Chapter 2
Background

Words, mademoiselle, are only the outer clothing of ideas.
Agatha Christie

Both constraint satisfaction and Boolean satisfiability problems have been around for centuries. However, the terms were only coined in the twentieth century. Boolean satisfiability has its roots in logic. In fact, any propositional logic formula is an instance of the *Boolean satisfiability problem* (SAT). That's why the terms *propositional satisfiability* or simply just *satisfiability* are also commonly used. Constraint satisfaction, on the other hand, belongs to the field of artificial intelligence. It covers a very wide range of problems. Graph colouring, n-queens and time scheduling are just a few examples of problems which can be modelled as a set of *constraints* that need to be satisfied. *Constraint programming* (CP) deals with solving these kinds of problems.

Boolean satisfiability and constraint satisfaction independently emerged as new fields of computer science and significantly different approaches have been used to solve problems in these two areas of knowledge. Interestingly enough, any propositional formula can actually be viewed as an instance of the *constraint satisfaction problem* (CSP), so SAT can be seen as a special case of CSP (for a comprehensive overview on the history of CSP and SAT see [FM06] and [FM09]).

In this chapter we review algorithms used for solving instances of CSP and SAT. Section 2.1 introduces the constraint satisfaction problem. Section 2.2 presents the most common algorithms used in standard constraint solvers; variations on the basic propagation and search strategies are also discussed. Section 2.3 introduces the Boolean satisfiability problem. Section 2.4 deals with the methods used in SAT-solvers. Section 2.5 considers hybrid approaches and Section 2.6 concludes the chapter.

J. Petke, *Bridging Constraint Satisfaction and Boolean Satisfiability*,
Artificial Intelligence: Foundations, Theory, and Algorithms,
DOI 10.1007/978-3-319-21810-6_2

2.1 Constraint satisfaction problem (CSP)

Any problem that needs to satisfy a set of *constraints* (also known as *relations*) between variables having finite domains is in fact an instance of the (finite-domain) *constraint satisfaction problem* (CSP) which is formally defined as follows.

Definition 2.1 (CSP) *An instance of the* constraint satisfaction problem (CSP) *is specified by a triple* (V, D, C)*, where*

- V *is a finite set of* variables*;*
- $D = \{D_v \mid v \in V\}$ *where each set* D_v *is the set of possible values for the variable* v*, called the* domain *of* v*;*
- C *is a finite set of* constraints*. Each constraint in* C *is a pair* (R_i, S_i) *where*
  - S_i *is an ordered list of* m_i *variables, called the constraint* scope*;*
  - R_i *is a relation over* D *of arity* m_i*, called the constraint* relation*.*

A simple example of a CSP instance is the 8-queens problem: one needs to place 8 queens on a 8×8 chessboard in such a way that no two queens attack each other. In this case queens can be regarded as variables, positions on the chessboard as domain values and constraints are the restrictions specifying that no two queens share the same row, column or diagonal (see Figure 2.1).

It is worth mentioning that in theoretical papers D often denotes a single set of domain values. In that context, to express that a variable v can only take values from some subset of D, unary constraints are introduced. In practice, however, CSP-solvers usually work by maintaining a list of available values for each variable and removing the unsatisfiable variable-value pairs. For this purpose, the above definition is more suitable.

Definition 2.2 *A* solution *to a CSP instance* $P = (V, D, C)$ *is an assignment of values from* D *to each of the variables in* V*, which satisfies all of the constraints in* C *simultaneously. Formally, a solution is a map* $h : v \in V \to \bigcup D_v$ *such that* $h(v) \in D_v$*, for all* $v \in V$*, and* $h(S_i) \in R_i$*, for all* i*, where the expression* $h(S_i)$ *denotes the result of applying* h *to the tuple* S_i*, coordinate-wise (in other words, if* $S_i = (v_1, \cdots, v_k)$*, then* $h(S_i) = (h(v_1), \cdots, h(v_k))$*).*

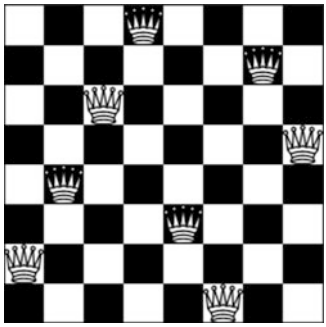

Fig. 2.1 An example solution to the 8-queens problem.

A CSP is called binary if each constraint scope includes exactly two variables. Any binary CSP can be represented as a constraint graph, in which each node corresponds to a variable and each edge represents a constraint between two nodes.

Definition 2.3 *Each node in the* primal graph *of a (binary) CSP instance P represents a variable of P. Two nodes are connected by an edge if and only if they are in the scope of the same constraint.*

Note that any non-binary CSP can be transformed into an equivalent binary CSP [RDP91]. Hence any CSP can be represented as a constraint graph[1]. In an *ordered* constraint graph the vertices are arranged in a linear order.

Definition 2.4 *[Fre82] The width of a vertex in an ordered constraint graph is the number of edges that lead back from that vertex to its predecessors (in the linear order). The width of an ordered constraint graph is the maximum width of any of its vertices. The* width of a constraint graph *is the minimum width of all the orderings of that graph.*

Definition 2.5 *[Dec06, Def. 7.10] The induced width of an ordered constraint graph is the width of the induced ordered graph, obtained by processing the vertices recursively, from last to first; when vertex v is processed, all its earlier neighbours are connected. The* tree-width[2] *of a graph is the minimal induced width over all its orderings.*

Let us now introduce another way by which one can represent a CSP instance graphically.

Definition 2.6 *Each node in the* incidence graph *of a CSP instance P represents either a variable or a constraint of P. Two nodes are connected by an edge if and only if one of them represents a constraint C and the other a variable in the scope of C.*

Moreover, we can translate a non-binary CSP into a binary one using the *hidden variable* method [RDP91].

Definition 2.7 *Let $P = (V, D, C)$ be a CSP instance. For each constraint $C_i \in C$ introduce a new variable v_{C_i} with domain D_{C_i} of tuples t satisfying C_i. Let $V' = V \cup \{v_{C_i} \mid C_i \in C\}$, $D' = D \cup \bigcup_{C_i \in C} D_{C_i}$ and $C' = \{v_{C_i} = t \leftrightarrow \forall_{t_j \in t}\ v_j = t_j$*

[1] It is worth mentioning, however, that sometimes the binary representation of a non-binary CSP needs to introduce new nodes or edges. For instance, consider the not-all-equal constraint on three variables. This is a ternary constraint and cannot be represented by a constraint graph with three nodes representing the original variables and edges representing binary constraints on these variables. Because of those often extensive additional space requirements, and the loss of information about the problem structure, the method of translating a non-binary CSP instance into a binary one is not usually used in practice.

[2] An alternative definition of tree-width was given in [RS86], where the concept was first introduced.

$| t \in D_{C_i} \wedge v_j \in V\}$, *where* v_j *represents the* j^{th} *variable in the scope of* C_i. *Then* $P' = (V', D', C')$ *is the* hidden variable representation *of* P.

Note that the primal graph of the hidden variable representation of P is the incidence graph of P.

The structure of a non-binary CSP instance can be also represented by a hypergraph.

Definition 2.8 *A hypergraph is a pair* $H = (V, E)$, *where* V *is an arbitrary set, called the vertices of* H, *and* E *is a set of subsets of* V, *called the hyperedges of* H.

For any CSP instance, the scopes of all the constraints can be viewed as the hyperedges of an associated hypergraph whose vertices are the variables.

2.2 CSP-solvers

An important implementation decision that creators of constraint solvers must make is the choice of an input format. There exists no standard modelling language. In the CSP-solver competitions benchmark instances are represented in the XML format [vDLR06, vDLR08]. However, higher-level modelling languages such as Zinc [NSB+07] and Essence [FGJ+07] have been developed in the quest to find a more expressive and succinct model for CSP input.

Example 2.9 *The 8-queens problem can be specified in Essence as follows:*

```
language ESSENCE 1.2.0

$ Index: column and row indices
letting Index be domain int(*1..8*)

$ arrangement: one queen is placed on each row,
$              at the column index specified by
$              this function; the bijection ensures
$              each column contains exactly one queen
find arrangement : function Index -> (*bijective*) Index

$ no queens share diagonals; neq stands for 'not equal'
such that forall q1, q2 : Index . q1 neq q2 implies
            |arrangement(q1) - arrangement(q2)| neq |q1 - q2|
```

The objective of a constraint solver is to find an assignment of variables such that the set of constraints imposing conditions on those variables is satisfied. One can identify two components in the algorithm underlying any constraint solver — *inference* (also called problem reduction) and *search*. Inference reduces the size of the search space which is then traversed in order to find a solution. It is usually done using *constraint propagation* and is interleaved with search.

2.2.1 Search

The simplest way to solve a CSP instance is to systematically generate all possible variable assignments and check if a particular combination of variables and values satisfies all the constraints. The algorithm stops either when such an assignment has been found or when all possible assignments have been tested and they all failed to satisfy at least one constraint. This algorithm is very inefficient as in the worst case it needs to check m combinations of variables and values, where m is the size of the Cartesian product of all the variable domains.

A more efficient way of searching for a solution to a CSP instance is backtracking. In its most basic form, the variables are instantiated in a sequence. As soon as all the variables of any constraint are assigned, it is checked whether this constraint is satisfied by this particular instantiation. If it is not, the process goes back to the last variable that has been assigned that has untried values and re-assigns another value to it. This can result in a much smaller number of assignments being considered. Unfortunately, the average complexity of backtracking still tends to be exponential in the number of variables [Kum92].

Several improvements on the basic backtrack algorithm have been developed. These include the so-called *look-back* methods [Bak95, GB65, FW92]. Backjumping is one of them. The principle behind this technique is that instead of re-assigning the most recently instantiated variable (that has untried values), a check is made to determine if it is better to re-instantiate another one which is higher up in the search tree. Let us give an example. Suppose the n variables $v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots v_n$ are instantiated sequentially and that v_{i+1} cannot be assigned whenever variable v_{i-1} gets assigned value 3. Suppose v_{i-1} is instantiated to 3. In the classic backtracking algorithm every possible value for variable v_i will be tested before the process backtracks to v_{i-1}. It would have been more efficient to backtrack directly to this variable. The purpose of backjumping is to find the best variable to be re-instantiated.

One of the backjumping techniques that has shown promising results is *conflict-directed backjumping* [CvB01, Pro93]. In this method a set of so-called conflicts is maintained for each variable: whenever an assignment of the current variable v_k is in conflict with an instantiation of a previous variable v_m, v_m is added to the conflict set of v_k. When every instantiation of v_k has failed, the process jumps back to the deepest variable in its conflict set, v_j. Simultaneously, v_ks conflict set (without v_j) is added to the conflict set of v_j and the search continues at v_j. This method provides some improvement on the classic backtracking algorithm, as on average fewer variable nodes are visited during search. However, this technique is not usually used in standard CSP-solvers. The so-called *look-ahead* methods involving constraint propagation (see Section 2.2.2) are more common as they are simpler to implement and, for instance, the Maintaining Arc Consistency algorithm (MAC) has been shown to be an efficient alternative to the backjumping algorithms [LBH04].

Another idea for boosting backtrack performance is by using *restarts with nogood recording* [LSTV07]. This so-called *look-back* technique was originally introduced in SAT-solvers. First, the number of maximum allowed backtracks is set. Let it be m. If during search m backtracks occur, the process is restarted with a different random variable ordering and the algorithm records a list of *nogoods*, that is instantiations of a subset of CSP variables that are not part of any solution [DF02]. Nogood recording prevents the search from traversing the same unsatisfiable search path twice. This technique was implemented in the Abscon 109 solver [LT06, LSTV07] which took part in the 2nd International CSP Solver Competition [vDLR06] and was in the top five in every category.

The order in which variables are chosen during search has a great impact on the efficiency of backtracking algorithms. One can distinguish static and dynamic variable orderings. A *static variable ordering* is usually chosen at the beginning of search and, as the name suggests, does not change during propagation. For example, variables may be ordered by the number of constraints they participate in, called their *degree*, either at the beginning of search or at the current state of search. These approaches are referred to as *degree-based orderings*. Variables may also be chosen for assignment in such a way that minimizes the width of a constraint graph [Fre82].

One of the most popular dynamic variable ordering heuristics orders variables according to the current size of their domains (*dom*) [HE80]. The variable with the smallest set of untried values is chosen first as it is most likely to fail. Other heuristics have also been proposed over the years, like *dom/deg* [FD95] which takes current domain sizes and variables' original degrees into account. None of the heuristics clearly outperforms the others, but dynamic variable orderings have generally been found to be more efficient in practice [BHLS04].

Whenever a backtracking algorithm makes a variable-value decision and checks if it satisfies all the constraints, we say that it *branches* on that decision [HM05]. There are essentially three branching schemes used in practice. In d-way branching a single value a is picked for a variable v and that variable assignment is branched on. In 2-way branching we additionally branch on the disequality $v \neq a$. Another possibility is to branch on the inequalities of the form $v > a$ and $v \leq a$.

Another key issue that influences the performance of a constraint solver is the order in which values are chosen for variable assignment [MOQ11]. Several value orderings have been tested [FD95]. The results show that the *min-conflicts* value ordering heuristic is often the most efficient one. In this method for each value of the current variable a count is kept of the number of other variable instantiations with which it conflicts. The value with the lowest count is chosen first, as it is the least probable to cause a conflict in the future.

It is worth mentioning that there are also other, less popular search methods, aside from backtracking algorithms, that are used in modern CSP-solvers. These include local search algorithms, which generally do not guarantee to find a solution even if one exists, and so are incomplete (for an overview see [HT06]).

2.2.2 Constraint propagation

All backtrack methods suffer from thrashing [Kum92], that is, they often do some redundant work when search fails several times due to the same problem. One of the main reasons is the occurrence of various inconsistencies in constraint problems. Consider, for example, a CSP instance containing a variable with domain $D_v = \{1, 2, 3, \ldots, 100\}$ ($|D_v| = 100$) and a unary constraint on that variable enforcing it to have value 100. Now, assuming an increasing value ordering is used, the first 99 values will be tried (and found to be incompatible with the constraint) before the variable is assigned the value 100. That's why almost all CSP-solvers use some form of propagation to get rid of such inconsistencies and hence reduce the search space.

In the theoretical literature problem reduction is often referred to as *consistency maintenance* or propagation [Tsa93]. The idea of using *local consistency* techniques to prune the search space is one of the oldest and most central ideas in constraint programming [Bes06]. It was introduced in 1974 by Montanari [Mon74]. Three years later Mackworth [Mac77] proposed algorithms for node-, arc- and path-consistencies which we present below.

The example we have just described lacks *node-consistency* which is defined as follows.

Definition 2.10 *A variable v is said to be* node-consistent *if and only if every unary constraint on v is satisfied by all $a \in D_v$.*

The algorithm for achieving node-consistency is very simple: one just needs to reduce the domain of each variable to the values that satisfy every unary constraint on that variable. In the example above, by enforcing node-consistency the first 99 values are removed from the domain of the variable in question before the search is started. Hence, the variable is instantiated to 100 immediately.

Another type of inconsistency that may arise is lack of *arc-consistency* as defined below.

Definition 2.11 *A variable v_i is said to be* arc-consistent *with another variable v_j if and only if for every value $a \in D_{v_i}$ there exists a value $b \in D_{v_j}$ such that the tuple (a, b) satisfies all binary constraints between v_i and v_j.*

Note that in Definition 2.11 above, if variable v_i is arc-consistent with variable v_j it does not necessarily mean that v_j is arc-consistent with v_i. Consider, for example, a CSP instance with two variables v_1 and v_2 with domains $\{1\}$ and $\{1, 2\}$ and a constraint $v_1 \neq v_2$. v_1 is arc-consistent with v_2 as for every assignment of v_1 there exists an assignment of v_2 that satisfies the constraint. However, v_2 is not arc-consistent with v_1, as for $v_2 = 1$ there is no valid value in the domain of v_1.

The most famous algorithm for achieving arc-consistency is AC3 [Mac77] presented in Algorithm 2.1. In this algorithm the REVISE procedure deletes values from the domain of variable v_i that would make edge (v_i, v_j) inconsistent in the constraint graph G. Procedure AC-3 ensures that all edges that are affected by such

a deletion are checked for arc-consistency again. The algorithm has worst-case time complexity of $O(ed^3)$ where e is the number of edges and d is the largest domain size.

Algorithm 2.1 The AC-3 algorithm [Kum92].

```
procedure REVISE(v_i, v_j);
  DELETE ← false;
  for each x in D_i do
      if there is no y in D_j such that (x, y) is consistent, then
          delete x from D_i ;
          DELETE ← true;
      end if;
  end for;
  return DELETE;
end_REVISE
procedure AC-3
  Q ← (v_i, v_j) in edges(G), i ≠ j;
  while Q not empty do
      select and delete any edge (v_k, v_m) from Q;
      if REVISE(v_k, v_m), then
          Q ← (v_i, v_k) such that (v_i, v_k) in edges(G), i ≠ k, i ≠ m
      end if;
  end while;
end_AC-3
```

Making a CSP instance arc-consistent does not immediately solve the problem, in general. Only if there is only one value left in every variable domain after making a constraint graph arc-consistent, then those values form a solution to the corresponding CSP. If the domain of some variable is empty, then the problem is unsatisfiable. If, however, some variable domain has size greater than 1, then one cannot be certain whether the CSP instance has a solution or not. Let $v \in CSP(V)$ be the variable that has values a_1 and a_2 in its domain after making the CSP arc-consistent. Let $v_1 \in CSP(V)$ and $v_2 \in CSP(V)$. It might be the case that v_1 is arc-consistent with v via value a_1 only and v_2 is arc-consistent with v via value a_2 only. Therefore for no fixed assignment of variable v is the CSP instance satisfiable.

Another form of inconsistency that may arise is a lack of *path-consistency* as defined below.

Definition 2.12 *A constraint graph is said to be* path-consistent *if and only if any pair of values allowed by the edge* (v_i, v_j) *is also allowed by all paths from* v_i *to* v_j. *A pair of values is allowed by a path from* v_i *to* v_j *if at every intermediate vertex values can be found that satisfy all the constraints along the path.*

Montanari [Mon74] presented an algorithm for enforcing path-consistency which was improved by Mackworth [Mac77]. Mohr and Henderson made further modifications [MH86] which were corrected by Han and Lee [HL88]. The algorithm they

came up with has the worst-case time complexity of $O(n^3d^3)$ where n is the number of variables and d is the largest domain size.

Several other algorithms achieving arc- and path-consistency have been proposed in the literature over the years. The most influential ones can be found in [Bes06].

Although enforcing path-consistency ensures a greater level of consistency than arc-consistency, it is still not sufficient for solving CSPs in general. Hence a question arises, if there exists a consistency enforcing algorithm that answers the question whether a CSP instance is satisfiable or not. Note that arc-consistency ensures consistency between any two nodes in the constraint graph that are connected by an edge, while path-consistency ensures consistency between any three such nodes. Hence the notion of consistency has been extended to *k-consistency* [Fre78, Coo89]. A formal definition is given below.

Definition 2.13 (*k*-consistency) *A constraint graph is said to be* k-consistent *if and only if any assignment of any set of* $k-1$ *variables that satisfies all the constraints among these variables can be extended to a set of* k *variables that satisfies all the constraints among the* k *variables. A constraint graph is said to be* strongly k-consistent *if it is* j*-consistent* $\forall j \leq k$.

A strongly k-consistent constraint graph with k vertices has a solution which can be found immediately, as the following result indicates:

Theorem 2.14 *[Fre82] If a constraint graph is strongly* j*-consistent and* $j > w$ *where* w *is the width of the constraint graph, then a search order exists that is backtrack free.*

The time complexity of the optimal algorithm for achieving strong j-consistency is polynomial for any fixed j [Coo89]. The major drawback of making a constraint graph j-consistent for $j > 2$ is that, when running an algorithm for achieving the desired level of consistency, the width of the constraint graph may increase. Hence a higher level of consistency then needs to be achieved. This approach is very costly in terms of efficiency and hence it is not generally used in practice.

Till now we have only considered binary constraint graphs. However, the notion of consistency can be extended also to non-binary constraints.

The equivalent of arc-consistency for non-binary CSP instances is called *generalised-arc-consistency* (GAC) as defined below.

Definition 2.15 (GAC) *A variable* v *is said to be* generalised-arc-consistent (GAC) *with a constraint if and only if for every value* $a \in D_v$ *there exists an assignment of all the other variables of the constraint such that it is satisfied.*

Several algorithms for enforcing GAC have been developed. They are all based on some underlying arc-consistency algorithm [BR97, BRYZ05, GJMN07]. Some standard CSP-solvers that implement GAC include Minion [GJM06], Choco [CHOCOt08] and Abscon [LT06].

There are several other types of consistencies that are relatively cheap to achieve. These include singleton consistencies. The notion of a *singleton consistency* is general and applicable to all levels of consistency [PSW00].

Definition 2.16 ([PSW00]) *A problem is said to be* singleton arc-consistent *if and only if it has non-empty domains and for any instantiation of a variable, the resulting subproblem can be made arc-consistent with all domains non-empty.*

Definition 2.17 ([PSW00]) *A problem is said to be* generalised singleton arc-consistent *if and only if it has non-empty domains, and for any instantiation of a variable the resulting subproblem can be made generalised-arc-consistent with all domains non-empty.*

An algorithm for singleton consistency is as follows: first we enforce some level of consistency, then we test each instantiation of a variable for that consistency level. If a value is not singleton consistent, it is removed and the desired level of consistency is established again. This process is repeated until all values are singleton consistent [DB97, DB05]. Singleton arc-consistencies have been used, for instance, in the Abscon [LT06] and Casper [CBA05] constraint solvers.

In order to identify as quickly as possible which constraints need to be propagated the idea of *watched literals* has been introduced. The so-called *2-watched literals* scheme has been very successful in SAT (see Section 2.4). The CSP-solver Minion [GJM06] also takes advantage of watched literals. For instance, the Boolean sum constraint is solved using the following technique. If the constraint requires that at least c variables are constrained to take value 1, then $c + 1$ variables that still can be assigned value 1 are watched. Once one of these variables gets assigned to 0, another one is looked for. If none can be found, all watched variables are assigned 1. The good thing about watched literals, is that they are *backtrack-stable*. This means that the selection of variables being watched does not need to be changed when the algorithm backtracks.

Aside from constraint propagation, there are also other methods for reducing the search space. One of these is *symmetry breaking*. In many constraint problems one can find some symmetries. Consider, for instance, three variables v_1, v_2 and v_3 with domain D and constraints $v_1 \leq v_2$ and $v_1 \leq v_3$. Note that once it is detected that either v_2 or v_3 cannot be assigned some value, say $a \in D$, the other variable cannot take value a as well. Clearly, if more symmetries are involved, identifying them might significantly reduce the search space, as it suffices to propagate only one set of variables from each symmetrically equivalent class. The symmetry in the above example is called a variable symmetry, as there is a permutation of variables that leaves the CSP invariant [Pug05]. A permutation of values that leaves a given CSP invariant is called a value symmetry [Pug05]. These two symmetries are called *constraint symmetries*. *Solution symmetries* have also been identified [CJJ+06] and they preserve the set of solutions of a CSP instance. Symmetry breaking has been used, for instance, in the Choco [CHOCOt08] and the newest version of Abscon [LT06] constraint solvers.

2.2.3 *Combining propagation and search*

A major difference between the various algorithms used in CSP-solvers is the way and extent to which constraint propagation is incorporated into search. A desired level of consistency is achieved at first and then the search phase begins. Usually solvers enforce arc-consistency, so we will use this example to illustrate the solving process. Propagation is triggered on variable instantiation. Arc-consistency is established at that variable, that is, at the node in the search tree that corresponds to that variable. There are three scenarios afterwards: either the sizes of all the domain variables are equal to 1, or else some variable domain is empty, or else some variable domain size is greater than 1. In the first case we have found a solution. In the second case the variable instantiation that triggered propagation is invalid, so we backtrack and another value for that variable is tried. In the third case a new variable with domain size greater than 1 gets assigned which triggers propagation as before.

Until the mid-1990s it was believed that full arc-consistency would be too costly to achieve, so only partial arc-consistency was enforced during search. In 1994 Sabin and Freuder [SF94] showed that algorithms that establish full arc-consistency can be much more efficient, especially when solvers are run on hard problem instances. In modern constraint solvers usually each type of constraint has an associated propagation algorithm which achieves the desired level of consistency for that constraint.

2.3 Boolean satisfiability problem (SAT)

Definition 2.18 *The problem of deciding whether there is a variable assignment that satisfies a propositional formula is called the* Boolean satisfiability problem (SAT).

SAT is known to be NP-complete [Coo71]. However, SAT-solvers have been widely used in practice as they can often efficiently handle problems with thousands and sometimes even millions of variables [ZM02].

2.4 SAT-solvers

SAT-solvers generally input propositional formulae in the form of conjunctions of disjunctive clauses (CNF). The main advantage of using CNF as the standard form of solver input can easily be seen when testing problem instances that have no solution. Once some disjunctive clause is found to be unsatisfiable, then the whole problem becomes unsatisfiable. There is no need for further checks. Note that any propositional formula can be transformed into CNF in linear time by introducing auxiliary variables as long as the original formula contains Boolean operators that

have linear clausal encodings [Pre09]. Such operators include $\wedge$ (*and*), $\vee$ (*or*), $\rightarrow$ (*implies*) and $\neg$ (*not*). Moreover, the standardised input format allows for quick and easy comparison of SAT-solvers.

As far as the solving techniques are concerned, modern standard SAT-solvers still use some variation of the algorithm developed by Davis, Putnam, Logemann and Loveland (DPLL) in the 1960s [DLL62, DP60]. The pseudo-code is shown in Algorithm 2.2, and the various subroutines mentioned are described in more detail below.

Algorithm 2.2 The DPLL algorithm [ZM02].

```
DPLL
  status = preprocess();
  if status ≠ UNKNOWN then
      return status;
  end if;
  while true do                                                    Decision
      make_branch_decision();
      while true do                                          Unit Propagation
          status = deduce();
          if status == INCONSISTENT then                            Conflict
              resolved = analyse_conflict_and_backtrack();
              if not resolved then
                  return UNSATISFIABLE;
              end if;
          else if status == SOLUTION_FOUND then
              return SATISFIABLE;
          else
              break;
          end if;
      end while;
  end while;
```

The aim of the pre-processing stage is to simplify the input formula. This usually means reducing the number of variables and adding some clauses. This can be done by using *propositional resolution* as defined below.

Definition 2.19 ([Rob65]) *The process of using a deduction rule to substitute* $(x \vee C_1) \wedge (\neg x \vee C_2)$ *with* $(C_1 \vee C_2)$*, where* C_1 *and* C_2 *are propositional clauses and* x *is a Boolean variable, is called* (propositional) resolution*. The resultant clause is called the* resolvent*.*

Resolution is known to be a refutation complete and sound proof system for CNF formulae.

Definition 2.20 *A* resolution proof *of a clause* C *from a set of initial clauses* Φ *is a sequence of clauses* $C_1, C_2, \cdots, C_m$*, where* $C_m = C$ *and each* C_i *follows by the resolution rule from some collection of clauses, each of which is either contained in*

Φ or else occurs earlier in the sequence. If C_m is the empty clause, then we say that the derivation is a resolution proof (or refutation) of Φ.

Another example of a simplification rule is the *pure literal rule* [GPFW96]. If a variable is never negated (that is occurs as a positive literal only) or is always negated (that is occurs as a negative literal only) then it can be assigned value *True* or *False* respectively, and hence all the clauses in which it occurs are satisfied. For instance, consider the two clauses $v_1 \vee \neg v_2$ and $v_1 \vee v_3$. v_1 occurs as a positive literal only, so it can immediately be assigned value *True*.

It is worth mentioning that simplification does not necessarily reduce the search space [LMS01]. Pre-processing also involves choosing an initial variable order. It turns out that randomisation at this stage often produces quite good results [GSK98].

2.4.1 Search

At the *make_branch_decision* stage a free variable is chosen for assignment. One of the most successful orderings is the Variable State Independent Decaying Sum (VSIDS) ordering used in the Chaff SAT-solver [MMZ+01]. To implement this ordering, for each literal a count is kept of the number of unresolved clauses in which that literal occurs. When at the deduction stage clauses are added, the literal counts are increased accordingly. The variable that appears in the literal with the highest count is chosen for assignment. Branching heuristics based on literal count have widely been used since the 1990s and include the largest individual sum heuristic (DLIS) and its variations [MS99]. What VSIDS adds to the picture is periodically dividing all the counts by some constant number. This method gives priority to variables that constrain the biggest number of clauses and have been recently active.

The idea of *activity* has also been used in the BerkMin SAT-solver [GN02]. It uses a technique similar to VSIDS, additionally increasing the counts of literals appearing in a clause that evaluates to false under the current assignment.

Once a variable is chosen for assignment, a decision level is assigned to it. The decision level informs the solver at which stage of search the decision for that particular variable was made. For instance, any variable that has been assigned a value at the pre-processing stage will be assigned 0 (or Top) decision level. The first variable that gets picked for assignment will have decision level 1. It is worth mentioning that if any variable gets assigned at the propagation stage, it will be given the same decision level as the variable that caused that assignment. For instance, given current decision $x_1 = \mathit{False}$ and clause $x_1 \vee \neg x_2$, x_2 must be assigned value *False* for that clause to be satisfied. Hence if x_1 has decision level, say 5, so will x_2. More details on propagation will be given in the next section.

The symmetry breaking technique used in CSP-solvers has recently been applied to SAT-solvers. The symmetries here mean permutations of literals which do not change the CNF formula. They just re-arrange the clauses and literals within the

clauses. Examples of SAT-solvers which exploit symmetry in CNF formulae include SymChaff [Sab05] and Shatter [AMS03].

2.4.2 Boolean constraint propagation (BCP)

At the deduction stage of the DPLL algorithm, variables are assigned according to some implication rules, until a conflict or a solution is found. This process is known as *Boolean constraint propagation (BCP)* and is carried out by the *deduce* subroutine in Algorithm 2.2.

The one rule that all SAT-solvers use is the *unit propagation rule* (or unit resolution):

Definition 2.21 (unit propagation) *The* unit propagation rule *states that if all but one literals in a clause evaluate to false under a partial assignment, then the last free literal has to be true. Such a clause is called a* unit clause.

For instance, consider the clause $v_1 \vee \neg v_2 \vee v_3$ and partial assignment $v_1 = \mathit{False}$ and $v_2 = \mathit{True}$. Then v_3 must be assigned value *True* for the clause to be satisfied.

Definition 2.22 *A* conflicting clause *is a clause that evaluates to false under the current assignment.*

The aim of the Boolean constraint propagation (BCP) stage in a SAT-solver is to identify unit and conflicting clauses as soon as possible. As the BCP stage is the most time-consuming part of the DPLL algorithm [BHZ06], a lot of research has been devoted to improving it. One method is based on keeping the count of true and false literals for each clause. By knowing the clause size it is then easy to check whether after some variable assignment the clause becomes unit or conflicting. As this method requires many counters to be updated whenever a variable gets assigned, it is usually not the most efficient one.

In the last decade algorithms have been developed that use the observation that it suffices to keep track of only two non-false literals per clause. Note that as long as a clause contains two non-false literals then it is neither unit nor conflicting. One method uses head/tail lists [ZS00]. For each clause the first and last non-false literals are kept track of by using head and tail pointers respectively. Variable assignments that do not affect those literals or assign them value *True* do not trigger any action. Once one of the literals pointed to by one of the head or tail pointers evaluates to false, the next non-false literal is looked for. If one is found that is not pointed to by both head and tail pointers then the algorithm continues, if head pointer meets the tail pointer (or the other way round) then, depending on the value of the literal they point to, we get either a conflicting or unit clause. A drawback of this method is that when the solver performs backtracking, work is done to move back pointers to their original positions. This problem has been resolved in the Chaff solver [MMZ+01] which uses the so-called *watched literals*.

In the *2-watched literal* scheme pointers to any two non-false literals are kept for each clause. Each variable has two lists of pointers pointing to positive and negative watched literals corresponding to it. Again, once after some variable assignment any of the watched literals evaluates to false, another one is looked for in the clauses which contain that literal. If one is found that is different from the other watched literal, the process continues. If the only choice is the other watched literal, then it is set to *True* if it has not already been assigned that value. If no non-false literal is found, then the clause is conflicting. The good thing about this approach is that during backtracking the pointers are not changed. This is because the literals being watched are the last to be assigned to *False*, so at backtracking they will become unassigned and hence can still be watched.

2.4.3 Conflict analysis

The last phase of the SAT solving process is conflict analysis and backtracking. It may happen that under the current assignment and after the propagation stage all literals in a clause evaluate to false. We say that a conflict occurred. SAT-solvers use learning in order to prevent the same conflict from happening again after backtrack. At this stage the solver also tries to figure out to which decision level in the search the process should backtrack. The simplest method for solving conflicts is trying the other value for the variable that has been assigned most recently as it directly caused the conflict. If both values cause conflict, another value is tried for the variable that has been assigned just before the latest one. This method thus backtracks just one decision level up the search tree and hence is not the most efficient. The idea of "jumping back" more than one decision level up was first introduced in CSP-solvers. It has since been used in SAT-solvers such as GRASP [MSS96].

In modern SAT-solvers learning is used to analyse the current conflict. In order to resolve a conflict, one needs to know what caused it, that is, what was the trail of decisions that forced the last non-false literal in the conflicting clause to be unsatisfied. All variable assignment decisions, made both at the search stage and inferred during propagation, can be represented by a so-called *implication graph*. Nodes of an implication graph represent variable assignments. By each node the decision level of each assignment is kept. A directed edge is drawn from node x to y if the unit resolution rule implies that assignment y must be *True* given x. Consider the clauses $v_4 \vee \neg v_2 \vee \neg v_1$, $v_6 \vee \neg v_3 \vee \neg v_5$, $v_5 \vee v_2$, $\neg v_2 \vee v_1$ and $v_4 \vee \neg v_2 \vee \neg v_1$. Suppose we first made a decision $v_4 = \mathit{False}$. Next, we assign $v_6 = \mathit{False}$ and then $v_3 = \mathit{True}$. The resulting implication graph is shown in Figure 2.2. Note that a conflict occurred at v_1, as once v_1 is assigned *True* by $\neg v_2 \vee v_1$, the clause $v_4 \vee \neg v_2 \vee \neg v_1$ evaluates to false. Once a conflicting clause is found under the current assignment, a so-called *conflict clause* is added to the problem.

Definition 2.23 *In the context of conflict analysis in SAT-solvers, a* conflict clause *is said to be a clause that can be deduced from the current conflict.*

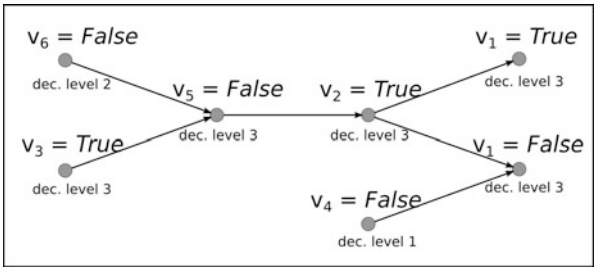

Fig. 2.2 An example implication graph.

For instance, in the example described above, a conflict clause could be $v_5 \vee v_4$. Such a learned clause significantly helps prune the search space after backtrack. Frequently more than one conflict clause can be deduced from the current conflict. Hence some sort of selection strategy needs to be used.

Single assignments at the current decision level that imply current conflict are called *Unit Implication Points* (UIPs) [BHZ06]. In other words, if conflict is reached at vertex y, then x is a UIP if and only if any path from the decision variable (that is a variable that is assigned at the search stage) of the decision level of x to y needs to go through x. For instance, there are three implication points in Figure 2.2, as a conflict occurs if at the current decision level either v_3 gets assigned *True* or v_5 gets assigned *False* or v_2 gets assigned *True*.

The most common selection strategy for choosing conflict clauses uses clauses that contain a variable that occurs in a UIP. As there may be several UIPs, a question arises which UIP to choose. Several learning heuristics have been tested and the so-called FirstUIP scheme often seems to be the best one [ZMMM01]. In this scheme a conflict clause is added that contains a variable corresponding to the UIP that is closest to the conflict. Moreover, such a clause must be an asserting clause as defined below:

Definition 2.24 *An asserting clause is a conflict clause that contains only one variable that is assigned at the current decision level.*

In the example from Figure 2.2 three asserting clauses could be derived: $\neg v_2 \vee v_4$, $v_5 \vee v_4$ and $\neg v_3 \vee v_6 \vee v_4$. In this case the first one would be added to the SAT instance being solved, as in the implication graph UIP $v_2 = \mathit{False}$ is the closest to the conflict. An example of a non-asserting clause would be: $\neg v_2 \vee v_5 \vee v_4$.

Finally, the algorithm backtracks to the decision level that is second-highest of all the literal decision levels in the *asserting clause*. Note that such a clause can always be found as at least the clause composed of all the decision assignments made so far is an asserting clause.

In order to describe in what way the asserting clauses are searched for, a definition of the *antecedent clause* needs to be introduced:

Definition 2.25 *The* antecedent clause*, in the context of conflict analysis used in SAT-solvers, is said to be the clause that directly triggered the latest assignment in*

the current clause and includes the variable contained in the literal that has been assigned last.

Once a conflicting clause is found each literal is substituted with its antecedent clause. If the resulting clause is not an asserting one, each of its literals is again substituted with its antecedent clause. The process continues until an asserting clause is found.

During conflict resolution many clauses might be added. Hence SAT-solvers usually implement some mechanism for removing redundant conflict clauses. BerkMin [GN02] solver counts the number of conflicts each clause has been involved in recently. Another heuristic deletes clauses that contain many non-false literals [BS97].

2.5 Hybrid solvers and SMT-solvers

SAT-solvers can actually be used for deciding the satisfiability of CSP instances. Such SAT-based solvers first translate the input instance into SAT and then use a SAT-solver. We will call such solvers SAT-based constraint solvers. Since constraint solvers and SAT-solvers share so much in common, hybrid solvers have also been recently introduced. One approach is to implement some domain-specific reasoning within a SAT-solver, another one is to implement a SAT engine within a constraint solver. The two architectures are discussed in [FS09]. The second approach has been used, for instance, as an extension of the G12 constraint solver [FS09]. The first approach is seen in another type of solvers coming from the research area of *SAT Modulo Theories* (SMT) [NOT06].

SMT-solvers decide the satisfiability of a ground first-order logic formula with respect to a background theory. Examples of theories include the theory of integers, theory of arrays or bit vectors. The formula $v - w \leq 5 \vee \neg x \vee v = f(w)$ is an example SMT clause, where v, w and x are variables (with x Boolean), and f is some unspecified function. At the core of an SMT-solver is a SAT-solver. The *theory solvers* simply check if the current assignments are feasible and can infer new facts which are then encoded as new clauses and passed onto the SAT-solver. SMT-solvers have been very successful in solving problems from the areas of hardware and software verification [NOT06].

2.6 Summary

Since the 1960s a lot of research has been going on in the area of constraint satisfaction and Boolean satisfiability. This has led to the development of various algorithms trying to solve the general CSP and SAT problems. The most efficient ones combine search and propagation. In the case of constraint solvers, constraint

propagation is the most significant part of the solving process, as most efficiency gains are achieved thanks to pruning the search space by applying propagation techniques. As far as SAT-solvers are concerned, unit propagation and conflict-directed learning play the most important roles.

There are many similar ideas used in both CSP and SAT-solvers. In both cases variable order has an impact on the solver performance [BHLS04, MMZ+01]. Some form of learning has been applied both in CSP (nogoods [LSTV07]) and SAT (conflicts [ZMMM01]). However, it is worth mentioning that the introduction of learning techniques into SAT had a major influence on improving solver performance, whereas it has not had such a huge impact on boosting CSP-solver runtimes. The idea of non-chronological backtracking used in CSP [Bak95, GB65] has also had its application in SAT [ZMMM01]. In the last few years the watched literals method has significantly boosted the performance of standard SAT-solvers [MMZ+01] and recently they have been applied in some constraint solvers [GJM06].

Although one can find many similarities between CSP and SAT-solvers, the translation of ideas between constraint satisfaction and Boolean satisfiability is not always obvious, if at all possible. One reason is that CSP contains a broad class of problems which includes SAT. CSPs are closer in their description to real-world problems. Moreover, most constraint solvers can be tuned, that is the user can choose a specific variable ordering or search strategy. SAT-solvers, on the other hand, typically act as a black box. This difference might have come about as there is no standardised input for CSP-solvers and they can model a broader set of problems than SAT. Hence some CSP solving algorithms take specific problem features into account. There exist algorithms whose sole purpose is efficiently solving a certain type of constraints, like the global cardinality constraint [QGLOvB05] or the so-called constraints of difference [Rég94].

Another difference between CSP and SAT-solvers is the impact of value order on their performance. It does not have that much significance in SAT-solvers as the variables can only take two values: *True* or *False*. Hence not much research has been done on which of those two values is worth trying first for a variable. Therefore it is hard to say what performance gain, if any, could be achieved if a particular value order were chosen.

Moreover, in the theoretical literature one might find some ideas for boosting solver performance that have not yet been implemented. For instance, most CSP-solvers do not investigate the underlying structure of an instance. One reason is that finding such a structure might be tricky and costly in terms of time and space, and hence outweigh the possible performance gains. Another reason is the way propagation is implemented in CSP-solvers. Constraints "talk to each other" through the domain. Essentially what a constraint solver does is remove unsatisfiable domain values. There is no way, with current solver architectures, of combining information from two or more constraints and determining the satisfiability of the problem based on their structure.

Throughout the years the core algorithm for SAT has been DPLL, while there is no such standardised algorithm for CSP. In the last few years CSP research has

even focused on finding the best algorithms for particular constraints rather than on a general model for propagating all constraints [GJM06, NSB+07].

Current SAT-solvers are considered to be extremely efficient. Hence some researchers have developed CSP to SAT translations and used a SAT-solver engine for dealing with CSPs. Such translations often produce huge SAT instances. Interestingly enough, such SAT-based solvers did quite well in CSP-solver competitions [vDLR09, vDLR08, vDLR06]. They even performed well on the problem instances containing highly structured so-called global constraints, which are said to be the natural domain of CSP-solvers.

Furthermore, because of these close connections between CSP and SAT, hybrid solvers have been developed. Moreover, the area of SMT solving tries to incorporate the best of the two worlds: domain-specific reasoning and fast SAT solving.

Summing up, in the quest to find efficient algorithms for CSP and SAT-solvers, two separate areas of research have developed, namely constraint satisfaction and Boolean satisfiability. Although there may be many differences in their approaches to problem solving, they both benefit from each others' findings and a thorough comparative study of the two areas might lead to further useful developments.

http://www.springer.com/978-3-319-21809-0

Bridging Constraint Satisfaction and Boolean Satisfiability

Petke, J.

2015, XI, 113 p. 19 illus., Hardcover
ISBN: 978-3-319-21809-0