2. Constraint Satisfaction Problems

2.1 Introduction

Many AI synthesis problems such as planning, configuration, scheduling, etc., may be modeled as *Constraint Satisfaction Problems (CSPs)*. A CSP is basically a search problem, the solution of which requires both powerful search heuristics and adequate means of representation. On the one hand, CS - techniques can be seen as simple search strategies in a given (usually very large) search space, on the other hand, we can interpret CS as a very strong problem solving paradigm with whom the objects, their variables with the respective value domains and the relations among these objects can be described. A CSP is typically defined as the problem of finding any labeling for a fixed set of variables satisfying all given constraints between these variables. The task of a general *inference mechanism* is to find a solution of this problem, what is done by finding a consistent instantiation of all objects involved. CS-methods provide the advantage of a strict declarative representation of a given problem. Furthermore the strategies applied to find a solution are independent of the problem area. So we have a strict distinction between the declaration of a problem and the inference mechanism. In the following sections we will introduce some definitions on (classical)CSPs.

“Classical“ constraint satisfaction problems are usually composed of *crisp* constraints, sometimes called *boolean* or *hard* constraints, and so relations can either be satisfied or not, without intermediate state. A solution must satisfy all constraints of the problem. If a problem has more than one solution, it is called an *underconstrained problem*. To get one solution, one or more additional constraints have to be considered. If a problem has no solution at all, it is called *overconstrained*. Some constraints must then be relaxed in order to find acceptable solutions. In case of underconstrained problems we can specify preferences in the valid domain of the variables in order to grade solutions that otherwise would be all equal. In case of overconstrained problems we can specify margins for constraints where they can be relaxed while not leading to unacceptable results.

2.2 Definition of classical (crisp) CSPs

Constraint satisfaction techniques are well equipped for defining and solving certain problems in AI and technical applications such as planning, configuration problems, scheduling etc. One important property of CSPs lies in the declarative problem-representation and the strict distinction between the definition of a problem and the inference mechanisms for finding a feasible solution.
Although there exists a variety of definitions what CSPs are and look like, the fundamental concepts are more or less the same. We introduce some basic definitions on constraints and constraint networks, taken from [Haselböck91].

- **Definition 1 (Constraint)**
  An *n-ary* constraint C consists of Variables $V_1,...,V_n$ over domains $D_1,...,D_n$ and a decidable relation $R \subseteq D_1 \times ... \times D_n$. The domains of the variables are finite sets of values.

- **Definition 2 (Constraint Network)**
  A *constraint network* on variables $V_1,...,V_m$ consists of constraints $C_1,...,C_n$, the variables of each being a subset of $V_1,...,V_m$.

- **Definition 3 (Satisfaction)**
  A tuple $(d_1,...,d_m)$ *satisfies* a m-ary constraint C_i if the sub-sequence of $(d_1,...,d_m)$ that corresponds to the variables of C_i is an element of the relation C_i.

- **Definition 4 (Solution)**
  A *solution* of a constraint network that is given by the constraints $C_1,...,C_n$ on variables $V_1,...,V_m$ over domains $D_1,...,D_m$ is a tuple $(d_1,...,d_m) \in D_1 \times ... \times D_n$ that satisfies the constraints. A solution is a completeinstantiation of all variables $V_1,...,V_m$.

The task of finding one, some, or every solution of a constraint network is called constraint satisfaction problem. In general, CSPs are NP-complete and thus hard to solve. Therefore, algorithms have been developed that do not compute a solution but just restrict the space of searching for such a solution by limiting the domains of these variables such that they fulfill certain more local requirements. These requirements can be expressed in terms of consistency, k-consistency, and strong k-consistency, that are defined as follows:

❑ **Definition 5 (Consistency)**

A *constraint network* on variables $V_1,...,V_m$ over domains $D_1,...,D_m$ is *consistent* if for each $j \in \{1,...,m\}$:
every tuple $(d_1,...,d_{j-1},d_{j+1},...,d_m)$, $d_i \in D_i$ for all $i \in \{1,...,m\} \setminus \{j\}$, that satisfies the constraints of the network can be extended to an m-ary tuple $(d_1,...,d_m)$, $d_j \in D_j$, that satisfies the constraints.

❑ **Definition 6 (k-Consistency)**

A *constraint network is k-consistent* if each k-variable subnetwork is consistent.

❑ **Definition 7 (Strong k-Consistency)**

A *constraint network is strongly k-consistent* if it is i-consistent for all $i \leq k$.

EXAMPLE:
A classical constraint satisfaction problem is the n - queens - problem. For n = 4 on a 4×4 - chessboard 4 queens have to be positioned so that any two queens cannot beat each other. So we can define the following CSP:

$V = \{v_1,v_2,v_3,v_4\}$ representing the 4 rows on the board.
$D_i = \{1,2,3,4\}$ where $1 \leq i \leq 4$
$C = (v_i \neq v_j) \wedge (|v_i - v_j| \neq j - i)$ where $1 \leq i < j \leq 4$
$T = \{(2,4,1,3),(3,1,4,2)\}$ are the tuples $(d_1,...,d_m) \in D_1 \times ... \times D_n$ that satisfy the constraints.

2.3 Constraints

Constraints are a general method to formulate dependencies and relations among different objects (variables). Formulating constraints gives us more independence in solving a problem in comparison to functional methods. A functional term like

$$z = x + y$$

can only be evaluated if the values of x and y are known. A constraint of the form

$$C_{xyz} : z = x + y$$

leads to the inference of different relations:

- *Consistency Check*: All of the three variables are instanciated. We can check, whether C_{xyz} is satisfied or not.

- *Computation*: Two of the three variables are instanciated. We can compute the value of the third variable.

- *Filtering*: One or none of the three variables is instanciated. We can filter the domains of the remaining variables and reduce the search space.

Constraints can be formulated *intentional* or *extentional*. Intentional formulated constraints are formulated as compact relations, as e.g., $x < y$, which have to be improved by procedural methods. Extentional constraints are finite sets of possible tuples of values, as e.g., {(2,3),(5,10),...}.

Depending on the type of a constraint we can differentiate between

- *Boolean Constraints*: the inference apparatus checks all variables of a solution on the validity of their values and returns *true* or *false* on each check. Boolean constraints are used especially in search algorithms such as backtracking etc.

- *Numerical Constraints*: problems, which are described by numerical relations among the variables need techniques that find instantiations of the variables in order to satisfy a maximum of constraints. Such systems are called *numerical optimization problems*.

2.4 Partial Constraint Satisfaction

The above method may work for solvable constraint satisfaction problems. Partial constraint satisfaction relaxes the concept such that only a subset of variables is instanciated, satisfying a subset of the constraints. Some constraints may be violated in order to permit more valid instantiations.

A constraint satisfaction problem is *solvable* if there exists a tuple $(d_1,...,d_m) \in D_1 \times ... \times D_n$ that satisfies every constraint. An unsolvable constraint satisfaction problem is called *overconstrained*. A *partial solution* of a constraint network $C_1,...,C_n$, on variables $V_1,...,V_m$ over domains $D_1,...,D_n$ is a tuple $(d_1,...,d_m) \in D_1 \times ... \times D_n$ that violates some constraints. Hence partial constraint satisfaction is a *relaxation* of constraint satisfaction. The one partial solution with the smallest number of violated constraints is usually declared solution of the partial constraint satisfaction problem.

2.5 Solving Constraint Satisfaction Problems

The search space of a Constraint Satisfaction Problem is usually very large, and its size can depend exponentially on the number of involved variables and the size of their respective domains. In most cases the valid solutions are scattered sparsely over the space of all possible variable-value assignments. We distinguish two basic methods for solving a Constraint Satisfaction Problem. One class is the *constructive approach*, where a solution is generated systematically by extending a partial valid variable-value assignment until a complete solution is reached. The other way is to generate a first, more or less random based instantiation of all variables and to *repair* all inconsistent value assignments, by selecting other assignments from the domains of the respective variables. To determine the satisfaction degree of a solution, both constructive and repair algorithms must provide an *evaluation function*, which is capable of aggregating an overall satisfaction degree for a set of constraints. In classical CSPs constraints are assumed to be binary and finite. In this case the evaluation function is equivalent to a consistency test for a solution. Considering the repair based approach and partial constraint satisfaction, we want the evaluation function to establish a ranking or ordering of instantiations or solutions, in order to grade solutions or to decide between possible repair steps.

2.5.1 Constructive Approaches

A constructive algorithm provides a deterministic behavior and guarantees that an optimal solution, if there exists one, will be found. Constructive algorithms generate subsequently variable-value assignments until a solution is found. Constructive algorithms work only with a small number of variables and simple problem situations. The progress of the value-assignment is usually displayed by the algorithm’s search tree.

2.5.1.1 Backtracking

A backtracking algorithm tries to expand a partial solution of a problem to a complete solution by assigning subsequently values to all variables of the constraint satisfaction problem. In its standard version, the backtracking algorithm traverses the variables in a predetermined order, provisionally assigning consistent values to a subsequence $(V_1,...,V_i)$ of variables and attempting to append to it a new instantiation of V_{i+1} such that the whole set is consistent.

It starts with selecting a value from the first variables domain. Then a value for the next variable's domain is selected. Each new assignment is checked against the previous assignments that are related to the actual assignment by constraints. If the n-th assignment is inconsistent and violates a constraint, the algorithm tries to find a new value from the domain of the n-th variable. If no value of this variable's domain satisfies the related constraints, a local failure occurred and the previous assignments have to be observed. This case is called *dead-end* situation. The backtracking algorithm then backs up to the (n-1)-th variable and assumes its value being incompatible with the n-th variable. A new value for the (n-1)-th variable is selected and the algorithm tries again to instantiate the n-th variable with a value of the appropriate domain. When changes on the (n-1)-th level do not lead to a correct assignment on the n-th level, then the (n-2)-th assignment has to be considered incompatible with its successors etc.

The algorithm succeeds when all variables are assigned values with no constraints violated by the assignments. If all combinations were checked and no valid instantiation of the variables was found, then the CSP has no solution at all.

The substantial disadvantage of simple backtracking as described above, is the so-called *trashing*, that can occur when a violation is encountered and the algorithm may try a lot of assignments on the actual level, while the reason for the violation lies elsewhere in the search tree.

A simple backtracking algorithm taken from [Dechter90] is given below:

```
Forward(x1,...,xi)
begin
     1. if i = n exit with the current assignment.
     2. Ci+1 ← Compute_candidates(x1,...,xi,Xi+1)
     3. if Ci+1 is not empty then
          xi+1 ← first element in Ci+1 and
          remove xi+1 from Ci+1 and
          Forward(x1,....,xi+1)
        else
          Go-back(x1,...,xi)
end.


Go-back(x1,...,xi)
begin
     1. if i = 0 exit. No solution exists.
     2. if Ci is not empty then
          xi ← first element in Ci and
          remove xi from Ci and
          Forward(x1,...,xi)
        else
          Go-back(x1,...,xi-1)
end.
```

EXAMPLE: Referring to the n-queens problem we illustrate the backtracking mechanism for n = 4:

$V = \{v_1,v_2,v_3,v_4\}$ representing the 4 rows on the board.
$D_i = \{1,2,3,4\}$ where $1 \le i \le 4$
$C = (v_i \neq v_j) \wedge (|v_i - v_j| \neq j - i)$ where $1 \le i < j \le 4$

$T = \{(2,4,1,3),(3,1,4,2)\}$ are the tuples $(d_1,\ldots,d_m) \in D_1 \times \ldots \times D_n$ that satisfy the constraints.

The backtracking algorithm proceeds as follows:

```
root  → (v1=1) → (v2=1) → fail
               → (v2=2) → fail
               → (v2=3) → (v3 =1)→ fail
                        → (v3=2) → fail
                        → (v3=3) → fail
                        → (v3=4) → fail
               → (v2=4) → (v3=1) → fail
                        → (v3=2) → (v4=1) → fail
...
      → (v1=2) → (v2=4) → (v3=1) → (v4=3) → success
```

The search tree grows as follows:

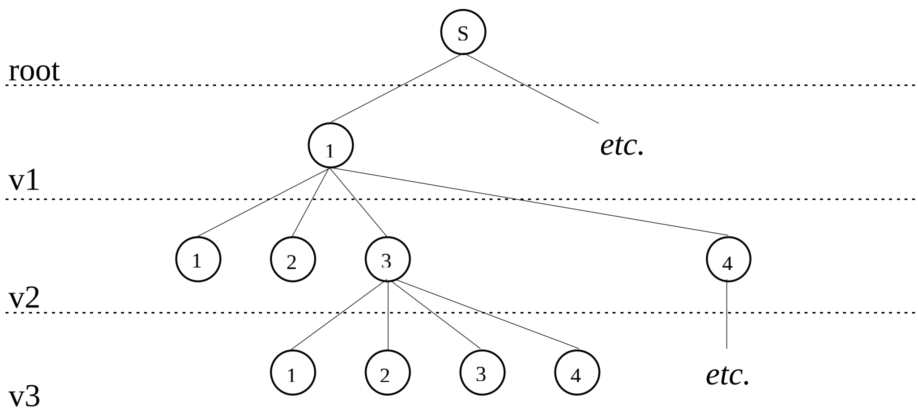

Figure 2.1: search tree built up by the backtracking algorithm

2.5.1.2 Branch and Bound

Branch and bound is a backtracking extension for partial satisfaction problems. This algorithm takes into account the actual best evaluation score and tries to find a better solution. When it comes clear that a partial solution cannot lead to a better solution than the best one found so far, the assignments for the remaining variables can be discarded. This occurs when the variables in the current search path violate the constraints to a higher extent than the current best solution. The lowest degree of violation encountered so far is called *necessary bound*. A new instantiation with a lower degree of violation updates the necessary bound. The algorithm succeeds when a solution reaches a given *sufficient bound*, or all combinations of the variables have been tested. In real time application often a *time bound* is given, after which the algorithm delivers the best solution found so far (anytime feature).

2.5.1.3 Prospective and Retrospective Methods

Due to the bad performance of classical backtracking algorithms some enhancements were proposed. There exist two fundamental approaches to reduce trashing while backtracking. A *prospective algorithm* tests a new value against the domains of variables not yet contained in the current search path in order to detect inconsistencies to future variables, before this variables are considered. The most important prospective methods are basically *Partial Look Ahead* and *Forward Checking*. *Retrospective methods,* such as *Backchecking, Backmarking* and *Backjumping* test a new value against the variable-values contained in the current search path. For detailed information on these techniques please refer to [Haselböck91], and [Dechter90].

2.5.2 Repair based Approach

Repair based heuristics (sometimes also called ‘iterative improvement techniques’) have successfully been applied to solve problems that previously seemed intractable, such as the n-queens problem and the graph-coloring problem. They have generally scored much better than constructive approaches that start from an empty instantiation and build successively more and more complete partial instantiations, such as constructive backtracking methods. Like the branch and bound extension of backtracking, repair based methods inherently possess the anytime property: they return a good solution from the beginning, but can continue the search for a very good solution as long as there is time left.

Unlike constructive approaches, repair based approaches start from a complete variable-value assignment, the latter often being generated randomly selecting values from the respective domains. In most cases, this initial assignment might be suboptimal and erroneous. Repair based methods try to improve the consistency of the assignments by applying local repair heuristics until a consistent instantiation of all variables is achieved.

2.5.2.1 Tabu Search

„Tabu search is a 'higher level' heuristic procedure for solving optimization problems, designed to guide other methods (or the component process of these methods) to escape the trap of local optimality. Tabu search has succeeded in obtaining optimal and near optimal solutions to a wide variety of classical and practical problems, in applications ranging from scheduling to telecommunications, and from character recognition to neural networks. [...]" [Glover89]

Tabu search is a general purpose *metaheuristic* for other procedures to prevent them from being trapped at locally optimal solutions. The method can be used to guide all those processes, where a set of moves is used for transforming one solution into another, and which provide an evaluation function for measuring the attractiveness of these moves.

A repair based scheduler using tabu search proceeds as follows [Glover89], [Glover90]:

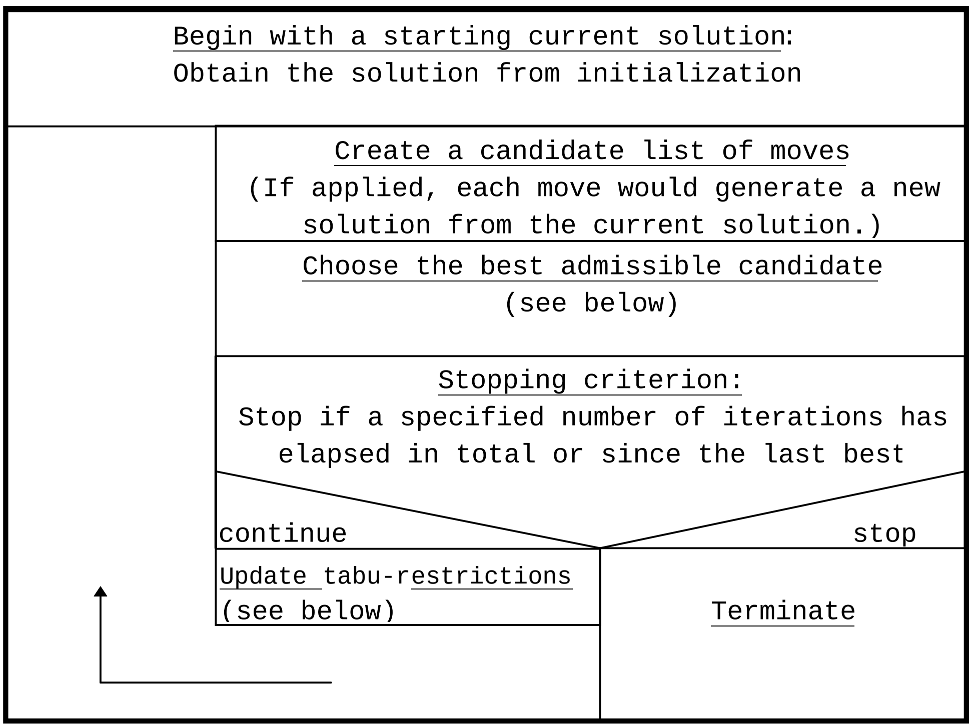

Figure 2.2: Tabu Search algorithm

The best admissible candidate is chosen among a list of possible repair steps by an evaluation function, which can be based on the change of the global evaluation function before and after the repair step. Instead of a possibly expensive global evaluation function, a relaxed heuristic approximation (a local measure of attractiveness, [Glover89]) will also do. The selection of the best admissible candidate in detail:

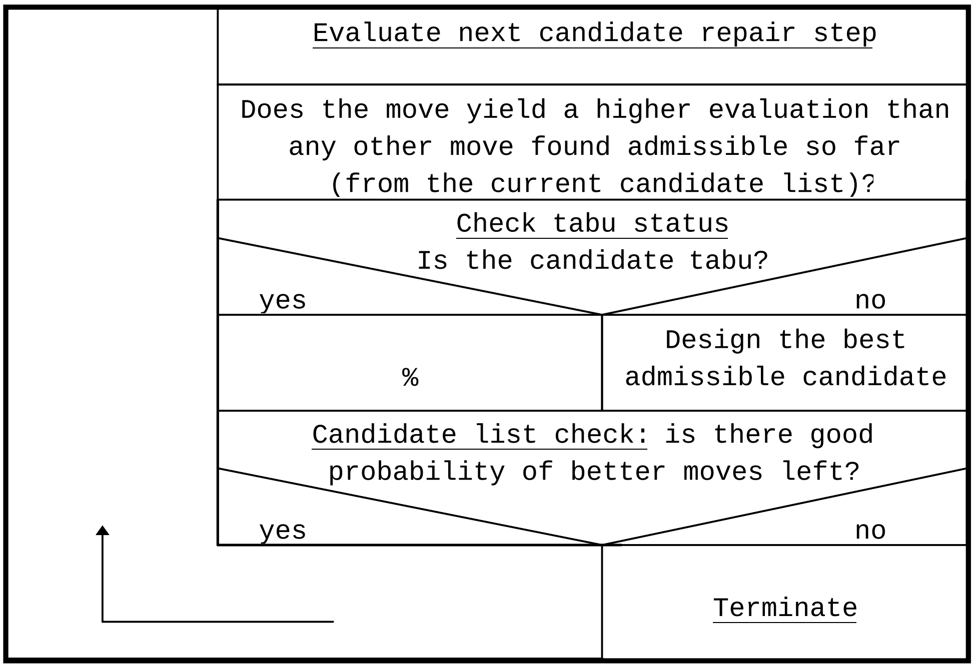

Figure 2.3: selection of the best admissible repair step

The tabu restrictions operate by identifying *attributes* of repair steps. A standard type of repair step is the transfer of operations from one position to another one, which can be described by the pair repair_step(source, destination). Other attributes can be the evaluation scores before and after the transfer. The tabu restrictions are implemented by a set of such attributes and by recording them on a *tabu list*, holding the last number t of iterations. For each repair step that is added to the list, the least recent entry is dropped. A repair step is tabu when its attributes are on the list, and it is released from tabu status, once its attributes are removed. The tabu list prevents the optimization algorithm from being trapped in a local maximum and from pathological cycles up to a maximum length t as it forces the algorithm to try out other repair steps than the t previously performed.

2.5.3 Genetic Algorithms

An other approach to solve Constraint Satisfaction Problems or Scheduling Problems as a special case of CSPs, are the genetic algorithms, deriving originally from cellular biology. This methods are completely different from the others we have seen before, but are very promising attempts to get good solutions with quite simple operations on the schedule.

Genetic algorithms operate on a finite set of possible task orderings, which is modeled after a biological population. Every task ordering in the population has its ordering encoded into a sort of *chromosome*. A *coding scheme* provides the bijective mapping between orderings and the respective chromosomes. The extent to which a possible task ordering satisfies the constraints can be seen as a measure of the orderings *fitness*. Like other repair based approaches an initial solution (population) is generated and transformed by the *genetic operators* into populations with higher fitness values.

For further description of genetic algorithms please refer to [Goldberg89]. To illustrate the principles of genetic operators, we describe three types of genetic operators:

- *reproduction*: Duplicates individual chromosomes according to their fitness value
- *crossover*: Choose pairs of individuals (fitter individuals more often, or at random) and produce new pairs:

$$
\begin{array}{r|l}
X = 101101 & 1101 \\
Y = 011011 & 0011 \\
\multicolumn{1}{r}{i} & \\
\end{array}
\qquad\qquad
\begin{array}{r|l}
X' = 101101 & 0011 \\
Y' = 011011 & 1101 \\
\multicolumn{1}{r}{i} & \\
\end{array}
$$

Figure 2.4: crossover-operator

  The crossover position i is chosen at random

- *mutation*: Is performed for each bit in each chromosome and flips random bit's values according to a *mutation rate*.

2.6 Conclusion

Constraint Satisfaction Problems involves finding values for variables subject to constraints on acceptable combinations of values. Classically, a CSP is defined by a set of variables, each associated with a domain of possible values and a set of constraints on subsets of the variables. A constraint specifies which values from the domains of its variables are compatible. A compatible set of values, one for each variable participating in a constraint, is said to satisfy the constraint. A solution to the CSP is an assignment of values to all its variables which satisfies all the constraint. Depending on the problem, one may be interested in finding one or all the solutions. Some incremental filtering and search techniques that have been developed in Artificial Intelligence provide efficient means to solve CSPs.

Classical CSPs usually are composed of crisp constraints. As we have seen above, a crisp constraint can either be satisfied or not by the value d_i of a variable V_i, and so a crisp CSP-solver may reject a „solution“ because of an „insignificant“ violation of a constraint. In the next section we will introduce fuzzy theory and *soft constraints*, in order to accept such insignificant violations.