the final solution. Minimization of the violation costs produces therefore the best solution with most soft constraints satisfied. This simple formulation gives opportunity to propose specific filtering methods for soft constraints. They are based on the assumption that we can propagate a specific cost value that need to be achieved to prune domains of variables involved in soft constraints. For example, the formulation of Example 6.1 requires that all inequality constraints need to be satisfied if $cost = 0$ and therefore the solver enforces their consistency. More complex methods can be proposed for specific combinatorial constraints. Authors of [68] propose, for example, a soft different constraint formulation based on this approach.

6.2 Symmetry Elimination

Many problems solved using constraint programming methods have many solutions that are not unique, i.e., one can obtain one solution from another one by a transformation. These solutions are equivalent even they have different assignment to their FDVs, one solution can be obtained from the other by a specific mapping. The situation is similar to the chess board that is rotated by 180°. This “new” chess board looks exactly the same as the one before rotation while its fields are numbered differently. This simple example illustrates a symmetry in the chess board and how a new board can be obtained from the original one by rotation.

Consider Example 4.1 and an optimal bin assignment with the cost 20 that is depicted in Figure 4.1. This solution has been obtain by the following assignments: $r_1 = 1, r_2 = 1, r_3 = 2, r_4 = 1, r_5 = 0, r_6 = 2, r_7 = 0, r_8 = 2, r_9 = 1, r_{10} = 0$. Since all processors are identical we can obtain 6 (3!) symmetrical solutions to the given one by permutating processors p_1, p_2 and p_3. This is equivalent to exchanging all tasks between processors. Figure 6.1 shows all six symmetrical solutions to the

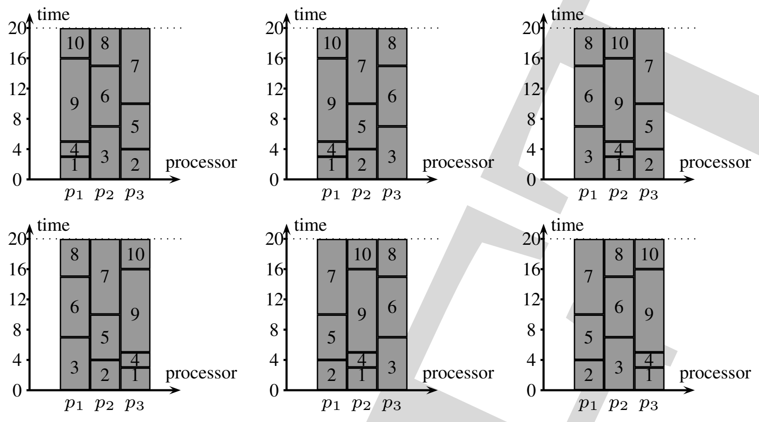


Figure 6.1: Different symmetrical assignments of tasks to three processors for Example 4.1

.

solution defined above. It is enough to find a single representative solution for all six symmetrical and equivalent solutions.

Symmetries can be either inherited in the problem or introduced into the problem because of the modeling style. They cause that there exist many symmetrical solution and thus the size of the search tree becomes very large. It can be difficult to prove optimality of solutions when many symmetrical solutions have to be visited. It is therefore a good reason to either avoid modeling styles that introduces symmetries or find the ways to avoid repeatedly visit symmetrical solutions.

Formally, symmetries in CSP are defined using one to one mapping that preserves solutions, i.e., that maps solutions to solutions and non solutions to non solutions [73].

Definition 7 (Symmetry [73])**.** *A symmetry σ for CSP $\mathcal{S} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is a one to one mapping (bijection) from decisions to decisions of $\mathcal{S}$ s.t.*

i) for all assignments, $A = \{v_i = d_i\}, \sigma(a) = \{\sigma(v_i = d_i)\}$

ii) for all assignments A, $a \in sol(\mathcal{S})$ iff $\sigma(A) \in sol(\mathcal{S})$

where $sol(\mathcal{S})$ denotes the set of all solutions of $\mathcal{S}$.

Consider symmetries defined in Figure 6.1. The symmetrical solution depicted on the second sub-figure can be obtained from the first one by the following bijection $\sigma(r_i = 1) = (r_i = 1), \sigma(r_i = 2) = (r_i = 3), \sigma(r_i = 3) = (r_i = 2)$, i.e. swap tasks between processors 2 and 3. This illustrates the case of *value symmetry*. CSP problems can also have *variable symmetries* or both.

There exist four main approaches to symmetry breaking [33]:

- reformulate the problem,
- add symmetry breaking constraints before search,
- adapt search algorithm to break symmetry, and
- build search tree so that no symmetry arises.

The first two methods assume that we understand the problem and its symmetries and are able to either define the problem in such a way that it has no symmetries or we can break symmetries by adding additional constraints. The last two methods, on the other hand, require changes in search. The search has to be organized in such a way that symmetrical solutions will not be explored and the search in this parts of a search tree will be cut. The research in this area has made a big progress during last ten years and many interesting results have been obtained. In particular, a very interesting connection between computational group theory (CGT) and symmetry breaking has been observed and used in practice.

6.2.1 Problem reformulation

In some cases symmetry can be eliminated or reduced by using different modeling style. This is only possible for specific problems and is limited by constraints that are available in our constraint system. For example, bin packing problem discussed in the previous section exhibits symmetry since we use three identical processors (bins) and they were artificially distinguished by their numbers. If one can have a constraint that can make partitioning to n identical bins and do not make any numbering of these bins, the problem could be modeled without symmetries. In this case we would probably need some kind of set partitioning constraints that are defined for sets. This is not always possible and obviously our bin packing problem cannot be easily modeled in this way with existing constraints.

6.2.2 Symmetry breaking constraints

The method for symmetry breaking that is used based on additional constraints is probably one of the first approaches proposed to eliminate symmetries from constrain models. It assumes that the programmer understands symmetries existing in our model and can add special constraints that prevent the solver from exploring symmetrical solutions. This obviously requires good insight to the problem. The method might require many additional constraint and therefore combinatorial constraints based on lexicographical order have been proposed. They can be easily used for symmetry breaking purpose as it will be indicated later in this subsection.

The method based on lexicographical order uses ordering of solutions. It is based on the principle that a canonical solution is selected and its symmetrical variants are lexicographically greater or equal to this solution. If the ordering is defined we can eliminate symmetrical solutions that are greater or equal to our canonical one. In this way the search will only explore solutions that are canonical.

Example 6.2: Consider Example 4.1 and all symmetrical solutions with cost 20 as depicted in Figure 6.1. The solution can be represented as a matrix over tasks and processors where 1 on position i, j means that task j is assigned to processor i.

Task	1	2	3	4	5	6	7	8	9	10
p_1	1	0	0	1	0	0	0	0	1	1
p_2	0	0	1	0	0	1	0	1	0	0
p_3	0	1	0	0	1	0	1	0	0	0

Symmetric solutions can be easily generated by permutating rows of the matrix and therefore we have six different symmetric solutions. One way to restrict generation of symmetric solutions is to order rows by lexicographical order. We introduce new relation $\leq_{lex}$ that orders vectors of integers. For example, $[0, 1, 2] \leq_{lex} [1, 2, 3]$. If constraints $row_1 \leq_{lex} row_2, row_2 \leq_{lex} row_3$ that only one symmetrical solution will be generated and all other will be rejected.

In our case, we do not have a constraint that enforces lexicographical order on rows of the solution and therefore we can use arithmetical constraint to do it. We simply obtain values $b_{i,j}$ using reified constraints of the form $r_i = j \Leftrightarrow b_{i,j}$ and then impose constraints.

$$\begin{aligned} &b_{1,10} \cdot 2^0 + b_{1,9} \cdot 2^1 + \cdots + b_{1,1} \cdot 2^9 = v_1 \\ &b_{2,10} \cdot 2^0 + b_{2,9} \cdot 2^1 + \cdots + b_{2,1} \cdot 2^9 = v_2 \\ &b_{3,10} \cdot 2^0 + b_{3,9} \cdot 2^1 + \cdots + b_{3,1} \cdot 2^9 = v_3 \\ &v_1 \leq v_2, v_2 \leq v_3 \end{aligned} \tag{6.1}$$

Adding these constraints to our problem definition reduces the number of solutions by factor of 6, from 72 to 12.

Lexicographical ordering constraints are often implemented as combinatorial constraints and are available in a number of constraint programming systems, see for example constraint `lex_chain` available in SICStus [83]. They have linear computational complexity.

6.2.3 Adaptation of search algorithm

The method assumes that symmetries existing in our problem can be formally defined. This information can then be used by a search algorithm that it will not visit parts of the search tree that perform symmetrical assignments. One can distinguish between a sub-tree that already produced a solution and therefore the symmetrical sub-tree do not need to be visited and a sub-tree that failed and its symmetrical sub-tree will also fail and therefore do not need to be visited.

There are basically two methods to implement this approach. The first one uses constraints that are added during search. These constraint rule out visiting equivalent nodes in future.

The second methods is built based on the dominance detection. Every node is checked before entering it and it is not entered if an equivalent node has been visited before. Both methods implement in some sense computational group theory (CGT) methods to detect dominance. A method that uses CGT for symmetry breaking has been implemented using ECLiPSe and GAP (CGT system) [34]. A user provides definition of symmetries existing in the problem and the GAP constructs a symmetry group, performs search for dominating elements when required, and provides the ECLiPSe with the information needed to restrict search.

6.2.4 Heuristic methods for building search trees

The methods in this group relay on the search procedure that is constructed in such a way that no symmetrical solutions will be visited. Authors of [93], for example, discuss a simple yet complete methods for breaking value symmetries in a number of classes of CSP problems. Their symmetry breaking method performs in constant time and space during search.

One of the CSP classes identified by the authors are value-interchangeable CSPs where graph coloring problem belongs. The main idea is to limit the number of values that has to be explored by the search procedure and avoid symmetries. This is achieved by trying the "representative" values that do not rule out the solution but break symmetries. For the graph coloring problem, the rule simply defines how to color vertex v (assuming that we have n colors that are ordered).

- color v with one of colors that are already used, or
- color v with an arbitrary color that has not yet been used.

This method breaks all $n!$ symmetries. A similar method has been used in [27] for breaking symmetries for processor assignment problem. The author proposes a method that assigns tasks to processor based on similar criteria used by graph coloring. First tasks are assigned to already used processors and if this fails the search procedure tries to assign the task to one of several identical processors only.

6.3 Connection to Integer Linear Programming

Linear programming (LP) is an approach that defines optimization problems as a set of constraints and an optimizations criteria. The constraints are defined as inequalities over linear terms and the cost function as a linear term. The goal is to find a solution that minimizes or maximizes the cost function and fulfills all defined constraints. Formally it can be defined as follows.

Definition 8 (Standard form of Liner Programming)**.** *The linear programming standard form is defined as*

$$\min\ z = \sum_{j=1}^{n} c_j x_j$$

subject to

$$\begin{array}{ll} \sum_{j=1}^{n} a_{ij} x_j = b_i & i = 1..m \\ x_j \geq 0 & j = 1..n \end{array}$$

where x_i are variables to be solved and a_{ij}, c_j and b_i are known coefficients.

While Standard Form requires that all variables are non-negative and only equalities are allowed as constraints, most LP tools allow to define the problem using inequalities and variables that are not limited to non-negative values. Such problems are transform internally, in a solver, to Standard Form using simple transformations. Basically the problem is rewritten using the following rules [56].

- each x_i variable that is not non-negative is replaced by expression $x_i^+ - x_i^-$, where x_i^+ and x_i^- are two new non-negative variables,
- each inequality of the form $e \leq r$ where e is a linear expression and r is a number can be replaced with $e + s = r$ where s is a new non-negative *slack* variable.

It is interesting to note that the linear programming problems are solvable in polynomial time. Many solvers use, however, the simplex algorithm that has an exponential complexity in the worst case. Such cases seem never to be encountered in practical applications.

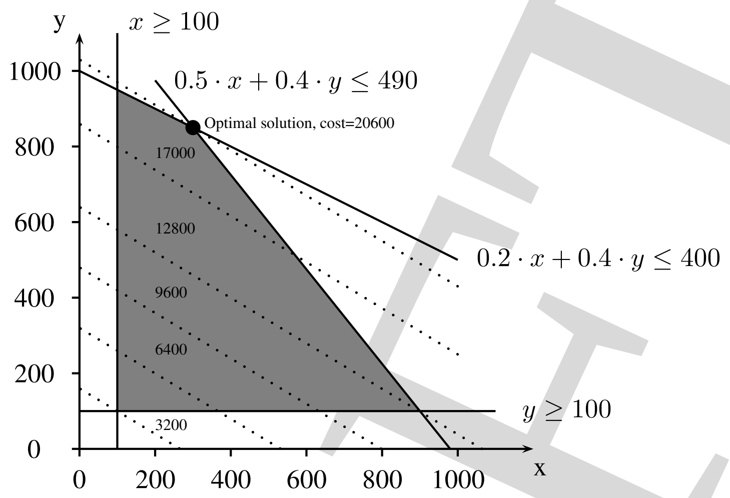


Figure 6.2: Graphical represnetation of the LP formulation from Example 6.3.

Example 6.3: Consider a simple LP formulation below.

$$\begin{aligned}
&\max\ 12 \cdot x + 20 \cdot y && (6.2)\\
&\text{subject to}\\
&0.2 \cdot x + 0.4 \cdot y \leq 400\\
&0.5 \cdot x + 0.4 \cdot y \leq 490\\
&x \geq 100\\
&y \geq 100
\end{aligned}$$

This LP formulation has very clear graphical interpretation that is depicted in Figure 6.2. All linear inequalities limit the space for possible solutions. In Figure 6.2 this space is depicted as the gray area. All points within this area and on the edges of the lines are possible solutions to the imposed constraints. The optimal solution has been found to have cost 20600. It has $x = 300$ and $y = 850$. It is located in one of the corners of the polygon. Selected values of the cost function are also depicted in the figure as doted lines.

Example 6.3 illustrates a property of linear programming. The set of constraints defines a polytope (in Example 6.3 it is a polygone in two dimensional space). The optimal solution is located in one of its vertices. Therefore the simplex method starts from one of the verticies and moves to an adjacent one with a better value of the cost function. The detail description of the simplex algorithm can be found in many books. A good description with examples is included, for example, in [56].

Integer linear programming (IP) is a subclass of the linear programming where the decision variables are limited to take integer values. It basically adds additional constraint to the problem that constraints all x_i variables to be integers. If binary decision variables are used instead of integer ones, the problem is called 0/1 linear programming problem.

A simple method for solving IP problem can be constructed using simplex algorithm together with branch-and-bound search. More advance techniques can use, for example cutting planes to be able to improve efficiency of the solver. The method based on branch-and-bound search uses simplex algorithm to find first a solution to the relaxed LP problem. Then the simplex algorithm is used to solve a number of constrained problems, i.e., problems with some variables constrained to be integer. This process is controlled by branch-and-bound algorithm. The branch-and-bound algorithm tries to restrict the value for each variable to be integer. It is achieved by defining branching on two values, either $x_i \leq \lfloor v_i \rfloor$ or $x_i \geq \lceil v_i \rceil$. After adding one of these additional constraints the simplex algorithm tries to find a solution. The algorithm can fail and this indicates that the constraint defines infeasible solution. It can also deliver a new partial solution with a new value of the cost function. In this case, if we have a previously obtained cost function value, we can either continue the search (the new cost is lower than the previous cost) or cut the search (the new current cost is higher than the previous cost).

Example 6.4: Consider a simple IP formulation below.

$$
\begin{aligned}
&\text{min } 7 \cdot x_1 + 12 \cdot x_2 + 5 \cdot x_3 + 14 \cdot x_4 \\
&\text{subject to} \\
&300 \cdot x_1 + 600 \cdot x_2 + 500 \cdot x_3 + 1600 \cdot x_4 \geq 700 \\
&x_1 \leq 1 \\
&x_2 \leq 1 \\
&x_3 \leq 1 \\
&x_4 \leq 1 \\
&x_1, x_2, x_3, x_4 \text{ non negative integer}
\end{aligned}
\tag{6.3}
$$

Since we constraint all x_i to be 0 or 1 we have, in fact here 0/1 integer linear programming problem. The simplex algorithm finds solution $x_1 = 0, x_2 = 0, x_3 = 0,$ and $x_4 = 0.44$. This solution obviously does not fulfill the constraint that the values must by non negative integers and therefore the branch-and-bound search is performed. It is depicted in Figure 6.3. The branch-and-bound search tries two branches for each variable. Since $0 \leq x_i \leq 1$ the choice is made between $x_i = 0$ and $x_i = 1$. After each choice the simplex algorithm is run and a new solution is obtained. In two cases there is no solution that is indicated as *infeasible*. After finding the first solution with cost 12 the search is cut for solutions that indicate cost 13 or 14.

There exist a number of similarities and differences in optimization process performed by CP and IP approach. In CP each time a feasible solution with cost C is found, a constraint on cost function $cost < C$ is added. Since variable $cost$ is linked to problem variables, propagation is performed. At each node, when variables are instantiated and propagation is performed, bounds of $cost$ are updated. In IP, at each node the LP relaxation is solved providing a lower bound on the problem. If the lower bound is worse than the current upper bound, the node is not further extended. Otherwise, a new non integral variable x is selected and the branching is performed on its bounds. Furthermore, an initial upper bound can be in general computed. In Example 6.4 it had value 6.12, for example.

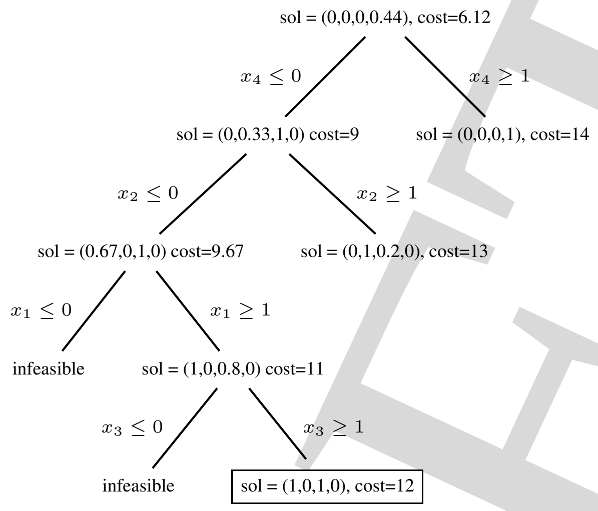


Figure 6.3: Branch and bound search tree for Example 6.4.

There exist a number of reasons to integrate CP and IP. They use similar solving methods but they are based on different mathematical principles. These methods can complement each other and provide a stronger environment for solving complex optimization problems. Obviously, the main motivation for integrating modeling and solving techniques from CP and IP is to combine the advantages of the two approaches. CP offers flexible modeling capabilities and good interaction among constraints. This is provided by a rich set of constraints, in particular by combinatorial constraints that offer good modeling power and efficient pruning capabilities. IP on the other hand provides global reasoning on optimality and efficient solution methods (polynomial time algorithms for solving LP problems and advance methods for IP, e.g., cutting planes). Such an integration can try to overcome the limitations of both approaches, such as poor reasoning on the cost function in CP or not flexible models and lack of symbolic constraints in IP.

The work on integration of CP and IP is still in progress and a number of techniques have been proposed [63]. There are no general guidelines to know in advance which technique is the most appropriate but there exist two main approaches. Based on examples from literature the authors of [24] discuss two generic cooperative schemes: *decomposition* scheme and *multiple search* scheme. These schemes can be defined for combining CP and IP as well as CP with other optimization techniques, such as local search and operations research algorithms.

The decomposition scheme assumes that the original problem is decomposed into a number of sub-problems and applies a “slave” solver to a set of these sub-problems. The sub-problem can be a reduced problem with less variables, a relaxed problem, a problem that is an approximation of the main problem or a tightened version of the main problem. These sub-problems can be solved by different techniques and provide partial results. These partial results can then be collected by a master and used to reduce the search space and guide the search for the original problem. Most often the “slave” solvers provide a new tightened bounds for variables.

The well known decomposition in CP copes with linear sub-problems. The linear inequalities and possibly a cost function can be passed to LP linear solver for narrowing domains of FDVs as well as obtaining better lower bounds on the cost function. In such case, CP solver works in a same way as before but has possibility to use the strength of LP methods.

Example 6.5: Consider that our solver accepts constraints defined as linear inequalities over integers and provides specific methods for consistency for these constraints. The following constraint has been defined for the purpose of this example.

$$\begin{aligned} C = \{ & \\ & -3x_1 + 2x_2 \leq 0, \\ & 3x_1 + 2x_2 \leq 6, \\ & x_1 \leq 2, x_2 \leq 2, \\ & x_1 \geq 0, x_2 \geq 0, \\ & \text{integral}([x_1, x_2]) \\ & \} \end{aligned} \tag{6.4}$$

Applying traditional bounds consistency produces the following domains $x_1 :: 0..2, x_2 :: 0..2$.

However if a consistency method for constraint C is based on the simplex algorithm it can narrow domains further. It detects, for example, by maximizing x_2, subject to the linear inequalities over the rational numbers, that the upper bound of x_2 is 1.5 and thus the maximal value in the domain of x_2 can be reduced to 1. Therefore we can infer the primitive constraint $x_1 \leq 1$ and add it to the constraint store.

The multiple search scheme assumes, in principle, that two or more solvers are used to solve the original problem. The solvers are run in parallel and might communicate their results. A framework where IP and CP solvers cooperate and exchange information on the bounds of a cost function and decisions variables has been presented in [65]. The author has combined IP and CP models by adding a method for communication between both models. This has been implemented as a special propagator that connects LP model constraints with finite domain constraints. Therefore, if any of the solvers found new bounds for variables these bounds are propagated to another solver. Since the solvers use different solving techniques one can expect that better bounds can be established and propagated between solvers. This has been confirmed experimentally and the author reports speed-up of one order of magnitude and memory saving by the same amount.

An interesting approach of combining CP and IP has been proposed in [12, 13]. The authors propose a uniform framework that they call *branch-and-infer*. Their approach is based on the observation that there exist a class of constraint both in IP and CP that have polynomial time solving algorithms. These constraints are called primitive constraints and are used to define restrictions on the solution during search when decisions are made. They are inferred from symbolic constraints, i.e., the symbolic constraints generate (infer) primitive constraints. The primitive constraints of IP are formed using LP relaxation of the problem, i.e., the constraint that decision variables are limited to take integer values is removed. Primitive constraints of CP are domain

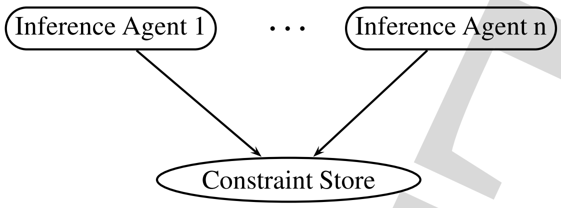


Figure 6.4: The architecture for constraint solving with inference agents.

constraints $a \leq x$, $x \leq b$, $x \neq c$ with integers a, b, c and equation between two FDVs $x = y$.

The computation is carried out by a solver that handles primitive constraints stored in the constraint store. The inference agents are responsible for generation of primitive constraint for non-primitive constraints. The idea is that each non-primitive constraint has an inference agent that is specific for this constraint and can generate primitive constraints that restrict domains of variables. Therefore the primitive constraints, stored in the constraints store, always define a relaxation of the original problem. There exist efficient method for solving this relaxation. The general idea is illustrated in Figure 6.4 [12].

Example 6.6: As an example we can consider the traveling salesperson problem, This problem is difficult to define in IP because of exponential explosion of sub-tour elimination constraints (see section 4.2.7). In branch-and-infer framework this can be solved by defining a special constraint for sub-tour elimination that extends IP solver. This extension defines a special inference algorithm that will add additional primitive constraints to the constraint store for IP solver.

Inferring primitive constraint is a powerful principle but it needs to be combined with an efficient solving method since the inference process may get stuck and no more primitive constraints can be inferred. This is similar to the finite domain constraints that cannot provide more domain narrowing and the search is used to find solutions. In branch-and-infer scenario there exist two basic transition rules that formalize this scheme. The scenarion is defined by the rules of the following form.

$$\texttt{rule\_name:}\ \frac{\langle P, S\rangle}{\langle P', S'\rangle}\ \texttt{if}\ Cond \tag{6.5}$$

where $\langle P, S\rangle$ defines a computation state and if $Cond$ is fulfilled the computation proceeds from state $\langle P, S\rangle$ to a new state $\langle P', S'\rangle$. $P = \{C_1, \ldots, C_m\}$ and it corresponds to the disjunction of C_i. The set S denotes a set of feasible solutions. At the beginning we start with the computation state $\langle\{C\}, \emptyset\rangle$ and the final state is $\langle\emptyset, \{S\}\rangle$.

Two rules define the core of the branch-and-infer framework. One for inferring new primitive constraints and the other one for branching.

$$\texttt{bi\_infer:}\quad \frac{\langle\{(c \wedge C)\} \cup P, S\rangle}{\langle\{(p \wedge (c \wedge C))\} \cup P, S\rangle}\quad \texttt{if} \begin{cases} p : \text{primitive}, \\ c : \text{non-primitive} \\ \text{Prim}(C) \not\to p, \\ \text{Prim}(C) \wedge c \to p \end{cases} \tag{6.6}$$

$$\texttt{bi\_branch:}\quad \frac{\langle C \cup P, S\rangle}{\langle\{c_1 \wedge C, \ldots, c_k \wedge C\} \cup P, S\rangle}\quad \texttt{if} \begin{cases} C \equiv C \wedge (\bigvee_{i=1}^{k} c_i) \\ c_i \text{ primitive} \\ \text{Prim}(C) \not\to c_i \end{cases} \tag{6.7}$$

where $\text{Prim}(C)$ denotes the set of primitive constraints and $\to$ is logical implication.

Rules 6.6 and 6.7 are the same for IP and CP. It is interesting to note that in CP context rule 6.6 is basically defined by the constraint propagators since they produce new restrictions on FDVs domains. For IP it defines additional cuts using cutting planes method, for example. Rule 6.7 is simply applied during search when we create a choice point for several decisions. For example, the DFS creates a choice point for $x_i = 0, x_i = 1, \ldots, x_i = n$, i.e., creates new primitive constraints.

Two rules defined above are, in principle, sufficient to solve any IP and CP problems. The branch-and-infer framework defines, however, additional rules that are used for unsatisfiability check, lower and upper bounds finding and for optimization. The authors did experiments with this approach and applied it to warehouse location problem [12]. For this example they created a new assignment constraint and showed that the branch-and-infer framework is superior in comparison both to IP and CP. It runs much faster than pure CP or IP approaches.

6.4 Local Search

We have discussed in chapter 5 different search methods to find a solution for the defined set of constraints. This search is based on depth-first-search (DFS) principle and basically builds a solution by systematically assigning values to FDVs. In operation research community there are different search methods that do not use DFS search. These methods are used successfully to solve difficult combinatorial problems. There exist, for example, a set of interesting local search methods that are based on different search principles than DFS. The local search starts with an initial solution and then it tries to improve it by applying local transformations. The general algorithm is sketched in Figure 6.5.

The general principles of the local search are following. The search starts from a solution. It can be randomly generated solution or a solution found with other methods. The local search tries to improve this solution by locally changing parts of this solution. This is achieved in the loop that contains step 2 and 3. In step 2 the search chooses a solution from the neighborhood of a current solution and transforms it to obtain a next solution ($x_{next} \in \mathcal{N}(x_{current})$). The next solution is then checked whether it fulfills selection criteria and the stop criteria does not apply. If this is the case the next solution is accepted and assigned to the current solution. Additionally, the best cost and the best solution are updated if the cost of the next solution is better than the best cost achieved so far.

It can be noted that the algorithm selects the next solution based on a specific criteria and it does not mean that the next solution has to be always better than the

```
// Step 1 - Initialization
Select a starting solution $x_{current} \in \mathcal{X}$
$x_{best} \leftarrow x_{current}$
$best\_cost \leftarrow c(x_{best})$

// Step 2 - Choice and termination
do
  Choose a solution $x_{next} \in \mathcal{N}(x_{current})$
  if the choice criteria cannot be satisfied
     by any member of $\mathcal{N}(x_{current})$,
     or the terminating criteria apply
       break

  // Step 3 - Update
  $x_{current} \leftarrow x_{next}$
  if $c(x_{current}) < best\_cost$
     $x_{best} \leftarrow x_{current}$
     $best\_cost \leftarrow c(x_{best})$
while (true)
```

Figure 6.5: The general structure of a local search algorithm.

previous one. The algorithm might, in some special circumstances, accept worse solutions. This is justified since we would like to have opportunity to escape from local optima and therefore have an algorithm that is not greedy and will not stuck in a locally optimal solution.

There exist many different local search algorithms. They are either problem specific heuristics or more general meta-heuristics that can be applied to many problems by defining a set of specific actions. Most popular meta-heuristic algorithms are *simulated annealing* and *tabu search*. These algorithms follow the general principles of local search but apply them differently. Simulated annealing, for example, assumes random acceptance of a solution while tabu search is totally deterministic.

The simulation annealing algorithm is a general framework for building local search optimization algorithms. Similarly to the general local search algorithm it starts from a initial solution. It also assumes a initial temperature and the function for computing temperature reduction. This is needed for this algorithm since it simulates annealing process of a material (typically a metal). In this process, the material is first heated and then slowly cooled down. This makes that the microstructure of a material is altered, causing changes in its properties such as strength and hardness. Typically it removes crystal defects and the internal stresses. By simulating this process we would like to produce a stable solution with no "defects". This final result of the algorithm represents an optimal solution. After initial steps the algorithm executes two loops. The internal loop is executed N times and it basically generates a next solution, x_{next}. This solution is always accepted when its cost is lowered $\delta < 0$, however, it can be sometimes accepted when it cost is not lowered. The acceptance of worse solutions is random and it is based on $e^{\frac{-\delta}{t}}$ factor and a random number p. Since our factor depends on δ and temperature it has lower values when temperature is higher and δ is smaller. therefore it there is higher probability to accept worse solutions at the begin-

```
$x_{current} \in \mathcal{X}$ // an initial solution
$t \leftarrow t_0$       // an initial temperature $t_0 > 0$
           // a temperature reduction function $\alpha$
do
  do
    Select $x_{next} \in \mathcal{N}(x_{current})$
    $\delta \leftarrow f(x_{next}) - f(x_{current})$
    if $\delta < 0$
      $x_{current} \leftarrow x_{next}$
    else
      generate a random number $p$ in the range $(0, 1)$
      if $p < e^{\frac{-\delta}{t}}$
        $x_{current} \leftarrow x_{next}$
  while $iteration\_count \leq N$
  $t \leftarrow \alpha(t)$
while stopping condition $\neq$ true
return $x_{current}$
```

Figure 6.6: The general structure of the simulated annealing algorithm.

ning of the annealing process (higher temperature) than at the end (low temperatures). It is also more likely to accept worse solutions that do not differ very much from the current solution. The outer loop controls temperature by decreasing its value using function α. It also checks for stopping criteria. This criteria usually checks if our solution is "stable" that is typically done by checking three consecutive solutions and if they are the same we assume that it is stable.

The simulated annealing algorithm is a meta-heuristic and requires definition of several parameters and specific algorithms. First, one has to define how to select x_{next} form the neighborhood. This is usually called a "move" and is specific to the considered optimization problem. Consider, for example, a graph partitioning problem as discussed in 4.4. The move operation can be defined as swapping of two nodes between two partitions. This will generate next solutions with different cost function. The algorithm requires also selection of the initial temperature and the cooling scheme (α function).

Example 6.7: Consider a graph coloring problem and a benchmark `fpsol2.i.2` [22]. This is a problem based on register allocation for variables in real codes. The problem graph has 451 nodes and 8691 edges. We have defined a simulated annealing algorithm that finds an optimal solution with 30 colors. The algorithm starts with an initial solution that is obtained by assigning randomly natural numbers (between 1 and 30) to all nodes. It then counts how many nodes connected by an edge have the same color. This value is used as a cost function for minimization. The value zero means that all constraints are fulfilled and the coloring with 30 colors has been found. The move in our case is implemented as a simple change of a value of a single node that does not violate the constraint between two nodes. Obviously it can reduce the cost function or increase when this change will create another conflict with other connected nodes. The values of the cost function, called often penalty, are depicted in Figure 6.7. it is interesting to note that

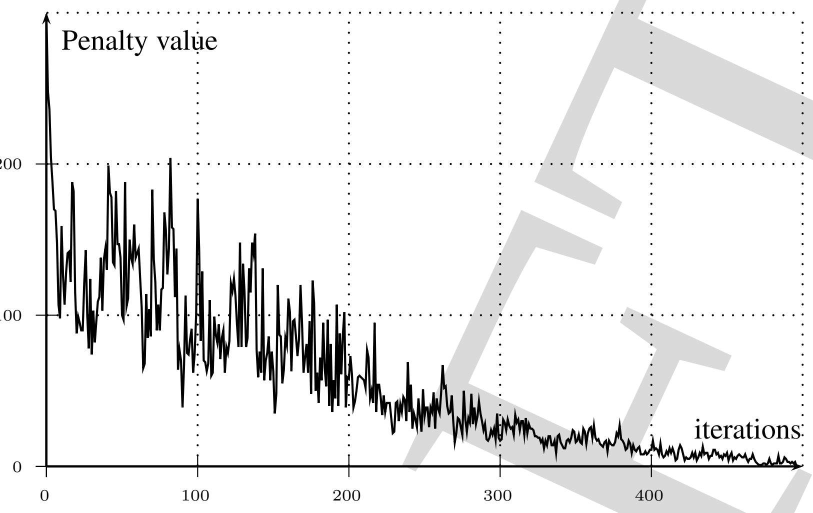


Figure 6.7: The penalty values for the outer loop of the simulated annealing search for graph coloring problem with 451 verticies and 8691 edges during.

at the beginning of the search there are many different solutions that are explored and with the progress of the search the algorithm starts to converge to the stable value that is zero in our case.

Simulated annealing is not the only LS algorithm. There exist other LS algorithms with different principles of selecting a neighborhood, defining moves and strategies to escape local optima. They are often not random. A well known algorithm in this group is the *tabu search* (TS) algorithm [37]. TS is based on the assumption that intelligent search should be based on more systematic forms of guidance rather than random selection as it is the case of simulated annealing. It exploits the history of the search to control the search process. The history is stored in a special structure that encapsulate short-term and long-term memory. The chief mechanism for exploiting this memory is to classify a subset of the neighborhood moves as forbidden (called *tabu*). To achieve this TS maintains a selective history of the moves executed during the search (it might also store the list of states). To avoid repeating or reversing the moves done recently, the most recent moves are usually classified as tabus. Typically a tabu will only be valid for a number of iterations (called *tabu tenure*). When a tabu move would result in a solution better than any visited so far, its tabu classification may be overridden. A condition that allows such override to occur is called an *aspiration criterion*.

TS has succeeded in obtaining optimal and near optimal solutions to a wide variety of classical and system synthesis problems. It outperforms, in many cases, classical heuristics and other neighborhood-search methods, such as simulated annealing [28].

Both LS and CP offer systematic ways to explore solution space. They use different methods and it might be profitable to combine them to obtain more powerful solving strategy. There does not exist a single solution but different authors propose their own methods that are based on their experience and problem domains. The authors of [30], for example, propose several methods to combine local search heuristics with constraint programming methods and they try to make a simple classification by grouping the methods into two basic families.

- The first group of methods is a local search method that uses CP to explore the neighborhood. This method basically uses CP search to choose the next solution x_{next}. The local search explores the space of solutions while CP is used locally to explore the neighborhood. The neighborhood is usually not very big and therefore CP can be used for this purpose efficiently.

- The second group uses CP as the main search method but in each node of the search tree uses LS for exploration of possible solutions. This makes the whole search not complete but it creates opportunity to explore promising solutions more efficiently.

The detail discussion of these methods is beyond the scope of this book but we can illustrate this approach using the first method and its application to the problem of mapping of tasks into computational resources for the Nostrum Mesh Architecture discussed in section 4.5. Assume that we use a local search algorithm as the main search method to solve our problem. This algorithm starts with the initial valid solution that is, in our case, an assignment of tasks to computational resources. Then the algorithm tries to improve the current assignment by making local moves. These moves can be explored using CP formulation. In such a case, the local search algorithm selects a number of tasks and deselects their assignment to computational resources. Then the CP model is build that finds the best possible assignment of the tasks. This assignment is then considered by local search algorithm and it is either accepted or not. Note, that in the local search formulation of our problem one does not need to enforce always valid assignments. Local search algorithms can cope with invalid assignments and penalizes them in the cost function. The optimization process minimizes the penalty and therefore gets most constraints fulfilled.

CP community tries also to develop specialized systems for solving constraint problems using purely local search methods. Michel and Van Hententryck, for example, developed a number of local search frameworks that include both languages and search engines [60, 61, 62]. The systems that have been developed have some common properties. They try to define the problem declaratively by specifying facts. This facts are divided into *invariants*, i.e., facts that does not change and need to be satisfied in all solutions and objects that represent that quality of moves that are called *differentiable objects* in [62]. Having these two classes of facts it is possible to define local search method together with moves and by evaluating the quality of moves make decisions on acceptance of a given solution.

One

JaCoP examples

A.1 Example 1

```
import org.jacop.core.*;
import org.jacop.constraints.*;
import org.jacop.search.*;

public class Main {
    static Main m = new Main ();

    public static void main (String[] args) {
        Store store = new Store();  // define FD store
        int size = 4;
        // define finite domain variables
        IntVar[] v = new IntVar[size];
        for (int i=0; i<size; i++)
            v[i] = new IntVar(store, "v"+i, 1, size);
        // define constraints
        store.impose( new XneqY(v[0], v[1]) );  // $v_0 \neq v_1$
        store.impose( new XneqY(v[0], v[2]) );  // $v_0 \neq v_2$
        store.impose( new XneqY(v[1], v[2]) );  // $v_1 \neq v_2$
        store.impose( new XneqY(v[1], v[3]) );  // $v_1 \neq v_3$
        store.impose( new XneqY(v[2], v[3]) );  // $v_2 \neq v_3$

        // search for a solution and print results
        Search<IntVar> search = new DepthFirstSearch<IntVar>();
        SelectChoicePoint<IntVar> select = new InputOrderSelect<IntVar>(store, v,
                                                                new IndomainMin<IntVar>());
        boolean result = search.labeling(store, select);

        if ( result )
            System.out.println("Solution: " + v[0]+", "+v[1] +", "+v[2] +", "+v[3]);
        else System.out.println("*** No");
    }
}
```

Bibliography

[1] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Math. Comput. Modelling, Pergamon Press Ltd.*, 17(7):57–73, 1993.

[2] P. Baptiste and C. Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI*, pages 600–606, Montréal, Canada, August 20–25, 1995.

[3] P. Baptiste, C. Le Pape, and W. Nuijten. Constraint-based optimization and approximation for job-shop scheduling. In *Proc. AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, 14th International Joint Conference on Artificial Intelligence*, pages 5–16, Montréal, Canada, August 19, 1995.

[4] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publisher, 2001.

[5] R. Barták. On-line guide to constraint programming. First edition. `http://kti.ms.mff.cuni.cz/~bartak/constraints/`, 1998.

[6] N. Beldiceanu, E. Bourreau, H. Simonis, and P. Chan. Partial search strategy in CHIP. Presented at 2nd Metaheuristic International Conference MIC97, Sophia/Antipolis, France, July 21–24, 1997.

[7] N. Beldiceanu and M. Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. *Lecture Notes in Computer Science*, 2239, 2001.

[8] A. Bender. Design of an optimal loosely coupled heterogeneous multiprocessor system. In *Proc. European Design and Test conference*, pages 275–281, Paris, France, 1996.

[9] C. Berge. *Graphs and hypergraphs*. North-Holland, 1973.

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

[11] C. Bessière and J-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings IJCAI'97*, pages 398–404, Nagoya, Japan, 1997.

[12] A. Bockmayr and T. Kasper. Branch-and-infer: an unifying framework for integer linear programming and finite domain constraint programming. *INFORMS Journal of Computing*, 10(3):287–300, September 1998.

[13] A. Bockmayr and T. Kasper. Branch-and-Infer: A framework for combining CP and IP. In M. Milano, editor, Constraint and Integer Programming. Toward a Unified Methodology, 2004.

[14] A. Bockmayr, N. Pisaruk, and A. Aggoun. Network flow problems in constraint programming. In *Principles and Practice of Constraint Programming, CP'2001*, pages 196–210, Paphos, Cyprus, November 2001. LNCS 2239, Springer-Verlag.

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

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

[17] M. Carlsson and C. Schulte. Finite domain constraint programming systems. Tutorial at CP'2002, Int. Conf. on Principles and Practice of Constraint Programming, 2002.

[18] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Proc. 11th International Conference on Logic Programming*, Santa Margherita Ligure, Italy, 1994. MIT Press.

[19] Y. Caseau and F. Laburthe. Cumulative scheduling with task intervals. In *Proc. Int. Conference and Symposium on Logic Programming*. MIT Press, 1996.

[20] Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In *Proc. of the 14th International Conference on Logic Programming*, pages 316–330, 1997.

[21] J. Chang and M. Pedram. Codex-DP: codesign of communicating systems using dynamic programming. *IEEE Trans. Computer-Aided Design*, 19(7):732–744, 2000.

[22] COLOR02/03/04: Graph coloring and its generalizations. Available on: `http://mat.gsia.cmu.edu/COLOR02/`.

[23] COSYTEC. *CHIP System Documentation*, 1996.

[24] E. Danna and C. Le Pape. Two generic schemes for efficient and robust cooperative algorithms. In M. Milano, editor, Constraint and Integer Programming. Toward a Unified Methodology, 2004.

[25] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[26] D. Diaz. *GNU Prolog A Native Prolog Compiler with Constraint Solving over Finite Domains, Edition 1.7, for GNU Prolog version 1.2.16*, September 2002.

[27] C. Ekelin. *An Optimization Framework for Scheduling of Embedded Real-Time Systems*. PhD thesis, Chalmers University of Technology, May 2004. Available from `http://www.ce.chalmers.se/~cekelin/thesis.pdf`.

[28] P. Eles, K. Kuchcinski, and Z. Peng. *System Synthesis with VHDL*. Kluwer Academic Publisher, 1997.

[29] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Kluwer Journal on Design Automation for Embedded Systems*, 2(1):5–32, 1997.

[30] F. Focacci, F. Laburthe, and A. Lodi. Local search and constraint programming. In M. Milano, editor, Constraint and Integer Programming. Toward a Unified Methodology, 2004.

[31] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[32] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1999.

[33] I. Gent and J. F. Puget. Symmetry breaking in constraint programming. Tutorial on Tenth International Conference on Principles and Practice of Constraint Programming, Toronto, Canada, September 27 – October 1, 2004.

[34] I. P. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic SBDD with GAP and ECLiPSe. Technical Report APES-57-2003, APES Research Group, January 2003. Available from `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`.

[35] S. H. Gerez. *Algorithms for VLSI Design Automation*. John Wiley and Sons, December 1998.

[36] M. L. Ginsberg and W. D. Harvey. Iterative broadening. In *National Conference on Artificial Intelligence*, pages 216–220, 1990.

[37] F. Glover, E. Taillard, and D. de Werra. A user's guide to tabu search. *Annals of Operations Research*, 41:3–28, 1993.

[38] F. Gruian and K. Kuchcinski. Operation binding and scheduling for low power using constraint logic programming. In *Proc. 24th Euromicro Conference, Workshop on Digital System Design*, Västerås, Sweden, August 25-27, 1998.

[39] F. Gruian and K. Kuchcinski. Low-energy architecture selection and task scheduling for system-level design. In *Proc. 25th Euromicro conference*, Milan, Italy, September 8–10, 1999.

[40] P. Hall. On representatives of subsets. *Journal of London Mathematical Society*, 10:26–30, 1935.

[41] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *IJCAI-95*, Montreal, Canada, 1995.

[42] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[43] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc., 1993.

[44] I. Katriel and S. Thiel. Complete bound consistency for the global cardinality constraint. *Constraints*, 10(3):191 – 217, July 2005.

[45] S. Kirkpatrick, C. D. Gelatt, and M.P. Vecchi. Optimization by simulating annealing. *Science*, 220(4598):671–680, 1983.

[46] R. E. Korf. Improved limited discrepancy search. In *AAAI/IAAI, Vol. 1*, pages 286–291, 1996.

[47] D. Ku and G. De Micheli. Relative scheduling under timing constraints. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 59–64. ACM Press, 1990.

[48] K. Kuchcinski. Embedded system synthesis by timing constraints solving. In *Proc. 10th International Symposium on System Synthesis*, Antwerp, Belgium, September 17–19, 1997.

[49] K. Kuchcinski. An approach to high-level synthesis using constraint logic programming. In *Proc. 24th Euromicro Conference, Workshop on Digital System Design*, pages 74–82, Västerås, Sweden, August 25–27, 1998.

[50] K. Kuchcinski. Constraints driven design space exploration for distributed embedded systems. *Journal of Systems Architecture*, 47(3-4):241–261, 2001.

[51] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(3):355–383, July 2003.

[52] K. Kuchcinski and Ch. Wolinski. Global approach to assignment and scheduling of complex behaviors based on HCDG and constraint programming. *Journal of Systems Architecture*, 49(12–15):489–503, 2003.

[53] J. Larrosa and G. Valiente. Graph pattern matching using constraint satisfaction. In *Proc. Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 189–196, Berlin, Germany, March 25–27 2000.

[54] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. A constraint-based application model and scheduling techniques for power-aware systems. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 153–158. ACM Press, 2001.

[55] A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 245–250, Acapulco, Mexico, August 2003.

[56] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[57] P. Martin and D. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In W.H. Cunningham, S.T. McCormick, and M. Queyranne, editors, *Proc. 5th International IPCO Conference*, pages 389–403, Vancouver, British Columbia, Canada, 1996.

[58] K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 306–319. Springer-Verlag, 2000.

[59] Z. Michalewicz. *Genetic Algorithms + Data Structure = Evolution Programs*. Springer-Verlag, 1994.

[60] L. Michel and P. van Hentenryck. LOCALIZER a modeling language for local search. In *Proc. Third International Confernece on Principles & Practice of Constraint Programming*, Schloss Hagenberg, Austria, oct 1997.

[61] L. Michel and P. Van Hentenryck. Localizer++: An open library for local search. Technical Report CS-01-02, Brown University, January 2001.

[62] L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, November 2002.

[63] M. Milano, editor. *Constraint and Integer Programming. Toward a Unified Methodology*. Kluwer Academic Publisher, 2004.

[64] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The Nostrum backbone-a communication protocol stack for Networks on Chip. In *Proceedings of the 17th International Conference on VLSI Design*, pages 693–696, 2004.

[65] T. Müller. The Mozart constraint extensions tutorial. Available on: `http://www.mozart-oz.org/documentation/cpitut/index.html`, June 2004.

[66] W. J. Older and F. Benhamou. Programming in CLP(BNR). In *Principles and Practice of Constraint Programming*, pages 228–238, 1993.

[67] P. Paulin and J. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Trans. Computer-Aided Design*, 8:661–679, June 1989.

[68] T. Petit, J-C. Régin, and Ch. Bessière. Specific filtering algorithms for over-constrained problems. *Lecture Notes in Computer Science*, 2239:451–464, 2001.

[69] S. Prakash and A. Parker. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16:338–350, 1992.

[70] F. P. Preparata and M. I. Shamos. *Computational Geometry. An Introduction.* Springer-Verlag, 1985.

[71] PrologIA. *Prolog IV Manual*, 1998.

[72] J. F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the Fifteenth National Conference on Artiffcial Intelligence (AAAI'98)*, pages 359–366. American Association for Artificial Intelligence, 1998.

[73] J. F. Puget. Symmetry breaking revisited. In *Principles and Practice of Constraint Programming – CP 2002*, volume 2833 of *Lecture Notes in Computer Science*, pages 446–461, Ithaca, NY, USA, September 2002. Springer-Verlag.

[74] C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Principles and Practice of Constraint Programming – CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 600–614, Kinsale, Ireland, November 2003. Springer-Verlag.

[75] J-C. Régin. A filtering algorithm for constraint of difference in CSPs. In *12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, Menlo Park, California, USA, 1994.

[76] J-C. Régin. Generalized arc consistency for global cardinality constraint. In *13th National Conference on Artificial Intelligence (AAAI-96)*, pages 209–215, Portland, OR, USA, 1996.

[77] J-C. Régin. Solving the maximum clique problem with constraint programming. In *Proc. CPAIOR*, 2003.

[78] J-C. Régin. Global constraints and filtering algorithm. In M. Milano, editor, Constraint and Integer Programming. Toward a Unified Methodology, 2004.

[79] V. Scholz. Knowledge-based locomotive planning for the Swedish railway. Technical Report T2000-05, Swedish Institute of Computer Science, 1998.

[80] Ch. Schulte and G. Smolka. Finite domain constraint programming in Oz. A tutorial. Available on: `http://www.mozart-oz.org/documentation/fdt/index.htm`, 2001. version 1.2.1.

[81] R. Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, 1988.

[82] H. Simonis. Sudoku as constraint programming. In P. Prosser B. Hnich and B. Smith, editors, *Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 13–27, Sitges (Barcelona), Spain, October 1, 2005.

[83] Swedish Institute of Computer Science. *SICStus Prolog User's Manual, Version 3.11*, June 2004.

[84] R. Szymanek, F. Catthoor, and K. Kuchcinski. Data assignment and access scheduling exploration for multi-layer memory architectures. In *Proc. of ISSS+CODES, satellite workshop – ESTImedia*, Stockholm, Sweden, September 2004.

[85] R. Szymanek, F. Catthoor, and K. Kuchcinski. Time-energy design space exploration for multi-layer memory architectures. In *Proc. of 7th ACM/IEEE Design, Automation and Test in Europe Conference*, Paris, France, February 16–20, 2004.

[86] R. Szymanek, F. Gruian, and K. Kuchcinski. Digital systems design using constraint logic programming. In *Proc. The Second International Conference on The Practical Applications of Constraint Logic Programming, (PACLP 2000)*, Manchester, UK, April 10–12, 2000.

[87] R. Szymanek and K. Kuchcinski. Design space exploration in system level synthesis under memory constraints. In *Proc. 25th Euromicro conference*, Milan, Italy, September 8–10, 1999.

[88] R. Szymanek and K. Kuchcinski. A constructive algorithm for memory-aware task assignment and scheduling. In *Proc. 9th International Symposium on Hardware/Software Codesign*, Copenhagen, Denmark, April 2001.

[89] R. Szymanek and K. Kuchcinski. Partial task assignment of task graphs under heterogeneous resource constraints. In *Proc. of the 40th Design Automation Conference*, Anaheim, USA, June 2–6, 2003.

[90] E. Tsang. *Foundations of Constraint Programming*. Academic Press Ltd., 1993.

[91] F. Vahid and T. Givargis. *Embedded System Design – A Unified Hardware/Software Introduction*. John Wiley and Sons, 2002.

[92] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *7th International Conference on Principles and Practice of Constraint Programming - CP 2001*, pages 625–639, Paphos, Cyprus, November 26 - December 1 2001. Springer-Verlag, Lecture Notes in Computer Science, Volume 2239.

[93] P. Van Hentenryck, P. Flener, J. Pearson, and M. Ågren. Tractable symmetry breaking for CSPs with interchangeable values. In *International Joint Conference on Artificial Intelligence (IJCAI'03)*, Acapulco, Mexico, August 2003.

[94] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica, A Modeling Language for Global Optimization*. The MIT Press, April 1997.

[95] W. J. van Hoeve. The `alldifferent` constraint: A survey. Submitted for publication. Availbale on: `http://www.cwi.nl/~wjvh/papers/alldiff.pdf`, 2003.

[96] M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On benchmarking constraint logic programming platforms. Response to Fernandez and Hill's "A comparative study of eight constraint programming languages over boolean and finite domains". *Constraints: An International Journal*, 9:5–34, 2004.

[97] T. Walsh. Depth-bounded discrepancy search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI*, pages 1388–1395, Nagoya, Japan, 1997. Morgan Kaufmann.

[98] W. Wolf. *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufman Publishers, 2000.

Index