University of Alberta

SAT with Global Constraints

by

Md Solimul Chowdhury

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2011
Edmonton, Alberta

Abstract

Propositional satisfiability (SAT) has been a dominant tool in solving some practical NP-complete problems. However, SAT also has a number of weaknesses, including its inability to compactly represent numerical constraints and its low level, unstructured language of clauses. In contrast, Constraint Programming (CP) has been widely used for scheduling and solving combinatorial search problems. In this thesis, we develop a tight integration of SAT with CP, called SAT(gc), which embeds global constraints, one of the most efficient features of CP, into SAT. We devise a DPLL-based algorithm for a SAT(gc) solver. A prototype system is implemented by integrating the state of the art SAT solver Zchaff and the generic constraint solver named Gecode. Experiments are carried out for benchmarks from puzzle domains and planning domains to reveal insights in compact representation, solving effectiveness, and novel usabilities of the new framework.

Acknowledgements

I am very thankful to my supervisor, Dr. Jia-Huai You, for all the encouragement, guidance and support throughout this work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Constraint Programming and Boolean Satisfiability

Constraint Programming (CP) [34, 42] is a programming paradigm, developed for studying and solving constraint problems. CP is closely related to declarative programming, where an end user only states the problem to be solved, using a knowledge representation language, without stating how the problem is solved. That is, the user does not need to devise an algorithm as part of a solution. It is the underlying solver that solves the represented problem using a generic search engine.

CP has wide applications. It has been applied to solving many practical problems from domains of scheduling, planning, and verification [49]. Generally, these constraint problems are combinatorial search problems, which means it is unlikely that an efficient polynomial time algorithm exists for solving constraint problems.

Historically, CP is developed from the Constraint Satisfaction Problem (CSP) [13, 42],[1] where a problem is represented by a collection of variables, each of which is over a finite domain, and a collection of constraints. A constraint is a relation over a subset of the CSP variables, which expresses allowed combinations of values for these variables (or alternatively, disallowed combinations of values).

For practical applications, languages for CP have been developed to facilitate the definitions of constraints in terms of primitive constraints and built-in constraints. One kind of these built-in constraints are called *global constraints* [48]. In general, a global constraint is a pre-defined and specially implemented constraint over a non-fixed number of variables. The use of global constraints is two-fold. Firstly, it facilitates problem representation. Secondly, the processing of a global constraint is usually more efficient than an equivalent representation using primitive constraints. This is because a global constraint is typically implemented by using special data structures and dedicated constraint propagation mechanisms (see, for example, [6, 26]). In the current practice of CP, the basic methodology of programming is to compose constraints using primitive and built-in constraints, most of which are in the form of global constraints.

Another way of solving combinatorial search problems is Boolean Satisfiability (SAT), in which a problem is represented by a collection of Boolean clauses, called a *formula*. To solve a SAT formula, we need to determine whether there is a truth value assignment that satisfies all the clauses.

Recent research in SAT solving indicates that the research on the central part of SAT solving has become saturated, as DPLL based modern SAT solvers have very efficient implementations, with their ability to solve a SAT problem using the technique of conflict directed

[1] Due to this, in this thesis CP and CSP are used to refer to the same style of constraint solving, though in general CP refers to a programming paradigm and CSP is meant to be a model of constraint solving.

backtracking and learning. However, it is not clear where the further breakthroughs would come from. In the past few years, a trend of incorporating efficient and useful features of other frameworks into SAT solver has been developed. For example, in SAT Modulo Theory (SMT) [37], theory solvers of various kinds are incorporated into a SAT solver, where part of the problem is encoded in an embedded theory and solved by a dedicated theory solver.

SAT and CSP share some common traits towards problem solving. But both have some inherent disadvantages. For example, unlike the SAT framework, CSP requires heavy tuning on the problem representation, such as the variable selection strategy for a given problem instance and choices of consistency techniques with a balance of search space pruning power versus overhead. In general, the effectiveness of CSP comes from representing problems with variables over non-trivial domains. This is due to the fact that in search space pruning for CSP, the key idea is what is called *domain reduction* - removing domain values that are inconsistent with the current partial assignment. Thus, CSP does not seem to be an effective method for representing problems that heavily rely on variables of two values, true and false, i.e., propositional variables. On the other hand, the language of SAT is based on the language of clauses, which is arguably a low-level, unstructured language. Some researchers even term SAT as the "assembly language" of hard problems.

For some applications, such as planning, SAT solvers are typically more efficient than CSP solvers, while for some other types of problems, like scheduling, the CSP approach is better suited than the SAT approach. The primary drawback for scheduling in SAT is due to the difficulty of representing numeric constraints in SAT. For a CSP variable over a large numeric domain, while a CSP solver tries to find values of variables from their domains so that the stated constraints are satisfied, in SAT one uses a Boolean variable to represent that a CSP variable takes a value, for each value in the domain. This can result in an exponential blow up of the resulting size of a SAT representation. To represent a numeric constraint this way often needs to enumerate all possible values of a domain. In other words, a numeric constraint may have to be solved before we know how to represent its solution by propositional clauses.

To deal with numeric constraints, the SAT community has moved to a different direction - *pseudo Boolean constraints*, where constraints are expressed by linear inequalities over sum of weighted Boolean functions (see, e.g., [10]). Pseudo Boolean constraints extend SAT as the latter can be seen as a special case. Pseudo Boolean constraint solvers have been built and competitions held as a special event of SAT Solver Competition.[2]

In this thesis we study how to incorporate some efficient features of CSP solving into SAT. Two possible modes in which SAT solver and CSP solver can be integrated are *loose integration* and *tight integration*. In a loose integration, calls to a CSP solver can be largely handled separately from the reasoning by a SAT solver. A tight integration implies tight interleaves between CSP and SAT solvers. In other words, a tight integration is an integration of solvers, while a loose integration consists of a collection of largely independent calls to different solvers. In general, a loose integration is insufficient in handling inter-dependent constraints, where a solution to a constraint may not lead to a solution to the overall problem, and backtracking is necessary. Any attempt to handle this automatically leads to a tight integration.

In this thesis, we pursue a tight integration of SAT and CSP. Tight integration poses a number of challenging issues like, how to represent a SAT problem in the presence of global constraints, and how deductions, conflict analysis and backtracking with learning can be performed in the presence of global constraints. In our work, we develop a framework to incorporate CSP style constraint solving into SAT solving with the anticipation that this tight integration will enhance the usability of SAT solver and increase its efficiency for some application domains.

[2] http://www.cril.univ-artois.fr/PB11/

A noteworthy point to mention here. The benchmarks that are used in the recent pseudo Boolean solver competitions involve only numerical constraints. If the pseudo Boolean solving community sticks to their current focus, then pseudo Boolean solvers will become more efficient in solving problems involving only numerical constraints. These include problem domains like scheduling, numerical optimization, etc. However, there is no guarantee that a problem that can be solved efficiently by a SAT solver can be solved in the same way by a pseudo Boolean solver, due to different underlying problem representation and solving techniques. For example, SAT planning is known to be a competitive approach to AI planning [22]. But there is no guarantee that the pseudo Boolean approach can solve the planning problem with the same level of efficiency. In other words, not all the problems that can be solved efficiently by the SAT approach can be solved with the same efficiency by the pseudo Boolean approach. In contrast, our developed framework is a true extension of SAT and CP, in that when one is absent, the system runs like the other.

1.2 Contributions

The contributions of this thesis are as follows:

1. We provide a critical review on the state of the art DPLL based SAT solving, CSP solving, and SMT solving. This material also serves as the background for the technical development of the thesis.

2. We develop a framework of integrating CSP and SAT, by embedding global constraints into SAT. The resulting framework takes the advantage of the modeling power, as well as efficient processing, of CSP, in terms of global constraints, and makes it available to SAT solving.

3. Our framework is closely related to SMT. We identify similarities and differences between the two.

4. To demonstrate the feasibility of this framework, we implemented a prototype system, called SATCP, in which we integrated the modern DPLL based SAT solver, ZCHAFF, and a generic constraint solver, GECODE. Since user-defined constraints can be called in the same way as global constraints, SATCP can be seen as embedding GECODE in SAT.

5. To demonstrate the utilities of the framework, we report experimental results on four benchmarks, two from the puzzle problems domain and two from the planning domain.

1.3 Thesis Layout

The thesis is organized as follows. The next chapter presents a succinct review on modern DPLL based SAT solving, followed by an introduction to constraint programming in Chapter 3 and SMT in Chapter 4. Chapter 5 presents an embedding of global constraints into SAT, called SAT(gc), and Chapter 6 provides some implementation details in integrating ZCHAFF and GECODE. We carry out experiments with four benchmarks for our implemented prototype system, two of which are taken from the puzzle problem domain (Latin Square Problem and Magic Square Problem), and the other two from the planning domain (Ferry Planning Problem with numerical constraints and Block Stacking Planning Problem with numerical constraints). Chapter 7 describes each of the benchmarks, their encoding in SAT(gc), experimental results, and an analysis of the results. Chapter 8 concludes our work and points to future directions.

Chapter 2

Boolean Satisfiability

Given a propositional formula of collection of clauses, the task of determining whether there exists a variable assignment such that the formula evaluates to true is called the Boolean Satisfiability Problem, abbreviated as SAT [53]. Let V be a finite set of propositional symbols, called *variables*. For any $v \in V$, v and $\neg v$ are called the *literals* of v denoting the positive and negative phases of variable v, respectively. A *clause* is a disjunction of literals $l_1 \vee \cdots \vee l_n$. A *unit clause* is a clause with one literal. In general, a unit clause may also refer to a clause with n literals, where $n-1$ of them are false in the current partial assignment. A SAT instance is a propositional formula consisting of a conjunction of clauses, which is also called a Conjunctive Normal Form (CNF) [37].

Example
The following formula F (taken from [29]) is a CNF formula with three variables a, b and c.

$$F = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$$

The Boolean satisfiability problem for formula F is to determine whether there exist a truth assignment for a, b and c, so that the formula F is satisfied. Figure 2 shows a complete search tree which can be used to determine the satisfiability of the formula. The branches leading to green leaves and red leaves are the satisfiable and unsatisfiable branches respectively.

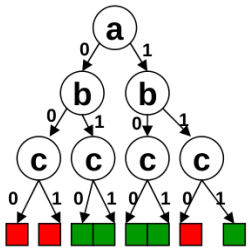

Figure 2.1: Search tree for finding the satisfiability of formula F.

2.1 Application of SAT

SAT solvers have been used in many application areas. They are used in areas like automated theorem proving, verification, artificial intelligence, and electronic design automation. In the following we shall briefly mention some of these [32].

1. Automated test pattern generation: Fabricated integrated circuits may contain defects, which in turn may cause circuit failure. Automated test pattern generation (ATPG) is used for detecting fabrication defect on circuits.

2. Combinatorial equivalence checking: After designing two circuits, an essential task is to check their equivalence. That is to see, whether or not they are providing the same output. The problem of equivalence checking can be formulated as a SAT instance on which a SAT solver is employed to give a YES or No answer, i.e., the SAT instance is satisfiable if the circuits are equivalent, and unsatisfiable otherwise.

3. Bounded Model Checking: One important usage of a SAT solver is bounded model checking. Given a transition system, a temporal logic formula, and a user-supplied time bound, the bounded model checking constructs a SAT formula, which is satisfiable if and only if the temporal logic formula is valid along a path of certain length and unsatisfiable otherwise.

4. AI Planning: SAT based AI planning was motivated by bounded model checking. SAT based planning decides whether it is possible to reach a goal state from an initial state within a specified number of steps by performing a series of actions. If one of the goals is reachable, the SAT instance is satisfiable, otherwise it is unsatisfiable.

2.2 Boolean Satisfiability Solvers

The Boolean Satisfiability Problem was the first problem to be proved to be NP-complete. This means, any problem in NP in theory can be solved by encoding the problem as a SAT instance and by invoking a complete SAT solver on it.[1] On the other hand, SAT is a hardest problem in NP. Despite of this NP-completeness, due to the large applicability of SAT, in the last few decades extensive research for finding efficient algorithms to solve many interesting SAT instances was undertaken. Almost all complete modern SAT solvers are based on what is called DPLL, reflecting the fact that it was invented by the four researchers, Davis, Putnam, Logemann, and Loveland [28]. In this section, we shall discuss the basic DPLL framework and the evolution of its components.

2.2.1 The basic DPLL framework

In 1960, Davis and Putnam proposed a resolution based algorithm for solving the SAT problem, abbreviated as DP algorithm [11]. This algorithm is the original algorithm for many SAT solvers. However, this resolution based algorithm suffers from memory explosion. Two years later, Davis, Putnam, Logemann and Loveland proposed a modified version of DP algorithm, widely known as DPLL algorithm [28], that uses search instead of resolution. This overcomes the problem of memory explosion.

In order to satisfy a CNF formula, each and every clause must be satisfied individually. If there exists a clause, where all of its literals are assigned to the value 0 (we use *true* and 1 interchangeably, and $false$ and 0 interchangeably), then it is said that the solver is in conflict and the current variable assignment cannot be extended so that the formula can be satisfied. A clause, which has all of its literals assigned to value 0, is called a *conflicting clause*.

The recursive version of the DPLL algorithm is given in Algorithm 1.

The DPLL algorithm is called with a CNF formula and with a variable assignment (initially empty). This variable assignment is called a *partial assignment*. The algorithm first makes all the possible deductions on the input formula, based on the current partial assignment, and adds these new deductions to the current partial assignment, thus producing a new assignment. A process of deduction is constraint propagation (in most SAT solvers this is done by what is called *unit propagation*). If the formula is either satisfied or unsatisfied under the current variable assignment, the recursion ends. Otherwise, it will select a free variable from the formula and branch on it for both of the phases.

[1] A complete solver guarantees an answer YES when the given SAT instance is satisfiable, and NO otherwise.

```
1 DPLL(formula, assignment)
    necessary = deduction(formula, assignment)
    newasgnmnt = union(necessary, assignment)
    if is_satisfied(formula, newasgnmnt) then
      return SATISFIABLE
    end if
    if is_conflicting(formula, newasgnmnt) then
      return CONFLICT
    end if
    var = choose_free_variable(formula, newasgnmnt)
    asgn1 = union(newasgnmnt, assign(var, 1))
    if DPLL(formula, asgn1) == SATISFIABLE then
      return SATISFIABLE
    else
      asgn2 = union(newasgnmnt, assign(var, 0))
      return DPLL(formula, asgn2)
    end if
```

Algorithm 1: The Recursive Version of the DPLL Algorithm

An iterative and improved version of DPLL algorithm has been developed [33], which, unlike the recursive version of DPLL algorithm, uses non-chronological backtracking. Algorithm 2 describes such an algorithm.

Different DPLL based SAT solvers can be described by giving variations they have made on the functions described on Algorithm 2. From now, our discussion on the DPLL framework will be based on this version of the DPLL algorithm.

The algorithm described on Algorithm 2 starts by executing the *preprocess*() function on the CNF formula to do some initial deductions. If *preprocess*() determines the satisfiability of the CNF formula, then the algorithm simply returns the outcome. Otherwise, the execution will enter the outer while loop and call *decide_next_branch*() to choose a free variable from the pool of free variables to assign it to a value. This operation is called a *decision* and each *decision variable* is associated with a *decision level*, which starts from 1 and gets incremented on the subsequent decision level by 1. Then, the algorithm makes further deductions based on that decision along with the current partial assignment. This deduction is called *unit propagation* or *Boolean constraint propagation* (BCP). During the deduction, the newly assigned variables (these variables are forced to be true or false) get the same decision level as the decision variable. After all the possible deductions at a decision level, if the problem becomes satisfiable, then the solver will return SATISFIABLE; if there exists at least one conflicting clause, the solver will call the procedure *analyze_conflict*(), which performs a conflict analysis and returns an appropriate decision level as the backtrack point; if *analyze_conflict*() returns level 0, then the problem is unsatisfiable and the solver returns UNSATISFIABLE. Otherwise, the solver backtracks to the directed level and undo the relevant variable assignments. During the conflict analysis, the solver may generate a learned clause and store it into the clause database. This type of learning is called *conflict driven learning*. In case that the solver cannot determine either a conflict or satisfiability after decision and subsequent deduction, the solver chooses another decision variable from the free pool of variables.

Since a decision variable may appear positively or negatively in a clause, to refer to such a literal, we may use the term *decision literal* in general.

For more details, the reader can consult [53].

```
status = preprocess()
if status == KNOWN then
  Return status
end if
while true do
  decide_next_branch()
  while true do
    status = deduce()
    if status == CONFLICT then
      blevel = analyze_conflict()
      if blevel == 0 then
        return UNSATISFIABLE
      else
        backtrack(blevel)
      end if
    else if status == SATISFIABLE then
      return SATISFIABLE
    else
      break
    end if
  end while
end while
```

Algorithm 2: The Iterative Version of DPLL Algorithm

2.2.2 Components of a DPLL SAT solver

For the last few decades, various components of DPLL based SAT solvers have been developed and are the subject of extensive scrutiny. In this section, we shall review the main components of a DPLL based SAT solver, which have evolved over time. After reading this section, the reader will understand the state of the art techniques in building a fast SAT solver.

Decision heuristics

In the function *decide_next_branch*() of Algorithm 2, a variable from the unassigned pool of variables is selected in order to assign it to a value. The variable selection has a huge effect on determining the size of the search tree. An appropriate variable selection can prune the search space dramatically for the same basic algorithm executing on a specific problem instance. So, to increase the efficiency of the SAT solver, several decision heuristics were proposed over the years.

Some of the earlier decision heuristics are Bohm's heuristic [8], Maximum Occurrences on Minimum Sized Clauses (MOM) [14] and Jeroslow-Wang heuristic[21]. These heuristics are used to branch on a variable, which will generate the maximum number of implications or satisfy most clauses. Thus, these types of heuristics are *greedy* in nature. While these heuristics are useful for random problems, they are not efficient for structured problems as they do not capture the relevant information.

Another decision heuristic proposed in the literature, decides a variable by the literal count of the variable on the problem instance in hand [31]. This literal count heuristic counts the number of unresolved clauses, where the variable under consideration appears in either phase. When combined with dynamic largest combined sum (DLIS), this heuristic yields quite good results for a number of benchmarks being tested. But, it has a downside. The clause counts are state-dependent as different variable assignment will give different clause counts. As a result, in every decision the counts for all the unassigned variables need to be recalculated.

As the SAT solvers are becoming more and more efficient, the time needed to decide on a branching variable by using this literal count scheme tended to take much of the solving time. The need of more efficient decision heuristics motivated more research. Another decision heuristic, called Variable State Independent Sum (VSIDS), is described in the SAT literature [35]. In VSIDS, a score for every literal is kept. The scores are initialized with number of occurrences of the corresponding literals in the initial problem. As the modern SAT solvers incorporate clause learning, the VSIDS will increase the score of a particular literal if any of the added clause contains that literal. In addition, the scores are periodically divided by a constant number to give more weight to the literal count of the literals which appear on the most recently added clauses. VSIDS selects the variable with the highest combined scores of both of its literals to branch. Experimental results show that VSIDS takes little portion of the total runtime of the solver, thus making the solver more efficient.

In another approach [18], the heuristic tries to decide on the variable which is active recently. Unlike VSIDS, where activity of a variable is determined by its occurrences, in this heuristic, activity of a variable is determined by its involvement in conflicts. In this scheme, when a conflict occurs all the literals in all the clauses involved in the conflict will have their scores increased. To capture the recentness attribute of the decision variable, like VSIDS, the scores of the literals are decayed periodically. For branching this decision heuristic considers only those variables, which appear in the last added unresolved clauses.

Deduction algorithm

After the decision variable is assigned a value, the solver attempts to find what are the other deductions possible with respect to the current variable assignments. The function *deduce*() in Algorithm 2 does these further deductions. After finding all the possible deductions with respect to the current variable assignment, the function *deduce*() may return three status values. It returns SATISFIABLE if all the clauses in the problem instance are satisfied by the current partial assignment. A status CONFLICT is returned by *deduce*(), if there exists a clause which has all the literals assigned to zero. Otherwise, it will return UNKNOWN and the solver will continue to branch.

Among many other deduction rules proposed over the years, the *unit clause rule* [28] is the most efficient one, in the sense that it requires little computational time, though it can prune large search space. The unit clause rule attempts to deduce a clause in which all but one of its literals are assigned to 0. Such a clause is called a *unit clause*. The unit clause rule assigns 1 to the unassigned literal, which is called a *unit literal*, making the unit clause true. In the rest of this thesis, such a literal will also be said to be *implied* or *forced*. As we have mentioned earlier, the process of assigning 1 to a unit literal is called *unit propagation* or *Boolean Constraint Propagation (BCP)*, which is the central part of DPLL based SAT solving, as the SAT solver will pass most of its time doing BCP.

Implementation of BCP

After a variable assignment, BCP attempts to detect unit clauses and conflicting clauses. Several approaches are described in the SAT literature to implement BCP, to be described below.

Counter based implementation In counter based approach [51], for each variable we maintain two lists of clauses where the variable appears in positive and negative phases. Additionally, each clause maintains two counters, one of those keeps the count of value 1 literals in that clause and the other counter keeps the count of value 0 literals in the same clause. After a variable assignment, all the counters of the clauses that contain that variable gets updated. This approach identifies unit clauses and conflicting clauses by checking the value of these literal counters of every clause. If the count value of the 0 value counter of a clause becomes equal to the clause's length, then the clause is a conflicting clause. If the

count value of the 0 value counter of a clause is one less than the length of the clause, and the value 1 literal counter is 0, then the clause is a unit clause.

Despite its simplicity, counter based approach to BCP is not the most efficient one. If any CNF problem instance has m clauses, n variables and on average each clause has l literals, then whenever a variable is assigned, we need to update lm/n counters. On the occasion of a conflict, we need to undo the change in the counters of the clauses relevant to the variables which are unassigned during backtracking. On average, again we have to undo lm/n counters for each un-assignment. Apart from this, in modern SAT solvers, the learning mechanism tends to add a large number of long clauses. That makes counter based BCP relatively slow.

Head/Tail mechanism A more efficient approach for implementing BCP is Head/Tail approach [51]. In this approach, all the literals of a clause are stored in an array. Every clause has two pointers, namely the head and tail pointer. Initially, the head and tail pointers point to the first and last literal of a clause. Each variable maintains four linked lists, which contain pointers to clauses. There is a head pointer list and a tail pointer list for both phases of a variable. Thus, a variable v maintains four linked lists. Let us name them as $clause\_of\_pos\_head(v)$, $clause\_of\_neg\_head(v)$, $clause\_of\_pos\_tail(v)$ and $clause\_of\_neg\_tail(v)$. Each of them contains pointers to clauses where v appears in both phases at the head and tail position, respectively. When a variable v is assigned 1, all the clauses pointed to by $clause\_of\_pos\_head(v)$ and $clause\_of\_pos\_tail(v)$ will be ignored and all the clauses C pointed to by the list $clause\_of\_neg\_head(v)$ get the focus. Notice that, the head literal of a clause C turns out to be the negative literal of C. The head/tail approach starts searching from the head position to the tail position in C and attempts to find a literal in C which is not yet assigned to 1. The search process may encounter one of the four cases:

1. During the search, if we find a literal which is assigned to value 1, then the clause is satisfied and we do not need to do anything.

2. If we find a literal l other than the tail literal, which is unassagined, then C is removed from $clause\_of\_pos\_head(v)$ and we add C to the head list of the variable corresponding to the literal l. This operation is called *moving the head literal*.

3. If all the literals between those two pointers are assigned to the value 0 and the tail literal is an unassigned literal, then C is a unit clause with l being its unit clause.

4. If all the literals between and including the tail are assigned to the value 0, then C is a conflicting clause.

For $clause\_of\_pos\_tail(v)$, similar actions as 1 to 4 are performed starting from tail to head.

In the head/tail approach, when a variable is assigned value 1, the clauses that contain positive literals of this variable are ignored and vice versa. As compared to the counter based approach to BCP, on each variable assignment, on average m/n clauses need to be updated in the head/tail approach. In general this property makes the head/tail approach much faster than the counter based approach.

Though faster than counter based approach, during unassignment, the computational complexity of the head/tail approach, due to unassignments during backtracking, remains the same as in the counter based approach. This promoted research for faster BCP algorithms. Currently, the most efficient implementation is called *2-watched literal*.

The 2-watched literal scheme The 2-watched literal scheme [35] is similar to the head/tail approach of BCP. In this scheme we employ two pointers in every clause, but unlike the head/tail literal scheme, there is no imposed order on the position of the watched literals. Every variable has two lists, one of which contains pointers to the clauses where the variable appears positively and the other contains pointers to the clauses where the variable

appears negatively. Let us denote these two lists by $pos\_watched(v)$ and $neg\_watched(v)$. In contrast to the head/tail approach, the watched literals can move to any direction. Initially, both of watched literals are free. When a variable v is assigned to value 1, the clauses pointed to by the list $pos\_watched(v)$ are ignored. In this case, the scheme focuses only on the clauses pointed to by the list $neg\_watched(v)$ list. In every clause C, pointed by the list $neg\_watched(v)$, the 2-watched scheme will search for a literal l which is not set to value 0. During the search one of the four cases may occur:

1. If we find a literal which is assigned to value 1 and which is the other watched literal, then we do not need to do anything.

2. If we find a literal such as l, which is not the other watched literal, then C is removed from $neg\_watched(v)$ and we add pointer to C to the list of pointers corresponding to the literal l. This operation is called *moving the watched literal*.

3. If the other watched literal is the only such l, then the clause C is a unit clause and the other watched literal becomes a unit literal.

4. If all the literals in C are assigned to value 0 and no such literal as l exists, then C is a conflicting clause.

Like the head/tail approach, during the identification of unit and conflicting clauses, the 2-watched literal scheme has the same advantage of reducing the number of clauses to search for. In addition to that, we need less work while undoing variable assignments during backtracking. Backtracking in 2-watched literal is done in constant time. As the watched literals are the last literals to be assigned to value 0, backtracking guarantees that they will be unassigned and will become free again. Thus no action is needed to update the pointers for the watched literals. This makes 2-watched literal scheme significantly faster than the other two approaches described above. The 2-watched literal scheme has become the state of the art BCP mechanism for DPLL based SAT solvers.

Conflict analysis

In unit propagation, if the solver detects one or more conflicting clause, then it indicates that a conflict has occurred and the current search space cannot lead to a solution. It needs to backtrack to a certain point and enter into a new search space to continue. For this purpose, it needs to analyze the conflict to get the backtracking point. This conflict analysis is done inside the $analyze\_conflict()$ function of Algorithm 2. The basic DPLL solver employs a simple conflict analysis technique. After a conflict occurs, the basic DPLL SAT solver searches to find a decision variable v_d with the highest decision level which has not been assigned to its both phases yet (called an *unflipped variable*.[2]) Then, the DPLL solver unassigns all the assignments done previously between the current decision level and the decision level of the variable v_d. This method is said to perform *chronological backtracking*.

Though chronological backtracking works fine for some random problems, for structured problems generated from real world applications, chronological backtracking is not efficient. Because it always backtracks to the decision level of the last unflipped variable, it may not backtrack to the real reason of the conflict. A more efficient conflict analysis method would backtrack to a point which really has caused the conflict. This method may backtrack to any of the earlier decision levels. This method is called *non-chronological backtracking*. In addition, the state of the art conflict analyzer records the information about the current conflict as a learned clause, which is used to prune search space in the future. The addition of new clauses does not change the satisfiability of the original problem, because after the addition of the clauses, the extended problem remains equivalent to the original problem. This type of mechanism is called *conflict directed learning* [33]. The learned or recorded

[2]During the search, when a variable is tried with its both values, then we say that the variable is *flipped*.

clauses are called *conflict clauses*. When a unit literal is implied by a unit clause, that clause is called the *antecedent clause* of the corresponding variable. In conflict driven analysis, a learned clause is generated by the process of resolution that involves the conflicting clause and antecedent clauses of some other conflicting variables.

The pseudocode for conflict analysis is depicted in Algorithm 3. During unit propagation whenever a conflict clause is detected, the $analyze\_conflict()$ function is called to resolve the conflict. The function $choose\_literal(cl)$ chooses the literal from cl which was assigned last in cl. The function $resolve(cl, ante, var)$ returns a clause after executing resolution process on cl and $ante$, which contains all the literals in cl and $ante$ except the literals corresponding to var. The resulting clause is also a conflicting clause, because the inputs to $resolve()$ consist of a conflicting clause and a unit clause, respectively.

$cl = find\_conflicting\_clause()$
while $!stop\_criterion\_met(cl)$ **do**
  $lit = choose\_literal(cl)$
  $var = variable\_of\_literal(lit)$
  $ante = antecedent(var)$
  $cl = resolve(cl, ante, var)$
end while
$add\_clause\_to\_database(cl)$
$back\_dl = clause\_asserting\_level(cl)$
$return\ back\_dl$

Algorithm 3: Conflict Analysis and Clause Learning

Inside the while loop of Algorithm 3, clause generation continues until the stop criterion is met. In the state of the art SAT solvers, the stop criterion is met when the generated clause is an *asserting clause* - a clause which has all of its literals assigned to 0 and exactly one of its literal l_c, is assigned in the current decision level. The decision level of the literal l_a, with the second highest decision level in the asserting clause, is the *asserting level*. This asserting level is the backtracking level to which the search backtracks. After backtracking at the asserting level, the asserting clause becomes a unit clause as all the literals in the asserting clause except l_c are assigned to 0.

The FirstUIP scheme [52] is the most efficient heuristic for selecting an asserting clause from a bunch of candidate asserting clauses. In firstUIP, the stopping criterion of the while loop in Algorithm 3 is met, as soon as the first asserting clause is found.

Clause learning is very useful in pruning search space in the future. But some of them are redundant and not all of them are equally useful. Keeping all the learned clauses in the clause database may slow down the search process and may occupy unnecessary memory space. Therefore, it is often required to delete some of the learned clauses which are less useful and have too many literals. This operation is referred to as *forgetting a clause*.

Chapter 3

Constraint Programming

Constraint Programming (CP) is a programming paradigm, where relations between variables are stated in the form of constraints.

Informally, a constraint on a sequence of variables is a relation on their domain values. To solve a problem in CP, one way is to formulate it as a *Constraint Satisfaction Problem (CSP)*. A CSP[1] consists of three parts: a finite set of variables, a finite domain of values associated with each of the variables and a set of constraints restricting the values that variables can simultaneously take. A solution to a CSP is an assignment to all of the variables from their domains such that all the constraints are satisfied. If a CSP instance has a solution then it is said to be consistent, and otherwise inconsistent.

Formally, an instance of CSP can be defined as a triple $P(X, D, C)$, where C is a finite set of constraints $\{c_1, c_2, \ldots c_t\}$ over a finite set of variables $X = \{x_1, x_2, \ldots x_n\}$ and a domain $D = \{D_{x_1}, D_{x_2}, \ldots D_{x_n}\}$ that maps each variable $x_i \in X$ to a finite set of values. A constraint c_i is a relation R_i defined on a subset of variables $S_i \subseteq X$, where S_i is called the *scope* of c_i. If $S_i = \{x_{i_1}, x_{i_2}, \ldots x_{i_n}\}$, then R_i is a subset of the Cartesian product $D_{x_{i_1}} \times \cdots \times D_{x_{i_m}}$. See [13] for more details.

3.1 CP in a Nutshell

Most common and widely used approach to solving a CSP is based on backtracking depth-first search, employing some constraint propagation based on consistency techniques. Constraint propagation attempts to reduce the size of the domain for each CSP variable - it removes a value from the domain of a variable if the value is inconsistent with the current assignment of the other variables with respect to the set of constraints. This means that the current assignment will not lead to a solution. In general, constraint propagation may not solve the given problem instance [36].

In CP, variables, their values and the constraints are stored in an object called *constraint store*. A constraint store can have one of the three possible states:

- Solved: Each variable has exactly one value to choose from.
- Failed: At least one variable has no value left in its domain from which it can take a value.
- Distributable: Neither of the above.

Thus, the process of constraint propagation reduces the ranges of values of the variables. Figure 3.1 depicts a constraint store, where the constraint propagation reduces the domain of the CSP variables X and Y to {3,4} and {2,3} respectively. After constraint propagation this constraint store become a distributable store.

[1] In this thesis, we also use CSP to refer to an instance of CSP.

Variables	Values	Constraints
X Y	{1,2,3,4} {2,3,4,5}	Y<X

Figure 3.1: An example of a constraint store.

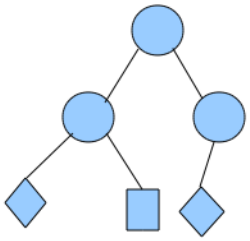

Figure 3.2: Sample CSP search tree.

To get a solution, a distributable store needs to be reduced further. Branching is used for that purpose. The process of branching removes the alternative solutions that exist in the distributable store. Branching takes a copy of the store and adds a constraint β to it. It also takes another copy of the constraint store and adds the constraint $\neg\beta$. For the distributable constraint store, obtained from the constraint store depicted in Figure 3.1, taking β as $X = 3$ would reduce the distributable store to the solved store. Branching forms a binary tree, where each store is a node in the tree. Every leaf node in the tree is either a failed or solved store, while the body of the tree consists of distributable stores.

To find a solution for a CSP, we need to explore the search tree. A search can be explored in different manners, such as depth first search or breadth first search. Usually, constraint propagation takes significant amount of time to be completed and search tree is not constructed until constraint propagation is finished. Search tree is lazily constructed, while the search engine is exploring the tree. Each node is constructed on demand. Figure 3.2 shows an example of a search tree where the circle denotes a distributable store, rectangle denotes a failed store and the diamond denotes a solved store.

3.2 Types of Constraints in the CP Paradigm

In this section, we will describe the typical types of constraints provided by the CP paradigm [7]:

- *Numerical Constraints*: For most applications numerical constraints are linear constraints that involve primitive arithmetic operations of variables and constraints. For example, $3x \leq 5y$ is a numerical constraint, where x and y are variables with numerical domains.

- *Symbolic Constraints*: High level constraints. Example of symbolic constraints are constraints defined on lists or data structures. For instance, $y[i] = x$ is a symbolic constraint, where i is an integer-valued unknown variable, y is an array of unknown variables and x is another variable.

- *Meta Constraints*: An arbitrary Boolean combination of constraints. Along with conjunction of constraints, disjunction and implication between constraints are also supported in the CP paradigm. For example, $x + y = 10 \vee x - y = 5$ is a meta constraint.

- *Global Constraints*: The use of global constraints in CP can greatly increase the efficiency of the constraint solver. The idea of global constraints can be expressed like this: a constraint c is named as global, when processing c as a whole gives better results than processing any conjunction of constraints that is semantically equivalent to c [3]. From syntactical point of view, a global constraint is a complex constraint that captures a relation between a non-fixed number of variables. In the following we will describe some frequently used global constraints and their implementations on a constraint solver named GECODE.

  – *All-different*: *allDiff*$(x_1, x_2 \dots x_n)$ assigns distinct values to the variables $x_1, x_2 \dots x_n$, that is, this global constraint stands for a conjunction of dis-equality constraints

  $$\bigwedge_i \bigwedge_{j>i} x_i \neq x_j$$

  The GECODE implementation of the propagator of *allDiff* is called *distinct* [45]. In GECODE, by posting

  $$distinct(home, x)$$

  we constrain that all the variables in array x to be pairwise distinct, where *home* is the pointer to the constraint store to which x belongs.

  – *Cumulative*: Cumulative is a scheduling constraint. It considers a set T of tasks and a limit L. Each task is composed of four components: origin, duration, end and height (the portion of a resource that the task uses at any given time point t). The cumulative constraint constrains that at each point in time, the cumulated height of the set of tasks that overlap that point, does not exceed the give limit L of the resources they are currently using. A task overlaps a point t if and only if (1) its origin is less than or equal to t, and (2) its end is strictly greater than t. The cumulative constraint also imposes that for each task, the constraint $origin + duration = end$ holds [2]. The cumulative global constraint is used to schedule tasks which require limited amount of resources with limited capacity.

  GECODE implements two versions of cumulative [45]. The first models a resource with a limited capacity that each task can use. Each task requires certain resource usage. The constraint is, at each time the total resources usage of the tasks must not exceed the capacity of the resource. This version of cumulative global constraint has the following signature:

  $$cumulative(home, c, s, d, u)$$

  where c is the capacity of the resource, s, d and u are the variable arrays which contain starting time, duration and the required resource usage by each of the tasks respectively.

  The second version models multiple resources that can be shared by a set of tasks. The syntax for this version cumulative propagator function is

  $$cumulative(home, resource, start, duration, end, height, limit, atmost).$$

  This propagator function posts a constraint over a set of tasks T, where each task T_i is defined by

  $$\langle resource_i, start_i, duration_i, end_i, height_i \rangle.$$

The $resource_i$ component specifies the potential resources that T_i can use, $start_i$, $duration_i$ and end_i indicates when T_i can occur and $height_i$ indicates the amount of resources that T_i can use. The resource R_i is defined by the limit $limit_i$, which specifies the limit or capacity of the resource R_i, and the Boolean parameter *atmost* indicates whether the limit specified by the limit parameters are maximum (*atmost* is true) or minimum (*atmost* is false) limits. The parameter *atmost* applies to all the resources.

– *Sum*: The Sum constraint enforces the sum of the product of collection of variables and their coefficient to conform to a constant number with respect to a binary relationship. The signature of the propagator function of sum is

$$sum(c, x, rel, s)$$

where c is the coefficients of the variables in the variable array x. The sum of the product of each c_i and x_i should conform to the constant s with respect to the binary relation rel [2].

In GECODE , the name of the sum propagator function is *linear* [45]. The propagator function

$$linear(home, a, x, IRT\_EQ, c)$$

posts the linear constraint $\sum_{i=0}^{|x|-1} a_i x_i = c$. If all the coefficients are 1, then a can be omitted from the propagator function. For instance,

$$linear(home, x, IRT\_GR, c)$$

posts the linear constraint $\sum_{i=0}^{|x|-1} x_i > c$.

3.3 State of the Art CP Solver

In this section we shall review the main components of a state of the art CP solver.

3.3.1 Variable and value ordering heuristics

During constraint propagation and branching, we need to select some variables and order the values of those variables. The *fail first principle* is the rule, that guides variable and value selection process. The essence of the *fail first principle* is, to succeed, first things to try are the things which are likely to lead to failure.

The most common variable selection heuristics are [7]:

- MINDOM: Selects the variable with the smallest domain.
- MAXDEG: Selects the variable connected to the largest number of constraints.
- DOMDEG: Favors variables with small domains and large degrees.[2]
- BRELAZ: MINDOM principle that breaks ties by returning the first variable connected to the largest number of unassigned variable in the corresponding constraint graph.
- LEX: User defined variable ordering.

The most common value ordering heuristics are [7]:

[2]The degree of a CSP variable x is the count of constraints, which have x in their scope.

- LEX: Lexicographical ordering of values.
- INVLEX: Reverse lexicographical ordering of values.
- MIDDLE: Selects median value of the domain first.
- RANDOM: Selects a value randomly.

All the value ordering heuristics mentioned above except RANDOM, are interpreted with respect to the problem at hand. In addition, in a typical solver constructs are provided so that a CP programmer can define his/her own heuristics as well.

3.3.2 Constraint propagation

Constraint propagation is performed by a process called *consistency checking*. The general notion of consistency is called *k-consistency*, which enforces that a partial solution with an assignment of $k-1$ variables be consistently extended to a partial solution with respect to the constraints at hand. If $k = 1$, it is called *node-consistency*, if $k = 2$, it is called *arc-consistency* and if $k = 3$, it is called *path-consistency*.

Formally, arc-consistency can be defined as follows:

Given a CSP $P(X, D, C)$, a constraint $c \in C$ over $\{x, y\}$, where $x, y \in X$, x is arc-consistent with respect to y over c if and only if for every assignment $x \rightarrow a \in D_x$, there is a corresponding assignment $y \rightarrow b \in D_y$, such that $x \rightarrow a$ and $y \rightarrow b$ satisfy c. A binary constraint whose arity is $\{x, y\}$ is arc-consistent if x is arc-consistent with respect to y and y is arc-consistent with respect to x. A CSP is arc-consistent, if all of its constraints are arc-consistent [13].

Example
Let $P(X, D, C)$ is a CSP, where $X = \{x, y, z\}$, $D = \{D_x, D_y, D_z\}$, $D_x = D_y = \{1, 2, 3\}$, $D_z = \{2, 3, 4\}$ and $C = \{x < y, y < z, z \leq 3\}$. By enforcing node-consistency on z the domain $D_z = \{2, 3, 4\}$ is reduced to $D'_z = \{2, 3\}$. By enforcing arc-consistency on x w.r.t. the constraint $x < y$, $D_x = \{1, 2, 3\}$ is reduced to $D'_x = \{1, 2\}$, as for $x = 3$ there is value of y such that $x < y$ is satisfied. Similarly, D_y is reduced to $D'_y = \{2, 3\}$. Considering the constraint $y < z$, D'_y is further reduced to $\{2\}$, as a result, due to the first constraint, D'_x becomes $\{1\}$ and D'_z is reduced to $\{3\}$. As the domain of every variable becomes singleton, no farther reduction is possible and the given CSP is solved [20].

Arc-consistency for binary and global constraints

Now we shall review arc-consistency algorithms developed for binary and global constraints.

Arc-consistency for binary constraints The state of the art constraint propagation engines are based on the arc-consistency algorithm called *AC-3*, developed by Waltz and Mackworth [7]. The idea of AC-3 [13] is to reason locally by considering each constraint in turn and reduces the domains of the variables of its scope, if needed. A queue is maintained, which contains the variables that have recently been modified. This type of implementation is called *variable based implementation*. In the *constraint based implementation*, the constraints that depend on the variables are modified.

Algorithm 4 is a variable based implementation of AC-3. The algorithm accepts a CSP problem as input and produces a consistent CSP problem equivalent to the input problem. The algorithm uses a function, named $revise()$ [13], whose pseudocde is described in Algorithm 5. The function $revise()$ takes a pair of variables (x_i, x_j) in a certain constraint c_{ij} and their domain as input and outputs the domains of x_i such that x_i is arc-consistent with x_j.

The AC-3 algorithm maintains a queue. In that queue, initially, all the variables which are within the scope of every constraint are stored pairwise. In each iteration of the while

for *each* $c_{ij} \in C$ **do**
  for every pair (x_i, x_j) with x_i and x_j in the scope of c_{ij} **do**
    $queue \leftarrow queue \cup \{(x_i, x_j), (x_j, x_i)\}$
  end for
end for
while $queue \neq \phi$ **do**
  select and delete (x_i, x_j) from queue
  $revise(x_i, x_j)$
  if $revise(x_i, x_j)$ causes a change in D_{x_i} **then**
    $queue \leftarrow queue \cup \{(x_k, x_i), k \neq i, k \neq j\}$
  end if
end while

Algorithm 4: AC-3

for *each* $a_i \in D_{x_i}$ **do**
  if there is no value $a_j \in D_{x_j}$ such that $(a_i, a_j) \in c_{ij}$ **then**
    remove a_i from D_{x_i}
  end if
end for

Algorithm 5: $revise(x_i, x_j)$

loop in Algorithm 4, a pair (x_i, x_j) is removed from the queue. Then it calls the procedure $revise(x_i, x_j)$, which tests every value a in $D_{x_i}$to see if there is a support value in D_{x_j}. If no support is found in D_{x_j}, then a is removed from D_{x_i}. Thus, $revise(x_i, x_j)$ removes the values from the domain of x_i which are inconsistent with the domain of x_j. After returning from $revise(x_i, x_j)$, if revise causes any removal from D_{x_i} then all the related pairs to x_i are added to the queue. The while loop continues execution until the queue becomes empty.

The time complexity of this AC-3 algorithm is $O(ek^3)$, where k bounds the domain size and e is the number of constraints. The optimal time complexity for checking arc-consistency for binary constraints is $O(ek^2)$. AC-4 is the algorithm that achieves this optimal time complexity, which is given in Algorithm 6 [13]. Like AC-3, AC-4 takes a CSP $P(X, D, C)$ and outputs a consistent CSP equivalent to P.

initialize $S(x_i, a_i)$, $counter(x_i, a_i, x_j)$ from all $c_{ij} \in C$
for all *counters* **do**
  if $counter(x_i, a_i, x_j) = 0$ $(if\ \ (x_i, a_i)\ \ is\ \ unsupported\ \ by\ \ x_j)$ **then**
    add (x_i, a_i) to χ
  end if
end for
while χ is not empty **do**
  Choose and remove (x_i, a_i) from χ, remove a_i from D_{x_i}
  for each $(x_j, a_j) \in S(x_i, a_i)$ **do**
    decrement $counter(x_j, a_j, x_i)$
    if $counter(x_j, a_j, x_i) = 0$ **then**
      add (x_j, a_j) to χ
    end if
  end for
end while

Algorithm 6: AC-4

Two variables x and y are neighbors to each other, if x and y are in the same scope of a specific constraint. The AC-4 algorithm associates each assignment (x_i, a_i) with the amount of support from the neighboring variables x_j. The value a_i is removed from the domain of

x_i, if it has no support from some of the neighboring variables. AC-4 uses a counter array, $counter(x_i, a_i, x_j)$, to store the number of support values the assignment (x_i, a_i) has from its neighboring variables. An array $S(x_j, a_j)$ is used to store all the assignments to other variables that (x_j, a_j) supports. χ is used to store all the unsupported values. In each step of while loop of Algorithm 6 an unsupported assignment is picked up from χ, the respective domain is reduced by removing the nonsupporting values and all the affected counters are updated. As a result of the updates, any value with counter being 0 is placed into χ. The process continues, until χ becomes empty.

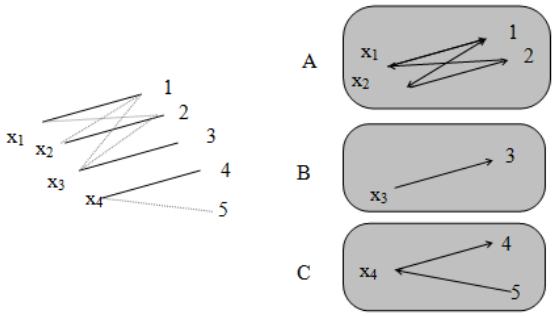

Figure 3.3: Regin's domain propagation algorithm for *allDiff* constraint.

Arc-consistency for global constraints

The CP literature has provided a wide range of constraint propagation algorithms for global constraints. Most of the algorithms are based on graph theory or operations research. In 1994, Regin proposed an algorithm [40] for domain propagation for the $allDiff$ constraint.

As mentioned earlier, the global constraint $allDiff$ assigns all different values to the variables it contains. To understand how Regin's algorithm works, let us consider the constraint $allDiff(x_1, x_2, x_3, x_4)$, where x_1 and x_2 have the same domain $\{1, 2\}$, the domain of x_3 is $\{1, 2, 3\}$, and the domain of x_4 is $\{4, 5\}$ [7]. The domains of x_1 and x_3 are both arc-consistent as long as $x_1 \neq x_3$. And the domain of x_3 is arc-consistent with x_2 and x_4 as well, when $x_2 \neq x_3$ and $x_3 \neq x_4$ are considered separately. But any solution in conjunction of all the dis-equalities excludes values 1 and 2 from the domain of x_3. Regin's algorithm can exactly compute the arc-consistent domains of the variables, taking all the CSP variables x_1, x_2, x_3 and x_4 into consideration.

In Figure 3.3, we depict the execution of Regin's algorithm. The algorithm maintains a value graph of constraints, which is essentially a bipartite graph, where each value and variable is represented by a vertex and a variable x connects to a value a, if and only if a is in its domain. A solution to the $allDiff$ constraint contains matchings between variables and exactly one of their values. At first, an initial matching is computed [19], shown by the solid line (Figure 3.3, left). The goal is to remove all the edges that does not participate to any of the matching (Figure 3.3, right). The edges that should be preserved should have one of the following properties:

- Belongs to original matching (region B).
- An alternative path/cycle of even length exists whose edges are alternatively chosen inside and outside the original matching (regions A,C).

Two edges $x_3 : 1$ and $x_3 : 2$ are removed from the graph as they are not original matching of the graphs, nor they are part of any even alternative path.

3.3.3 Conflict analysis in CP

Conflict analysis in CP is performed by a technique, named *conflict directed backjumping (CBJ)* [39]. When a conflict arises, CBJ analyzes the conflict, and finds a backjumping point which is the real reason of the conflict. The real reason is the deepest level in the decision tree which participates in the cause of the conflict. This type of intelligent backjumping helps a solver to avoid detecting the same inconsistencies multiple times. We shall describe the underlying technique of the CBJ procedure by considering a CSP problem as given [7]. Let

$$(c1) \quad x_4 \neq x_5$$
$$(c2) \quad x_2 + x_3 + x_5 \geq 2x_1$$
$$(c3) \quad x_1 + x_4 = x_5$$

If we assign x_1 to 0, then it generates a conflict, as $x_4 \neq x_5$ contradicts with $0 + x_4 = x_5$. In the propagation based solver, this contradiction is not detected immediately as it considers each of the constraints separately. Typically, a solver will not detect the inconsistencies until x_4 and x_5 are instantiated. For example, assigning x_4 to 0, the solver will deduce that x_5 should be assigned to value 0 (by c3), which leads to a contradiction (by c1).

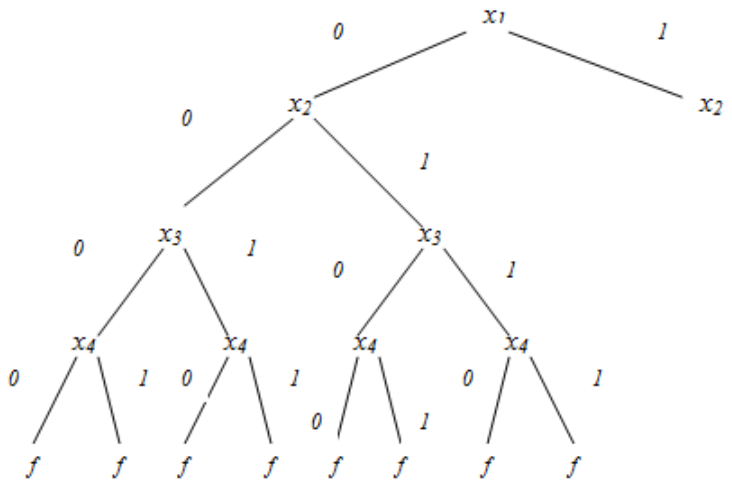

Figure 3.4: Part of the search tree for the example CSP.

Figure 3.4 depicts a search tree for the example CSP considered here. From this figure, we can understand the reason why a solver may repeatedly branch between the areas where the main reason of the conflict does not reside.

In our example, while inspecting the branches, the solver detects a conflict, after successively assigning values 0 and 1 to x_4. Then the successive assignment of x_4 to the values 0 and 1 are repeated in every corresponding branch of x_2 and x_3, though the real reason of the inconsistency is actually the choice $x_1 = 0$. Failure to identify the real reason of the failure makes a solver engaged in unnecessary backtracking.

The constraint that is violated in this case are $x_4 \neq x_5$ and $x_1 + x_4 = x_5$, which do not involve x_2 and x_3. So, instead of $x_2 = 0$ or $x_3 = 0$, the choice $x_1 = 0$ should be reconsidered. The CBJ technique maintains a *conflict set* at each node of the search tree, which contains the variables involved in the deductions performed by the propagation engine to reach that node. This conflict set helps to find the reason of the conflict.

When a failure is detected (in our example, the first is the left most node x_4 of Figure 3.5), the CBJ analysis engine evaluates the union of all the conflict sets of both branches of

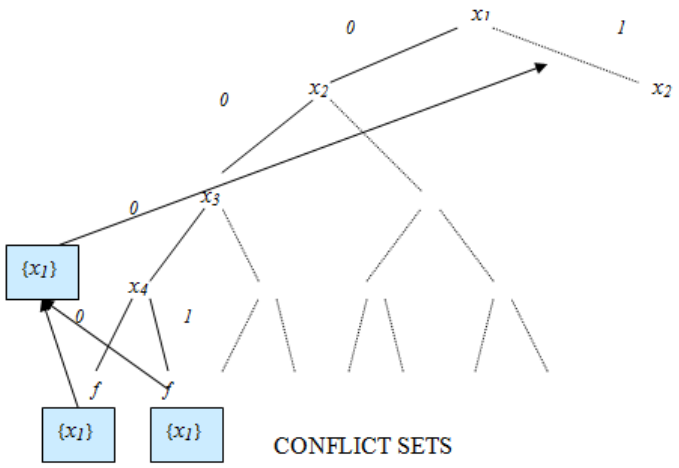

Figure 3.5: Conflict directed backjumping.

the conflicting node. In Figure 3.5, the union results in $\{x_1\}$. Then the solver backjumps to the deepest variable in the conflict set. In our example, as shown in Figure 3.5, the solver backjumps to x_1 directly.

Chapter 4

SAT Modulo Theory

For many applications, encoding a given problem as a SAT instance may not be a wise choice. A better choice may be to express parts of such a problem in a background theory. This is known as Satisfiability Modulo Theories (SMT). Given a formula F, the SMT problem for a theory T determines whether F is T-satisfiable, that is whether there exists a model of T that is also a model of F [37]. Such a formula is called a T-formula. For example, the problems arising from program verification usually involve arrays, lists, and other high level data structures. Therefore, naturally this type of problems are considered as satisfiability problems modulo the combined theory T of these data structures. In such applications, a problem instance consists of propositional literals as well as atoms over the combined theories. A formula in this context may look like the following

$$p \vee \neg q \vee a = f(b-c) \vee read(s, f(b-c)) = d \vee a - g(c) \leq 7$$

Formally, a theory T is a set of closed first order formulas. A formula F is T-satisfiable or T-consistent if $F \wedge T$ is satisfiable in the first order sense. Otherwise, it is called T-unsatisfiable or T-inconsistent.

The approaches for solving the SMT formulas can be broadly classified as *lazy* and *eager* ones [46]. In the following we will describe both of them.

4.1 Lazy SMT Approach

A DPLL based lazy SMT solver works by integrating a DPLL based SAT solver and a theory solver. For convenience, let us denote such a combined framework by DPLL(T) and call the corresponding SMT solver a DPLL(T) solver. The DPLL(T) solver takes a T-formula ϕ as an input and builds its Boolean abstraction $\phi^p = T2B(\phi)$, where $T2B(\phi)$ encodes the theory literals in ϕ into propositional variables (here $T2B$ stands for Theory to Boolean) [46].

The Boolean formula ϕ^p is given to an enumerator, which enumerates a complete collection of truth assignments $\{\mu_1^p, \ldots, \mu_n^p\}$ for ϕ^p. Whenever a new μ^p is generated its corresponding theory literals, denoted $\mu = B2T(\mu^p)$, are fed to a T-solver. If the T-solver finds it T-satisfiable, then DPLL(T) returns SAT. If it does not return SAT, then the enumerator again produces an assignment. The process is repeated until a satisfying assignment is found, or the enumerator generates no more μ^p. Figure 4.1 depicts the architecture of a typical lazy SMT solver.

In lazy SMT approach, there are two ways to integrate a SAT solver and a T-Solver. They are *offline approach* and *online approach*. In offline approach, for a formula ϕ^p, a complete collection of literal assignments μ^p (which satisfies ϕ^p), is generated by the enumerator. That μ^p is fed to a T-solver for determining the T-satisfiability of μ^p. But in online approach, the SAT solver embedded on DPLL(T) reasons and updates ϕ^p and μ^p incrementally. The reader can consult [46] for more details.

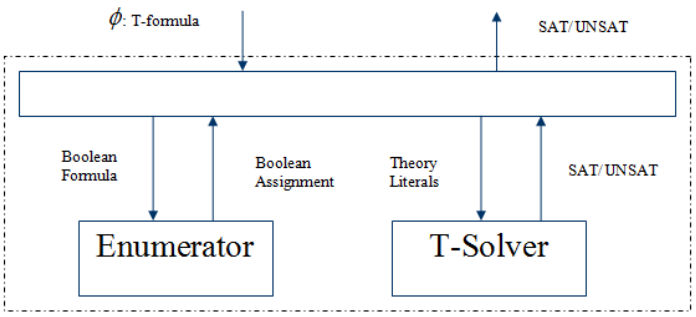

Figure 4.1: Lazy SMT approach.

Example Now we will use an example to show how lazy SMT works on online integration approach [46]. Consider a linear arithmetic formula ϕ:

$$
\begin{array}{ll}
(c1) & \{\neg(2x_2 - x_3 > 2) \vee A_1\} \\
(c2) & \{\neg A_2 \vee (x_1 - x_5 \leq 1)\} \\
(c3) & \{(3x_1 - 2x_2 \leq 3) \vee A_2\} \\
(c4) & \{\neg(2x_3 + x_4 \geq 5) \vee \neg(3x_1 - x_3 \leq 6) \vee \neg A_1\} \\
(c5) & \{A_1 \vee (3x_1 - 2x_2 \leq 3)\} \\
(c6) & \{(x_2 - x_4 \leq 6) \vee (x_5 = 5 - 3x_4) \vee \neg A_1\} \\
(c7) & \{A_1 \vee (x_3 = 3x_5 + 4) \vee A_2\}
\end{array}
$$

In this example, A_1 and A_2 are Boolean variables. The mapping $T2B(\phi)$ produces the Boolean abstraction of ϕ, denoted by ϕ^p, which can be expressed as follows:

$$
\begin{array}{ll}
(c1) & \{\neg B_1 \vee A_1\} \\
(c2) & \{\neg A_2 \vee B_2\} \\
(c3) & \{B_3 \vee A_2\} \\
(c4) & \{\neg B_4 \vee \neg B_5 \vee \neg A_1\} \\
(c5) & \{A_1 \vee B_3\} \\
(c6) & \{B_6 \vee B_7 \vee \neg A_1\} \\
(c7) & \{A_1 \vee B_8 \vee A_2\}
\end{array}
$$

The DPLL(T) solver accepts ϕ^p and starts execution. Suppose the DPLL(T) solver decides on the theory literal in order $\neg B_5, B_8, B_6, \neg B_1$ in c_4, c_7, c_6 and c_1. At this stage, it cannot unit-propagate any literal. So, the deduce engine of DPLL(T) (T-deduce) invokes the T-solver on $B2T(\{\neg B_5, B_8, B_6, \neg B_1\})$, which is

$$\{\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2)\}.$$

The T-solver returns SAT and deduces $\neg(3x_1 - 2x_2 \leq 3)$ as a consequence of $\neg(3x_1 - x_3 \leq 6)$ and $\neg(2x_2 - x_3 > 2)$. (T-propagation) of the first and last literal. $\neg B_3$, the corresponding Boolean literal of $\neg(3x_1 - 2x_2 \leq 3)$ is added to μ^p and then propagated. As a result, A_1, A_2 and B_2 are unit-propagated from c_5, c_3 and c_2. At this stage, as no more deductions are possible, T-deduce again invokes the T-Solver on $B2T(\{\neg B_5, B_8, B_6, \neg B_1, \neg B_3, A_1, A_2, B_2\})$, which consists of

$$\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2),$$
$$\neg(3x_1 - 2x_2 \leq 3), (x_1 - x_5 \leq 1)$$

But this time it finds inconsistency because of the first, second and eighth literals in the above set. The T-solver returns UNSAT. The deduction machine of the DPLL(T) solver returns a conflict. The conflict analyzer of DPLL(T) learns

$$(c8) \quad B_5 \vee \neg B_8 \vee \neg B_2$$

as a conflict clause (T-learning) and the solver backtracks (T-backjumping). Backtracking pops all the literals up to $\{\neg B_5, B_8\}$ and unit-propagate $\neg B_2$ on (c_8). Then starting from $\{\neg B_5, B_8, \neg B2\}$, also A_2 and B_3 are unit-propagated on $c2$ and $c3$ respectively. So, the μ^p and ϕ^p are updated and the search proceeds from there. The process repeats itself in this way by updating μ^p and ϕ^p until μ^p satisfies ϕ^p and also becomes T-satisfiable.

4.2 Eager SMT Approach

The eager approach reduces the T-satisfiability to SAT [37]. The input T-formula is translated into an equivalent Boolean formula which is given to a SAT solver to check its satisfiability. Two major families of encodings are worth mentioning:

1. Small Domain Encoding: For each variable v that belongs to a theory, appropriate range of values for v is found. After that, each variable is encoded into a vector of $\lceil \log 2(|v|) \rceil$ Boolean variables. Using binary arithmetic the original formula is transformed into a Boolean formula.

2. Per-Constraint Encoding: In this encoding scheme, for every atom ψ in the input formula ϕ, a Boolean variable A_ψ is introduced. Then, ϕ is encoded into a Boolean formula $\phi^p \vee \phi^T$, where ϕ^p belongs to $T2B(\phi)$ and ϕ^T is the Boolean encoding of the constraints that correspond to the T-atoms in ϕ^p.

4.3 Issues in Lazy and Eager Approaches to SMT

Some issues are crucial for both lazy and eager integration approaches to SMT [37, 46]. The efficiency of the lazy approach depends on the different features of both DPLL solver and T-solver that it integrates. The eager approach generally suffers from the blow up of the encoding size in propositional logic. In this section, we will briefly discuss these issues.

4.3.1 Issues in lazy approach

The applicability and usefulness of the lazy SMT solver depend on the following factors:

1. Some techniques used in the SMT approach require specific features of the T-solver. For example, T-backjumping and T-Learning described in the previous section require the capability of the T-solver to generate good enough conflict sets. The benefits of T-propagation depend on the deduction capabilities and efficiency of the T-solver attached.

2. The effect of integrating different techniques are not mutually independent. The combination of two techniques may not always produce positive effect. For example, pure literal filtering of DPLL can reduce pruning power of early pruning of T-solvers, by dropping some literals which may result in T-inconsistencies.

3. The benefits of many integration techniques depend on the addressed theory. In other words, the benefits depend on the trade off between the cost of T-solving and T-propagation and the amount of pruning brought to the Boolean search space by those techniques. For example, for a theory under difference logic, the cost of T-solving is relatively cheap. But for theories like linear arithmetic or bit vector T-solving may be very expensive. In the case of latter theories, the calls to the T-solver for Boolean search space reduction are not always beneficial.

4.3.2 Issue with the eager approach

In the eager approach, once the encoding of the T-formula is done, we can use any DPLL SAT solver for solving the T-satisfiability problem. Though implementation of the eager approach is easy, it often suffers from blow up in propositional encoding of the respective T-formula. For theories like difference logic and linear arithmetic, the blow up problem is more prominent. As shown in [12], an eager SMT solver UCLID is completely outperformed by DPLL-based lazy tools. Apart from this, no eager SMT solver took part in the recently held SMT competitions [1]. The SMT community now assumes that, in terms of efficiency, the eager approach to SMT is no longer a state of the art technique.

Chapter 5

The SAT(gc) Framework

In this chapter, we will formulate a framework where a SAT solver is integrated with a constraint solver. More precisely, the integration focuses on an efficient feature of CP, namely global constraints. We will first present the motivation for the work, followed by the details of a solver algorithm. The chapter will be concluded with a comparison to related work.

5.1 Motivation

5.1.1 Cross fertilization of SAT and CP

Modern DPLL based SAT solvers are very efficient and they are robust enough to solve some large real life problems without heavy tuning, which is typically required by the CP tools. On the other hand, SAT has some limitations in expressiveness, that is, not every problem can be expressed compactly in Boolean format.

As commented earlier, SAT and CP share many common traits in problem solving. In recent years, cross fertilization of these two areas has become a topic of interest. It is argued that complex real world applications may require effective features of both. In connection to this idea, a workshop on SAT and CP integration was organized by the CP community in 2006 [5]. Around the same time, a survey paper especially focusing on this issue by top researchers from both research communities was published [7].

5.1.2 Why global constraints

Global constraints are the most effective feature of CP and naturally this fact has motivated us to focus on these built-in constraints. Global constraints are efficient because special purpose algorithms are implemented for effective constraint propagation. These algorithms are taken from the domain of graph theory and operations research. Global constraints are also useful in modeling CSP. They can be used to replace an equivalent conjunction of more primitive constraints.

An example can be used to illustrate why the use of global constraints can improve the efficiency of constraint solving. For some problems, like proving that the pigeon hole problem is unsatisfiable, the branch and prune technique takes exponential time. But the use of the global constraint $allDiff(x_1, x_2 \ldots x_n)$ can make exponential difference as it is typically implemented with a specialized algorithm. The deduction techniques classically used in constraint solvers consider different constraints independently and do not use the global information on the group of inequalities. For proving unsatisfiability of the pigeon hole problem, it needs to explore a search space of size $(n-1)!$. By using a graph based algorithm, this $allDiff$ constraint can be propagated very efficiently [7]. If global constraints are used in modeling, then the conjunction of constraints are getting replaced by a single,

built-in constraint. Thus the usage of global constraints also simplifies the representation task.

5.1.3 Issues with SAT and CP

Modeling of Numbers and Numerical Constraints The choice between CP and SAT for a particular application is influenced by the question whether or not the application requires numerics extensively. CP provides arithmetics natively, while in SAT, numbers need to be converted into propositional logic. Though there are some techniques available to encode numbers in SAT, most of them are not direct and are complex to implement. Because of the Boolean nature, a SAT solver does not exploit the arithmetic operations supported by the processor. In contrast, in CP, numerical constraints are well supported and due to the internal representation of domain values as range of numbers, they have become space effective too. This property of CP is particularly important for applications like scheduling, which needs numerical constraints over variables typically with large domains [7].

Problem Structure Awareness The problem representation in SAT is flat and homogeneous, that is, all the constraints are represented as clauses of Boolean literals. The SAT solver does not get any information about the structure of the problem [7]. Even when there exist clear structures in problems, the SAT solver remains unaware of the structure, because the structure gets lost as they are encoded as clauses in propositional logic. On the other hand, CP provides a rich level of data structure (e.g., global constraints) to express the structures available in a problem instance. For this reason, in recent years research has been pursued to incorporate structures into SAT instances.

Efficiency of Modern SAT Solver Modern SAT solvers such as ZCHAFF and MINISAT [1] are very efficient in solving problems in the CNF format [7]. A CNF can be compactly stored in memory. Highly efficient deduction algorithms have been formulated and implemented over the years to perform deduction on CNFs. Efficient decision heuristics and the conflict driven learning mechanism are very effective in pruning the search space. As a result, the modern SAT solvers have become a highly specialized tool for solving CNF problem instances.

5.1.4 Motivation in a nutshell

By following the discussions in this chapter, we articulate our motivations in the following points:

1. The recent trend in the SAT community suggests that the central components of state of the art DPLL SAT solver have become saturated as far as accuracy is concerned. Pursuing research on fertilizing SAT by integrating efficient features of closely related paradigms is on the move. This fact has motivated us to pursue a research for the integration of global constraints in SAT solving.

2. The research on SMT is an attempt to enhance the applicability of DPLL based SAT solvers by incorporating a different theory satisfiability paradigm to the SAT paradigm. Following the trend in the SMT community, we became motivated to pursue the integration of SAT and CP on the hope that it may enhance the applicability of SAT solving.

3. Though the modern SAT solvers are highly efficient in solving problems expressed in the form of pure Boolean CNF, they are not the best choice in handling applications which require numerical constraints on variables with large domains. The modern

[1] http://minisat.se/

SAT solvers are not structure aware as well. Such an integration should introduce structure awareness to modern SAT solvers.

5.2 The SAT(gc) Framework

Here, we formulate SAT solving with global constraints, generally referred to as SAT(gc), where global constraints are embedded into SAT solving based on the DPLL architecture.

In this section, we will first provide some notations. Then we will present a solver algorithm, which is of the same structure of the standard SAT solver but deals with two main issues when embedding global constraints into a SAT solver. The first is that global constraints in formulas need to be represented in the language of SAT(gc) and handled in the solving process correctly. The second is the question of how to perform conflict-directed backtracking and learning in the presence of global constraints. We will follow the naming and style convention of the SAT solving procedures discussed in Chapter 2.

5.2.1 Language and notation

An instance of SAT(gc) is a conjunction of clauses, where a clause is a disjunction of literals. The difference from propositional clauses is that a literal here can be a *global constraint literal*, or just called a *gc-literal*, that represents a call to a global constraint. In this thesis, we require a gc-literal to appear in a clause positively. That is, our solver does not pursue the search of a satisfiability proof of a formula where a global constraint *must* be false. This assumption is due to the utility of global constraints in applications - forcing some constraints to be satisfied.

When we refer to a global constraint by a gc-variable, we mean a call to the underlying global constraint it represents. For example, suppose a gc-variable v_g in a clause represents a call to the global constraint allDifferent:

$$allDiff(x_0 : \{v_1, v_2\}, x_1 : \{v_2, v_3\}) \tag{5.1}$$

where x_0 and x_1 are CSP variables, each of which is followed by its domain, and the constraint requires x_0 and x_1 be assigned to distinct domain values.

Note that we could have just written expression (5.1) in clauses as a kind of meta atom. But for notational convenience, we will use a gc-variable to denote a call to a global constraint. Of course, in the implemented language, the correspondence between a gc-variable and its global constraint has to be recorded as part of input instance.

To see the representation power of the language of SAT(gc), let us consider a couple of examples. In the first, suppose given a 4 by 4 board where each cell contains a number from a given domain D, we want to state that at least one row has the sum of its numbers equal to a given number, say k. We can express this by

$$sum(x_{11} : D, \ldots, x_{14} : D, =, k) \vee ... \vee sum(x_{41} : D \ldots, x_{44} : D, =, k)$$

In the language of SAT(gc), one writes a clause consisting of four gc-variables as a shorthand:

$$v_{g_1} \vee v_{g_2} \vee v_{g_3} \vee v_{g_4}$$

with the correspondence between the gc-variables and global constraints recorded as part of input instance.

As another example, suppose we want to represent a conditional constraint: given a graph and four colors, $\{r, b, y, p\}$ (for red, blue, yellow, and purple), if a node a is colored with red, denoted by variable a_r, then the nodes with an edge from node a, denoted by $edge_{a,n_i}$ for node n_i, must be colored with distinct colors different from red. This can be modeled by

$$\neg a_r \vee \neg edge_{a,n_1} \vee \neg edge_{a,n_2} \vee ... \vee \neg edge_{a,n_m} \vee v_g$$

where v_g represents the global constraint

$$allDiff(x_{n_1} : \{b, y, p\}, ..., x_{n_m} : \{b, y, p\})$$

The idea of SAT(gc) solving is simple. As mentioned above, a global constraint is represented by a propositional variable, a gc-variable. The main difference from a standard Boolean variable is that a global constraint is true if and only if it is solvable.[2] The latter means the existence of one or more solutions. Such a solution can be represented by a collection of propositional variables, each of which is a proposition representing that a given CSP variable takes a particular value from its domain. Let us call such a proposition a *value variable* (it represents a CSP variable taking a value). Clearly, for a complete representation, for each CSP variable v and each value a in its domain, we need a value variable, say ϕ, and semantically, ϕ is true iff v is assigned to value a.

In the language of SAT(gc), we allow value variables to be explicitly written in clauses. This provides more convenience in representing concepts directly related to CSP variables. For example, the pigeon hole problem can be represented by an $allDifferent$ constraint where pigeons are CSP variables and holes are their domain values. One can in addition express that "pigeon-1 should be in hole `#2` or in hole `#3`". That pigeon-1 is in hole `#2`, for example, can be represented by a value variable.

With the language of SAT(gc) defined above, it should be clear that a gc-variable in a SAT(gc) instance is semantically equivalent to a disjunction of conjunctions of value variables, with each conjunction representing a solution of the corresponding global constraint. That is, a SAT(gc) instance is semantically equivalent to a propositional formula. Given a SAT(gc) instance Π, let us denote by $\sigma(\Pi)$ this propositional formula.

We now can state precisely what the satisfiability problem in the current context is:

> Given a propositional formula Π in the language of SAT(gc), determine whether there exists a Boolean variable assignment such that $\sigma(\Pi)$ evaluates to true.

In the rest of this thesis, we will use the following notations: Given a formula Π in SAT(gc),

- X_g is the set of CSP variables in global constraint g;
- $Dom(x)$ is the set of domain values of CSP variable x (in some global constraint in Π);
- $Val\_Var(x)$ is the set of value variables for the CSP variable x;
- Val_Var_Π is the set of value variables for all CSP variables appearing in Π;
- VG_Π is the set of variables that represent global constraints in Π;
- Var_Π is the set of standard Boolean variables appearing in Π, i.e., those Boolean variables that are not in Val_Var_Π, neither in VG_Π;
- $V_\Pi = Val\_Var_\Pi \cup Var_\Pi \cup VG_\Pi$.

We may omit subscript Π if Π is clear from the context.

Example The following Π is a formula in SAT(GC).

$$\Pi = ((e \vee f) \wedge v_g \wedge \neg a \wedge (\neg b \vee \neg g) \wedge \neg b \wedge \neg c \wedge \neg d)$$

[2] During search, if a global constraint is assigned to true or forced to be true, then a solution consistent with the current partial assignment should be generated, otherwise a conflict will result. This will be discussed later in the description of the solver.

where v_g is short hand for $c(x_0 : \{v_1, v_2\}, x_1 : \{v_2, v_3\})$, namely a global constraint under the name c, with CSP variables $\{x_0, x_1\}$, $Dom(x_0) = \{v_1, v_2\}$, $Dom(x_1) = \{v_2, v_3\}$, and $VG_\Pi = \{v_g\}$.

For a variable x_i and a value v_j, if we write the corresponding value variable by $x_{i_{v_j}}$, we then have

$$Val\_Var(x_0) = \{x_{0_{v_1}}, x_{0_{v_2}}\}$$
$$Val\_Var(x_1) = \{x_{1_{v_2}}, x_{1_{v_3}}\}$$
$$Val\_Var_\Pi = \{x_{0_{v_1}}, x_{0_{v_2}}, x_{1_{v_2}}, x_{1_{v_3}}\}$$
$$V_\Pi = \{d, e, f, g, x_{0_{v_1}}, x_{0_{v_2}}, x_{1_{v_2}}, x_{1_{v_3}}, v_g\}$$

A global constraint can have multiple solutions. We will denote the i^{th} alternative solution of the global constraint g as s_g^i.

We will use the following terms to differentiate different types of Boolean variables under manipulation.

- *value variable*: the variables in Val_Var_Π;
- *global constraint variable* (or *gc-variable*): the variables in VG_Π;
- *normal variable*: those in Var_Π.

Similarly, we will use the terms *value literal*, *gc-literal*, and *normal literal*, respectively.

5.2.2 A SAT(gc) solver

We formulate a SAT(gc) solver in Algorithm 7, which can be seen as an extension of Algorithm 2 of Chapter 2. Given an instance Π in SAT(gc), the solver first performs preprocessing by calling the function *gc_preprocess*() (Line 1, Algorithm 7), which applies the standard preprocessing operations, such as pure literal fixing and unit clause resolution on Π; however, it will not make any assignments on gc-variables. That is, a call to any global constraint will be delayed to the search stage in SAT(gc). If *gc_preprocess*() does not solve the problem, then following a predefined decision heuristic it proceeds to branch on an unassigned variable (Line 5, Algorithm 7). The decision heuristic assigns a value to the selected variable so that the assignment satisfies at least one clause in the clause database. After the decision for branching on a variable is done, the *gc_deduce*() procedure is invoked (Line 7, Algorithm 7).

Procedure *gc_deduce*() Recall that in standard BCP, we only have one inference rule, the *unit clause rule*, which derives unit literals. With the possibility of value variables to be assigned, either as part of a solution to a global constraint or as a result of decision or deduction, we need three additional propagation rules. In the following, when we say a gc-variable or a value variable is assigned, we mean the variable is either assigned by a decision or forced as a unit literal.

- **Complementary Addition (CA):** For any CSP variable x in a global constraint, only one value from its domain can be assigned. Thus, when a value variable $x_v \in Val\_Var(x)$ is assigned to true, the value variables representing that x is assigned to other domain values of x must be assigned to false.
- **Domain Propagation (DP):** When a CSP variable x is committed to a value a, all the occurrences of x in other global constraints must also commit to the same value. Thus, for any global constraint g and any $x \in X_g$, whenever x is committed to a, $Dom(x)$ is reduced to $\{a\}$. Note that "a CSP variable committed to a value" may result from a solution returned by the constraint solver, or from a positively assigned value variable, or from a singleton domain. Similarly, when a value variable is assigned to false, the corresponding value is removed from the domain of the CSP variable occurring in any global constraint.

```
1  status = gc_preprocess()
2  if status = KNOWN then
3      return status
4  while true do
5      gc_decide_next_branch()
6      while true do
7          status = gc_deduce()
8          if status == INCONSISTENT then
9              blevel = current_decision_level
10             gc_backtrack(blevel)
11         else if status == CONFLICT then
12             blevel = gc_analyze_conflict()
13             if blevel == 0 then
14                 return UNSATISFIABLE
15             else
16                 gc_backtrack(blevel)
17         else if status == SATISFIABLE then
18             return SATISFIABLE
19         else
20             break
```

Algorithm 7: An Iterative Algorithm for SAT(gc)

- **Global Constraint Rule (GCR):** If the domain of a CSP variable of a global constraint v_g is empty, v_g is not solvable, which is therefore assigned to false. If a global constraint v_g is assigned to true, the constraint solver is called. If a solution is returned, the value variables corresponding to the generated solution are added to the partial assignment; if no solution is returned, v_g is assigned to false.[3]

In the procedure *gc_duduce*(), BCP now consists of four propagation rules, the unit clause rule, CA, DP, and GCR, which are performed repeatedly, in an interleaving fashion, until no further assignment is possible.

Proposition 1. *Let* Π *be a* SAT*(gc) instance. For any partial assignment* π*, gc_duduce*() *generates the same extension* π' *of* π*, independent of the order in which the four propagation rules are applied.*

Proof. Without loss of generality and for the purpose of illustration, assume two orders of applying the deduction rules below.

$$(a) \ldots GCR^{i}, BCP^{i+p}, CA^{i+q}, DP^{i+r} \ldots$$
$$(b) \ldots DP^{j}, BCP^{j+p'}, GCR^{j+q'}, CA^{j+r'} \ldots$$

The superscript on a deduction rule above denotes the deduction step in which it is applied. Let PA^k denote the partial assignment after k^{th} deduction step is performed. In the above sequences a and b, as deduction by any of the four rules is monotonic in nature, if $j > i$, then for a GCR operation the relation $PA^i \subseteq PA^{j+q'}$ always holds, and similarly for all the other rules. This implies any assignment generated by any rule in a sequence is guaranteed to be generated by any different sequence. □

Since a global constraint v_g in Π is semantically equivalent to the disjunction of its solutions (in the form of value variables), when v_g fails and thus is assigned to false in the

[3]Note that this assignment to false is not forced

current assignment, the negation of the disjunction should be implied. But Algorithm 7 does not do this explicitly. Instead, it checks the consistency in order to prevent an incorrect assignment. In case of $INCONSISTENT$, the search backtracks to the current decision level (Line 10, Algorithm 7).

Then, SAT(gc) checks if any conflict has occurred. If yes, SAT(gc) invokes the procedure $gc\_analyze\_conflict()$ (Line 12, Algorithm 7), which performs the conflict analysis, possibly learns a clause, and returns a backtrack level/point.[4]

Procedure $gc\_analyze\_conflict()$ Before explaining how this procedure works, in the following we will review some necessary terms and concepts of DPLL based conflict analysis. The descriptions are based on the procedural process of performing (standard) BCP.

- *Antecedent clause (of a literal)*: the antecedent clause of a literal l is the clause which has forced an implication on l.
- *Conflicting clause*: the first failed clause, i.e., the first clause during BCP in which every literal evaluates to false under the current partial assignment.
- *Conflicting variable*: The variable which was assigned last in the conflicting clause.
- *Asserting clause*: the clause that has all of its literals evaluate to false under the current partial assignment and has exactly one literal with the current decision level.
- *Resolution*: The goal is to discover an asserting clause. From the antecedent clause *ante* of the conflicting variable and the conflicting clause *cl* (see Algorithm 8), resolution between the two combines *cl* and *ante* while dropping the resolved literals. This has to be done repeatedly until *cl* becomes an asserting clause.
- *Asserting level*: the second highest decision level in an asserting clause. Note that by definition, an asserting clause has at least two literals.[5]

Algorithm 8 is the pseudo-code for the procedure $gc\_analyze\_conflict()$. In the following we describe this algorithm in detail.

Similar to the one given in Algorithm 3, $gc\_analyze\_conflict()$ first finds a conflicting clause cl. Then it attempts to find an asserting clause using resolution, which is described by a while loop (Lines 2-31, Algorithm 8). Inside the loop, it first obtains the last failed literal *lit* in *cl* by *choose_literal*() (Line 3, Algorithm 8). After that, it checks the literal *lit*.

(a) If *lit* is a gc-literal, then it is clear that the conflict is due to the failure of the most recent call to the constraint solver for *lit*. That is, the call returned no solution. There are two subcases.

  (1) If no previous DP operation affected the domains of the CSP variables in the scope of *lit* and no call to the same gc-variable to the constraint solver has succeeded before, then this means that the gc-variable does not have any solution with the original domains of its CSP variables. In this case, the failure of *lit* is inherently in itself, hence only the other literals in *cl* may satisfy the clause. Thus, we simply drop *lit* from *cl* (Line 6, Algorithm 8). There are three subcases.

    (i) If *cl* becomes empty after dropping *lit*, the given SAT(gc) instance is not satisfiable (Line 9, Algorithm 8).

[4] A backtrack level leads to backtracking to the decision variable of that level, i.e., undoing all the assignments up to the decision variable of that level, while a backtrack point is a point of an assignment, which may or may not be a point of decision.

[5] The process of resolution can produce a unit clause, in which case no backjumping to any previous level is possible. So, in such case chronological backtracking is performed. We note that in the literature, this special case is sometimes not explicitly mentioned.

```
1  cl = find_conflicting_clause()
2  while !isAsserting(cl) do
3      lit = choose_literal(cl)
4      if lit is a gc-literal then
5          if no DP is performed on the variables in the scope of lit and lit never has
           succeeded then
6              drop lit from cl
7              if cl is empty then
8                  back_dl = 0
9                  return back_dl
10             else if cl is unit then
11                 back_dl = current_decision_level
12                 return back_dl
13         else
14             dl = decision_level(lit)
15             if lit is a decision literal then
16                 back_dl = dl − 1
17                 return back_dl
18             else
19                 back_dl = dl
20                 return back_dl
21     else
22         ante = antecedent(lit)
23         if ante == NULL then
24             back_dl = backtrack_point(lit)
25             return back_dl
26         cl = resolve(cl, ante, lit)
27         lit = choose_literal(cl)
28         ante = antecedent(lit)
29         if ante == NULL and lit is not a decision literal then
30             back_dl = backtrack_point(lit)
31             return back_dl
32 add_clause_to_database(cl)
33 back_dl = clause_asserting_level(cl)
34 return back_dl
```

Algorithm 8: Conflict Analysis in SAT(gc)

(ii) If cl becomes unit (i.e., cl has only one literal) after dropping lit, then it indicates that, cl cannot be an asserting clause (by definition asserting clause has at least two literals in it). So, in this case, we have to perform chronological backtracking. The procedure $gc\_analyze\_conflict()$ returns the current decision level as the backtracking level (Line 12, Algorithm 8).

(iii) Otherwise, continue with resolution.

(2) If the domain of a CSP variable in the scope of lit has been reduced or at least a solution was generated previously, then we have to perform chronological backtracking. In case that lit is a *decision variable* of the current decision level, $gc\_analyze\_conflict()$ simply returns the previous decision level as the backtracking level (Line 17, Algorithm 8); Otherwise lit is *forced* in the current decision level, in which case $gc\_analyze\_conflict()$ returns the current decision level as the backtracking level (Line 20, Algorithm 8).[6]

(b) If lit is not a gc-literal, it must be either a normal literal or a value literal. Any conflicting normal literal must have an antecedent clause, and a conflicting value literal may or may not have an antecedent clause, depending on how its truth value is generated.

If lit possesses no antecedent clause (i.e., $ante == NULL$) then it must be a value literal assigned by a DP or CA operation, which is

(1) triggered by a solution of a global constraint at the current decision level.

(2) or triggered by unit propagation/decision that forces a positive value literal assignment at the current decision level.[7]

In the case (b-1), SAT(gc) backtracks to the point where the corresponding global constraint is invoked, with the purpose of trying to generate an alternative solution for the same global constraint. In the case of (b-2), SAT(gc) backtracks up to the decision variable of the current decision level. For both of the cases, the backtrack point is identified by the procedure $backtrack\_point(lit)$ (Line 25, Algorithm 8).

In $gc\_analyze\_conflict()$, after the cases (a) and (b), inside the while loop, resolution is performed over cl and $ante$ which results in new cl (by removing literal lit and its complement from cl and $ante$ and then combining them).[8] Notice that, the resulting clause cl also has all of its literals evaluated to value 0 and is a conflicting clause. The $gc\_analyze\_conflict()$ procedure again checks the last assigned literal lit in cl. If $gc\_analyze\_conflict()$ finds that lit does not have any antecedent clause and lit is not a decision variable,[9] then it becomes the case of (b). Otherwise, this resolution process is repeated until cl becomes an asserting clause or either one of the above two cases (a) or (b) occurs. If an asserting clause is found, then the procedure $gc\_analyze\_conflict()$ learns the asserting clause cl (Line 32, Algorithm 8) and returns the asserting level as the backtracking level (Line 34, Algorithm 8).

[6]Note that the "last failed literal" identified by $choose\_literal()$ is done possibly repeatedly inside the while loop when cl is not yet an asserting clause. Thus, when a gc-variable is last failed, it must be at the current decision level.

[7]Let, x is a CSP variable and $Val\_Var(x) = \{a, b\}$. If a is unit propagated or decided to be true, then by CA operation we assign $\neg b$. During the unit propagation of $\neg b$, it may create a conflict. In this context, lit refers to $\neg b$.

[8]Note that after dropping lit from cl in case (a), the process can still come to the phase of resolution inside the while loop.

[9]Note that the last step of resolution produces an asserting clause cl. After any step of resolution, if lit possesses no antecedent clause, then lit can be triggered either by (b-1) or (b-2), or it can be the decision literal of the current level (decision variables do not posses any antecedent clause), where the last case can only occur after the last step of resolution (as lit being a decision variable guarantees cl to be asserting). This checking guarantees that, after any step of resolution in case antecedent clause of lit becomes NULL, if lit is a decision literal then cl is learned, otherwise search backtracks to the point obtained from $backtrack\_point(lit)$.

After *gc_analyze_conflict*() returns the backtracking level, procedure SAT(gc) checks the backtracking level *blevel*. If blevel is 0, then SAT(gc) returns UNSATISFIABLE (Line 14, Algorithm 7). Otherwise, it calls *gc_backtrack*(*blevel*) (Line 18, Algorithm 7).

Procedure *gc_backtrack*(*blevel*) The *gc_backtrack*(*blevel*) procedure distinguishes between different types of conflict cases by using the value of a flag variable (not explicitly shown in the Algorithm 8). Admissible values for this flag variable are different conflict types. When a conflict occurs, this flag variable is set to that conflict type. The *gc_backtrack*(*blevel*) procedure works as follows:

(a) If the backtracking level is obtained from an asserting clause, then the procedure *gc_backtrack*(*blevel*) backtracks to decision level *blevel* and unassigns all the assignments up to the decision variable of *blevel* + 1. After backtracking the learned clause (also asserting clause) *cl* becomes a unit clause and the execution proceeds from that point in a new search space within level *blevel* (which is obtained from line 33, Algorithm 8).

(b) Otherwise, we perform chronological backtracking (when *blevel* is obtained from line 11, 16, 19, 24, 30 of Algorithm 8 and line 9 of Algorithm 7) as follows:

  (1) If the backtracking point is obtained from the procedure *backtrack_point*(*lit*) (Lines 24, 30, Algorithm 8), then *gc_backtrack*(*blevel*) backtracks and unassigns assignments up to that backtrack point in the current decision level.

  (2) If conflict occurs because of a gc-literal fails to generate an alternative solution, then we backtrack to *blevel* and unassign assignments up to the decision variable of *blevel* (*blevel* is obtained from 11, 16, 19 of Algorithm 8).[10]

  (3) If inconsistency is detected during deduction (Line 8, Algorithm 7), the procedure *gc_backtrack*(*blevel*) performs backtracking similarly as (b-(2)).

  In case of (b-1), if the backtrack point is a gc-literal assignment, then after backtracking SAT(gc) attempts to generate another solution of the same gc-literal and the execution proceeds from that point. If no alternative solutions exist, then it becomes a case of failed gc-literal. In other cases of (b), after backtracking up to the decision variable of *blevel*, that decision variable is flipped. By flipping a variable, we mean to switch from one phase to the other for a normal variable or value variable, and switch from the current solution to the next one for a gc-variable; if a normal or value variable is already *flipped* or no further solutions for a gc-variable exist, then backtracking is meant to backtrack up to the decision variable of the preceding decision level (i.e., chronological backtracking).

Inside the SAT(gc) procedure, if *gc_deduce*() returns SATISFIABLE (Line 18, Algorithm 7), procedure SAT(gc) also returns SATISFIABLE. If *gc_deduce*() returns without creating any conflict and the formula is still not satisfied, then SAT(gc) goes for deciding another unassigned variable assignment.

5.2.3 Examples

In this section, we illustrate the solving process of SAT(gc) by using some examples. In the following, for a CSP variable x and a domain value v, we will write x_v for the value variable representing that x takes value v.

[10] After dropping a failed gc-literal (Line 6, Algorithm 8), if *cl* becomes unit, then we return the current decision level (Line 12, Algorithm 8) as the backtrack level. If that failed gc-literal is a *decision* literal, then we backtrack to the previous decision level of the current decision level.

Example 1 Suppose the SAT(gc) formula Π consists of

$$
\begin{array}{ll}
(c1) & x_{1_a} \vee x_{3_c} \vee v_{g_1} \\
(c2) & \neg x_{2_b} \\
(c3) & x_{3_g} \vee p \\
(c4) & \neg p \vee \neg x_{4_g} \vee q \\
(c5) & v_{g_2} \vee r \vee \neg p \\
(c6) & \neg q \vee \neg x_{4_g} \\
(c7) & \neg r \vee s
\end{array}
$$

where v_{g_1} and v_{g_2} are gc-variables denoting the calls

- $v_{g_1} : allDiff(x_1 : \{a, b, c\}, x_2 : \{b, c, d, e, f\}, x_3 : \{a, c, e, f, g\})$
  $v_{g_2} : allDiff(x_1 : \{a, b, c\}, x_4 : \{g, d, a\}, x_5 : \{a\})$

Using the notations introduced at the outside of this chapter, we have

- $VG_\Pi = \{v_{g1}, v_{g2}\}$, $X_{g_1} = \{x_1, x_2, x_3\}$, $X_{g_2} = \{x_1, x_4, x_5\}$
- $Dom(x_1) = \{a, b, c\}$, $Dom(x_2) = \{b, c, d, e, f\}$, $Dom(x_3) = \{a, c, e, f, g\}$,
  $Dom(x_4) = \{g, d, a\}$, $Dom(x_5) = \{a\}$
- $V_\Pi = \{p, q, r, s, v_{g1}, v_{g2}, x_{1_a}, x_{1_b}, x_{1_c}, x_{2_b}, x_{2_c}, x_{2_d}, x_{2_e}, x_{2_f},$
  $x_{3_a}, x_{3_c}, x_{3_e}, x_{3_f}, x_{3_g}, x_{4_g}, x_{4_d}, x_{4_a}, x_{5_a}\}$

With input SAT(gc) instance Π, *gc_preprocess*() in the SAT(gc) solver in Algorithm 7 assigns the value variable x_{2_b} to false (by unit clause $c2$). Then, the decision heuristics of SAT(gc) chooses an unassigned variable to assign a value. Suppose *gc_decide_next_branch*() chooses v_{g_1} and assigns it to be true. This is decision level 1. After performing the DP operation on v_{g_1}, the constraint solver is called for

$$allDiff(x_1 : \{a, b, c\}, x_2 : \{c, d, e, f\}, x_3 : \{a, c, e, f, g\})$$

Note here that, due to the assignment of x_{2_b} to false, the DP operation removes b from the domain of x_2. This call returns the first solution

$$s^1_{g_1} = \{x_1 : a, x_2 : c, x_3 : e\}$$

Hence, by applying GCR, v_{g_1} is assigned to true, the value variables x_{1_a}, x_{2_c} and x_{3_e} are assigned to true and the CA and DP operations are performed. During BCP on $\neg x_{3_g}$ (the complementary value literal generated by the CA operation on x_{3_e}), the normal literal p gets unit propagated (by $c3$). At this stage, no more deductions are possible at decision level 1, hence the call to *gc_deduce*() is returned.

Note that at this stage, the clauses that are not yet satisfied are:

$$
\begin{array}{ll}
(c4) & \neg p \vee \neg x_{4_g} \vee q \\
(c5) & v_{g_2} \vee r \vee \neg p \\
(c6) & \neg q \vee \neg x_{4_g} \\
(c7) & \neg r \vee s
\end{array}
$$

Now, suppose *gc_decide_next_branch*() decides the normal variable r to be false at decision level 2. During BCP for $\neg r$, the unit literal v_{g_2} is implied (by $c5$). Thus, a call to the constraint solver is made for

$$allDiff(x_1 : \{a\}, x_4 : \{g, d, a\}, x_5 : \{a\})$$

Notice the domain reduction on x_1 which is made by the DP operation after the call to v_{g_1} generated a solution.

The above invocation returns no solution. By applying GCR, $\neg v_{g_2}$ is added to the partial assignment. During BCP over $\neg v_{g_2}$, a conflict occurs (in $c5$).

At this point, *gc_deduce*() returns and a call to *gc_analyze_conflict*() is made. Inside the while loop of *gc_analyze_conflict*(), it finds that the conflicting variable is the gc-variable v_{g_2}. It also finds that, DP operation was performed previously on one of the CSP variable of gc-variable v_{g_2} (on x_1) and v_{g_2} is a forced assignment at the current decision level (decision level 2). Thus the procedure *gc_analyze_conflict*() returns the current decision level as the backtracking level. As described earlier, the *gc_backtrack*(*blevel*) backtracks up to the current decision level and unassigns all assignments up to the decision variable assignment $\neg r$ and assigns r to be true. This assignment satisfies the clause $c5$ and $c7$ (s is unit propagated from $c7$). At this point, no more deductions are possible. The remaining unsatisfied clauses are:

$$(c4) \quad \neg p \vee \neg x_{4_g} \vee q$$
$$(c6) \quad \neg q \vee \neg x_{4_g}$$

Then, the search process goes to the next decision level. Suppose, at this decision level 3, the procedure *gc_decidie_next_branch*() chooses the variable q to be true. The assignment q satisfies the clause $c4$. During BCP over q, $\neg x_{4_g}$ is unit propagated from the clause $c6$.

At this stage, all the clauses are satisfied. As satisfiability is already determined, the remaining unassigned variables are assigned arbitrarily. This is how the instance Π is solved by SAT(gc).

Example 2 Suppose a part of a SAT(gc) formula Π consists of

$$(c1) \quad e \vee f$$
$$(c2) \quad \neg e \vee \neg p$$
$$(c3) \quad p \vee v_g$$

where v_g is a gc-variable denoting the calls

- $v_g : allDiff(x_1 : \{a\}, x_2 : \{a\})$

So, we have

- $VG_\Pi = \{v_g\}$, $X_g = \{x_1, x_2\}$
- $Dom(x_1) = \{a\}$, $Dom(x_2) = \{a\}$

Let the current decision level be dl, where the normal variable e is assigned to be true. This decision satisfies the clause $c1$. During the BCP of e, $\neg p$ is unit propagated from the clause $c2$. Same as e, during the BCP of $\neg p$, v_g is unit propagated from $c3$. So, SAT(gc) invokes the constraint solver. But the call returns no solution. So, by applying GCR, $\neg v_g$ is assigned. This creates a conflict on the clause $c3$.

At this stage, the procedure *gc_analyze_conflict*() is called. It identifies that, the conflicting literal is a gc-variable, and determines that no DP operation has reduced any of the domains of x_1 and x_2 previously and also no call to v_g have succeeded previously. So, *gc_analyze_conflict*() drops v_g from the conflicting clause cl (which is $c3$). After dropping v_g, cl becomes unit. So, as described previously, we backtrack to the decision level dl (current decision level) and unassign every assignments up to the decision variable e and flip the decision variable e to $\neg e$. It satisfies $c1$ (as f is unit propagated) and $c2$ and no call for the solving v_g is made. The search continues from there.

Example 3 Suppose a part of the SAT(gc) formula Π consists of

$$\begin{array}{ll} (c1) & e \vee f \\ (c2) & \neg e \vee \neg h \\ (c3) & h \vee v_g \vee p \end{array}$$

where v_g is a gc-variable denoting the calls

- $v_g : allDiff(x_1 : \{a\}, x_2 : \{a\})$

So, we have

- $VG_\Pi = \{v_g\}$, $X_g = \{x_1, x_2\}$
- $Dom(x_1) = \{a\}$, $Dom(x_2) = \{a\}$

Let the current decision level be dl, and suppose at a previous decision level dl', $\neg p$ was assigned. At the decision level dl, e is decided to be true. After performing BCP on e, $\neg h$ is unit propagated from clause $c2$. During the BCP of $\neg h$, v_g is unit propagated from clause $c3$. But the call to the constraint solver for v_g returns no solution and thus by applying GCR, $\neg v_g$ is assigned. This creates a conflict on clause $c3$.

The procedure $gc\_analyze\_conflict()$ is called. It identifies that, v_g is the conflicting variable and it is intrinsically unsolvable. So, v_g is dropped from cl (i.e., c3). After dropping v_g from cl, it becomes

$$h \vee p$$

So, resolution continues with cl. But since cl is an asserting clause, cl is learned and search backtracks to the decision level dl' (decision level of p) and unassign all the assignments up to the decision variable of the decision level $dl'+1$. After backtracking cl becomes unit and h is unit propagated from cl. The search continues from there.

Example 4 Suppose a part of the SAT(gc) formula Π consists of

$$\begin{array}{ll} (c1) & \neg r \vee d \\ (c2) & r \vee v_g \\ (c3) & t \vee s \vee \neg x_{1_a} \vee p \\ (c4) & t \vee s \vee \neg x_{1_a} \vee \neg p \end{array}$$

where v_g is a gc-variable denoting the calls

- $v_g : allDiff(x_1 : \{a\}, x_2 : \{b\})$

So, we have

- $VG_\Pi = \{v_g\}$, $X_g = \{x_1, x_2\}$
- $Dom(x_1) = \{a\}$, $Dom(x_2) = \{b\}$

Let the current decision level be dl, and suppose at a previous decision level dl', $\neg s$ and $\neg t$ were assigned. At the decision level dl, $\neg r$ is decided to be true. After performing BCP on $\neg r$, v_g is unit propagated from clause $c2$. The call to the constraint solver for v_g returns the CSP solution $\{x_1 = a, x_2 = b\}$. So, by applying GCR, the corresponding value variables x_{1_a} and x_{1_b} are assigned to true. But during BCP of x_{1_a}, a conflict occurs on $c4$. So, the $gc\_analyze\_conflict()$ is called.

The $gc\_analyze\_conflict()$ resolves $c4$ (conflicting clause) with $c3$ (antecedent clause of the conflicting variable p) and thus cl becomes

$$t \vee s \vee \neg x_{1_a}$$

Then in the same iteration of the while loop of $gc\_analyze\_conflict()$, it checks the last assigned literal lit ($\neg x_{1_a}$) in cl and finds that it is generated by the solution of v_g. So, it returns the assignment point of v_g as the backtrack point. The procedure $gc\_backtrack(blevel)$ unassigns every assignments upto the gc-variable v_g. At this stage, v_g is assigned again and the constraint solver is again called for generating the next alternative solution. But at this time v_g generates no alternative solution. So, by applying GCR, v_g is assigned to false. As a result, a conflict occurs on the clause $c2$. The procedure $gc\_analyze\_conflict()$ is again called.

It identifies that, the conflicting variable is a forced gc-variable v_g and a solution was previously generated for v_g. So, $gc\_analyze\_conflict()$ returns the current decision level as the backtracking level. The procedure $gc\_backtrack(blevel)$ backtracks up to the assignment $\neg r$, and flips it to r. This flipping immediately satisfies the clause $c2$ and the literal d is unit propagated from $c1$ (thus satisfying $c1$). The search continues from there.

5.3 Correctness of SAT(gc) Solver

We can argue informally that the SAT(gc) solver formulated in this chapter is correct in the following sense:

> Given a SAT(gc) instance Π, Algorithm 7 returns $SATISFIABLE$ if and only if Π is satisfiable, and Algorithm 7 returns $UNSATISFIABLE$ if and only if Π is unsatisfiable.

Of course, this is under the assumption that the constraint solver is sound and complete, and terminating. A proof sketch of the statement can be constructed based on the following arguments.

- If there are no occurrences of gc-variables in Π, then Algorithm 7 reduces to a standard SAT solver with the FirstUIP scheme in conflict analysis and backtracking. Its correctness, along with the correctness of other variants, have been shown formally by a transition system [52]. That is, any learned clause is a logic consequence of Π so their additions do not change the nature of satisfiability of Π.

- If conflict analysis and backtracking in Algorithm 8 does not involve any gc-variable, i.e., no "last failed literal" is a gc-literal and every "last failed literal" has an antecedent clause. Clearly, in this case the algorithm behaves in a way similar to standard SAT, as no conflict is due to the failure of a call to a gc-variable, nor it is due to the generation of a solution to a gc-variable.

- Otherwise, the conflict does involve a gc-literal. But note that in Algorithm 8 the only possibly learned clause involving gc-literal is one, which remains non-unit after lit being dropped from cl. It is clear that in this case the failure can only be possibly rescued by the rest literals in clause cl. It can be shown by induction on the number of resolution steps that such a learned clause follows from Π. Note that in other cases of involving a gc-literal, backtracking is chronological so that no possibility of any satisfying assignment may be missed.

- It is clear that Algorithm 7 is terminating, as it traverses a search tree where the nodes are normal variables, value variables, and gc-variables. Algorithm 7 employs two while loops, one is nested inside the other. The inner loop terminates when no more deduction is possible in a decision level. Inside the inner loop,

  - it invokes the $gc\_deduce()$ function. This $gc\_deduce()$ function always returns, because it makes implication on normal, value and gc-variables. For normal and value variables $gc\_deduce()$ returns because it makes propagation in a finite formula. The only difference for a gc-variable is that it may have only one phase,

false (the corresponding global constraint is not solvable), or one or more phases corresponding to one or more solutions. Then, the termination of the constraint solver guarantees that a call to the *gc_deduce*() function always returns.

– In case of conflict, the $gc\_analyze\_conflict()$ is called. A call to this function is also guaranteed to return as the while loop of Algorithm 8 is guaranteed to break for all the possible types of conflicts. An asserting clause is guaranteed to be found if the conflict does not involve any gc/value variable, in which case the while loop breaks normally. In other cases, if the conflict involves a gc/value variable, Algorithm 8 returns either a backtracking point or a backtracking level which ensures the termination of the while loop.

As a call to *gc_deduce*() and $gc\_analyze\_conflict()$ returns, the situation that "No further deduction is possible at the current decision level" is guaranteed to occur in Algorithm 7. So the inner loop of Algoritm 7 always breaks after finite number of iterations.

With conflict directed backtracking and chronological backtracking employed by Algorithm 7, a variable cannot be assigned to the same value twice with the same partial assignment. Under such setting, repeated exploration of the same search space is not possible. So, after a finite number of traversal in a finite search tree, the outer loop of Algorithm 7 also terminates, which ensures the termination of Algorithm 7.

- Finally, it should be clear that if $SATISFIABLE$ is returned with an assignment θ, then Π is satisfiable. The only situation where this may not be the case is when value variable assignments in θ imply a solution of a gc-variable while v_g is assigned to false in θ. This is prevented by checking of consistency.

5.4 SMT verses SAT(gc) : A Comparison

As one of our inspirations for integrating SAT with a constraint solver is SMT, here we shall compare SMT and SAT(gc) and discuss their similarities and differences. In this section, SMT refers to the lazy approach to SMT solving.

Both solving frameworks adopt a DPLL based SAT solver as the overall solver. The embedded component of an SMT solver is a theory solver and for SAT(gc) it is a constraint solver. The SMT solver uses a theory solver to determine the satisfiability of a portion of a T-formula. On the other hand, the SAT(gc) solver uses a constraint solver to compute a solution of a global constraint for which the constraint solver is invoked. In the following we compare SMT with SAT(gc), in terms of problem representation, deduction and learning.

1. In SMT, the T-formula is abstracted in a Boolean representation by encoding the theory components of the T-formula into propositional literals. In SAT(gc), we also make a Boolean abstraction by representing a global constraint as a gc-literal.

2. The SMT solver treats a theory literal as a Boolean literal and makes assignments on those literals as it does for other non-theory literals. Whenever a partial assignment is obtained from the DPLL solving, the SMT solver extracts the theory literal assignment from that partial assignment and checks the T-satisfiability. However, unlike SMT, in SAT(gc) solving, whenever a positive global constraint literal assignment is found in the partial assignment, the constraint solver is called to solve the global constraint. Once a solution is returned, the propositional literals corresponding to the returned solution are added to the partial assignment. So, during deduction, a theory solver is used for checking the T-satisfiability of the T-literals, but in SAT(gc) a constraint solver is used to compute a solution for the global constraint for which it is invoked. In other words, in SMT, the T-solver is used to determine the consistency of the literal assignments already made by the DPLL solver, while in SAT(gc), the constraint solver is used to make new literal assignments.

3. In SMT, whenever an inconsistent assignment is found by the T-solver, it informs the DPLL solver about the inconsistency and the T-solver sends information back to the DPLL solver as theory lemma, so that the DPLL solver can learn a clause and backtrack to a previous point. On the other hand, in SAT(gc) no such conflicting information is sent back from the constraint solver. The DPLL component of SAT(gc) identifies the conflicts/inconsistencies related to the global constraint at hand and does the necessary domain setup for the respective CSP variables, clause learning and backtracking. The constraint solver is used as a black box, to solve the global constraints for which it is called.

5.5 Constraint Answer Set Solving and SAT(gc) : A Comparison

In a related work [15], the authors have developed a framework for integrating CSP style constraint solving in Answer Set Programming (ASP), referred to as CDNL-ASPMCSP. Following the lazy SMT approach, the framework creates an abstraction of constraints presented in an ASP program. The ASP solver passes the portion of its (partial) Boolean assignment associated with constraints to a CP solver. The constraint solver checks these constraints against its theory via constraint propagation. The call results either in a unsatisfiability signal or extension of the current partial assignment by adding relevant constraint atoms. For conflicting driven analysis, every inferred atom needs a reason from which it is inferred. As CP solver does not provides any reason for the solutions it generates, this approach constructs a non-trivial reason from the structural properties of the underlying CSP problem in the ASP program at hand. For a constraint variable which is conflicting in question, other variables it shares constraints with are considered as potential reasons. From this constructed reason, it finds a learned clause and backtracking level by using resolution based conflict analysis process.

SAT(gc) is more eager than CDNL-ASPMCSP, in the sense that, whenever a Boolean literal associated with constraints (i.e gc-literal) is implied, it calls the constraint solver immediately. If any solution is returned, the current partial assignment is also eagerly extended by adding relevant value literals.

In contrast to CDNL-ASPMCSP, whenever a conflict involves a gc-literal or value literal (i.e., constraint atoms), SAT(gc) performs chronological backtracking. The more eager approach of SAT(gc) identifies immediate occurrence of conflict due to a gc-literal or a value literal (if any exists) and enables SAT(gc) to perform chronological backtracking, as the backtracking points are obvious.

Chapter 6

Implementation of a Prototype of SAT(gc)

To demonstrate the feasibility of the SAT(gc) framework, we have implemented a prototype system of SAT(gc), which we refer as SATCP. In SATCP, ZCHAFF [30], a state of the art DPLL based complete SAT solver is used as the DPLL engine and a constraint solver named GECODE [44] is used as the constraint solving engine. Both ZCHAFF and GECODE are open source, implemented in C++. In the next two sections, we will briefly review these two solvers followed by the details on our implementation of SATCP .

6.1 ZCHAFF - A State of the Art SAT Solver

ZCHAFF accepts a CNF problem instance in the DIMACS format [41] and determines its satisfiability. In the case that formula is satisfiable, ZCHAFF outputs a variable assignment along with some statistics.

ZCHAFF stores the input clauses in a clause database. A clause consists of a fixed number of literals. All the literals in a given problem instance are stored in a large vector, called the *literal pool*, in which each clause has a pointer, which points to its starting literal in the literal pool. All the other literals pertaining to that clause are stored consecutively after that starting literal. ZCHAFF uses 2-watched literal scheme for unit propagation as described in Chapter2. To implement it, while storing each clause in the clause database, it marks two of the clause's literals as watched.

After storing the clauses in the clause database, ZCHAFF starts its solving process with preprocessing, in which it makes assignments on unused variables, applies *pure literal fixing*and makes assignment on the literals of unit clauses.

If preprocessing cannot solve the whole problem, then ZCHAFF proceeds with deciding branching variables and performing unit propagation. For making decisions on variables, it uses the VSIDS heuristics as described in Chapter 2.

After a decision is made and the decision variable is added to the partial assignment, it performs unit propagation. As mentioned earlier, for deduction purposes, it uses 2-watched literal scheme, under which it only checks those clauses which have only one of its watched literal unassigned to make new assignments or for detecting conflicts. If any conflict is detected, ZCHAFF calls a conflict analyzer, which is described in Algorithm 3. As a stopping criterion of the while loop in Algorithm 3, it uses the most efficient stopping heuristics - namely, first UIP. After finding the first UIP, the conflict analyzer learns a clause and returns the backtracking level. The solver then backtracks to the returned backtracking level. After backtracking the learned clause becomes an unit clause. ZCHAFF then proceeds by assigning the first UIP, as it gets unit propagated from the learned clause. If the returned backtracking level is 0, ZCHAFF returns UNSAT, as backtracking to level 0 indicates that the

conflict has occurred as a reason of an assignment during preprocessing, which was made before any of the decisions are made.

By adopting the 2-watched literal scheme, conflict directed learning and backjumping, ZCHAFF has become one of the most efficient SAT solvers, as evidenced by its performance in recent SAT solver competitions. This was one of our motivations to select ZCHAFF to develop a prototype implementation of the SAT(gc) framework.

6.2 GECODE - A State of the Art CSP Solver

GECODE is a software library for developing applications for solving constraint satisfaction problems. Like ZCHAFF, it is a free software developed in C++. GECODE provides a constraint solver with state-of-the-art performance while being modular and extensible. GECODE won all MiniZinc Challenges[1] so far (in all categories): 2010, 2009, and 2008 [44]. Motivated by these facts, we have used GECODE as the constraint solver for the prototype implementation of SAT(gc). In the following subsections, we shall briefly describe the data structures, search engines and modeling techniques of GECODE [36].

Data structure In Chapter 3, we have described the idea of constraint store, where all the constraints, the CSP variables and their domain values are stored. GECODE employs a construct named SPACE for implementing the constraint store. SPACE is the data structure which is used for solving a CSP problem in GECODE . In GECODE , a problem is modeled as a space with propagators, variables and their domains added into the space. The propagator of the space are defined and executed when the space is first constructed.

A space has several functionalities. A space can *clone* itself. The clone of a space is placed in a stack so that the search engine can come to it later after backtracking. A space also has a functionality, called *status*, to check its status to see, whether the store is failed, solved or distributable. The functionality *propagate*, forces the space to do propagation, so that it can become either a failed, solved or a distributable store. If the space becomes a distributable store after propagation, GECODE performs branching on variables and values to perform the searching.

```
class model : public Space {
  public:
    IntVarArray x;
  public:
    model(void) : x(*this, 4, 1, 16) {
       linear(*this, x, IRT_EQ, 34);
       branch(*this, x,INT_VAR_SIZE_MIN, INT_VAL_MIN);
    }
    model(bool share, model& s) : Space(share, s){
      x.update(*this, share, s.x);
    }
    virtual Space* copy(bool share) {
       return new model(share,*this);
    }
}
```

Figure 6.1: Example of modeling a CSP problem in GECODE.

Search engines Every search engine in GECODE is implemented in two parts, namely, templated part and general part. The templated version handles the specialized space in which the problem at hand is specified and the more general part handles search for any space object. To perform searching on the specialized space, the templated part calls functions in

[1] http://www.g12.cs.mu.oz.au/minizinc/challenge2010/

the more general version of the search engine. Search engine in GECODE works by utilizing a stack for handling the backtracking of search engines. The stack keeps tracks of which nodes the search engine has previously been. When a search engine needs a new search space to work with, it asks for it to the stack. GECODE employs a hybrid of two techniques named, cloning[2] and recomputation[3] for restoring a space.

6.2.1 Problem modeling in gecode

In GECODE, a CSP problem is modeled in object oriented fashion. Models of GECODE are implemented using *spaces*. A model of the CSP problem is programmed in a subclass, which inherits a Space class. The constructor of the subclass implements the model, which includes specification of variables, their domains, propagator functions and branching orders. In addition to the constructor, the subclass must implement a copy constructor and a copy function as a requirement to facilitate the searching. An object of the subclass and an object of a search engine, which is connected with the subclass instance must be created. The instance of the search engine can be used to obtain any solution of the CSP problem, specified in the constructor of the subclass.
Figure 6.1 depicts a skeleton code for modeling a CSP problem in GECODE . The CSP problem involves finding a solution which involves four variables with domains ranging from 1 to 16. The problem asks for a solution on the variables with the constraint "sum of these four variables must be equal to 34".

6.3 Integration of ZCHAFF and GECODE

To implement SATCP, we have integrated ZCHAFF and GECODE. In our implementation, we have used GECODE as a black box to get solutions of global constraints when needed. To develop SATCP, we have minimally modified ZCHAFF to provide the hook for the integration. Our implementation is in some sense incomplete, but it turns out to be sufficient for the experimentation presented in this thesis. The changes are as follows:

- The SAT(gc) framework allows a gc-variable to be assigned at any point of the search process (i.e., by decision or by unit propagation). But SATCP assigns the gc-variables only as decision variables.
- In the SAT(gc) framework, DP operations are performed immediately after a value variable is assigned to a value. But in SATCP, the DP operation is performed on the CSP variables of a gc-variable before invoking GECODE to obtain a solution of that gc-variable, based on the current partial assignment.
- In the SAT(gc) framework, when no more solution can be generated for a gc-variable, we assign that gc-variable negatively and a conflict is detected immediately. In SATCP, when no more solution can be generated for a gc-variable, instead of assigning it negatively, SATCP raises a flag to indicate that the conflict has occurred as no more solution can be generated for the last assigned gc-variable.
- In the SAT(gc) framework, when BCP for a value variable corresponding to a solution of a gc-variable creates a conflict, that conflict is identified inside the conflict analyzer (by checking *lit*, which does not have an antecedent clause). But in SATCP, in such case, it raise a flag to indicate that a conflict has occurred as the last alternative

[2]Cloning creates a clone of a space. Before following a particular alternative path the space at hand is stored in the stack and restored and used later, if required.

[3]Instead of storing the entire space, the recomputation technique stores the information to redo the effect of the brancher. The stored information is called a *choice*. Redoing the effect of earlier choice in a space is called *commit a space*. When an alternative space is asked for, then given a space, the stored *choice* is used to commit to that space.

solution by a gc-variable is conflicting. This flag-based identification of the two types of conflicts enable us to develop SATCP with minimal modification to the existing conflict analyzer of ZCHAFF.

- Under the setting described above, it is not possible that a partial assignment ψ implies a solution of a gc-variable v_g while v_g is assigned to false in the assignment. Therefore, an implementation of consistence checking is not needed for the intended experimentation, and is thus not carried out.

The implementation of SATCP follows Algorithm 7. In the following subsections, we will describe the implementation SATCP in terms of its functionalities.

6.3.1 Preprocessing

From the application point of view the SAT(gc) encoding of a problem instance has the following properties: A gc-variable occurs only in a unit clause positively. Due to this, we have changed the preprocessing function of ZCHAFF by ignoring *pure literal fixing* for gc-variables. They are not assigned during the preprocessing step. Otherwise, a series of invocations to GECODE would occur during the preprocessing step and we would need to implement the conflict analysis mechanism to handle the conflicts involving gc-variables inside the preprocessing step of ZCHAFF. In that case, we would need to modify ZCHAFF heavily.

6.3.2 Decision

Intuitively, a solution of a gc-variable tends to make a good amount of implications, typically due to shared CSP variables in different calls to global constraints. This intuition has lead us to implement a variable selection heuristic, which puts higher priority on gc-variables of a SAT(gc) formula Π. The order in which gc-variables are considered for decision is determined by the order of appearance of gc-variables in Π. We term this heuristic as *gc_heuristic*, which ensures that gc-variables are assigned only as decision variables. All the value literal assignments get the same decision level as the gc-variable, which generates them.

After all the gc-variables are assigned, the remaining unassigned propositional variables are assigned by *VSIDS*. Thus the decision heuristic that we have used in SATCP is essentially a combination of *gc_heuristic* and *VSIDS*.

6.3.3 Deduction

The deduction function of SATCP utilizes a queue, named *implication_queue*, to temporarily store the implications it finds in the current decision level. The deduction function pops out a literal assignment from the *implication_queue*, adds it to the current partial assignment and performs BCP for that literal assignment. During BCP, if it finds more implications which are added to the *implication_queue*. This operation is performed repeatedly until the *implication_queue* becomes empty (i.e no more deduction is possible in the current decision level) or a conflict due to a gc-variable assignment occurs.

The variable assignment that the deduction function pops out from the *implication_queue* must be one of these three types of variable assignment: a gc-variable assignment, a value variable assignment and a normal variable assignment. Depending on the type of the variable assignment SATCP performs deduction operation as follows:

- If the variable is a gc-variable, then the deduce function of SATCP invokes GECODE to obtain a solution for that gc-variable. Before calling GECODE, according to the current partial assignment, DP operations are performed on the CSP variables of that gc-variable. To perform the DP operation on a CSP variable x, SATCP drops a domain

value v from $Dom(x)$, if the corresponding value variable of the domain v is defined negatively in the current partial assignment.

After the invocation to GECODE returns, one of the following two things can happen:

- A CSP solution is returned. The gc-variable is assigned to a positive value and added to the partial assignment. The value literals corresponding to the returned CSP solution are added as positive assignments and for all of them BCP is performed. For each positively assigned value variable, CA operation (see Chapter 5) is performed. BCP is also performed for the value literals added as part of the CA operation.
- No solution is returned. It means that, with the current domain of the CSP variables of the gc-variable, no more alternative solution can be generated.

If any conflict occurs during BCP of any of the value variables generated by the solution of a gc-variable, the deduce function assigns a flag, named $conflict\_flag$ to *value_literal_conflict* and returns. If no more alternative solutions can be generated for a gc-variable, the deduction function assigns the $conflict\_flag$ to $gc\_failure\_conflict$ and returns. The conflict analyzer checks the value of this $conflict\_flag$ to see what type of conflict has occurred.

- Otherwise, the deduce function of SATCP adds the variable assignment to the current partial assignment. If the variable assignment is a positive value variable assignment, then we perform the CA operation. For every added assignments BCP is performed.

  If conflict occurs during the BCP, the conflicting clause is stored. The stored information is utilized during the conflict analysis.

If no conflict occurs during the BCP and *implication_queue* is empty, SATCP again calls its decision function for deciding next variable assignment. If any conflict occurs, then SATCP calls its conflict analyzer function.

6.3.4 Conflict analysis and backtracking

For conflict analysis in SATCP, we have modified the existing conflict analyzer function of ZCHAFF slightly. We also have slightly modified the existing backtracking function of ZCHAFF.

The conflict analyzer function of SATCP works as follows:

- If it finds that the $conflict\_flag$ is assigned to *value_literal_conflict*, then it indicates that, the current alternative solution of the last assigned gc-variable (at the current decision level) is conflicting. The conflict analyzer of SATCP returns the current decision level as the backtracking level. The backtracker of SATCP backtracks up to the decision variable (which is the last assigned gc-literal) of the current decision level. After backtracking the decision procedure again decides on the same gc-variable (by *gc_heuristic*), and the next alternative solution for that gc-variable is generated.

- If it finds that, $conflict\_flag$ is assigned to *gc_failure_conflict*, then it indicates that the most recent invocation to GECODE for the last assigned gc-literal fails to generate any solution. This failure is a conflict.

  The conflict analyzer of SATCP returns the previous decision level of the current decision level as the backtracking level. The backtracker of SATCP backtracks up to that level and unassigns all the assignments up to the decision variable of that decision level. At the previous decision level, the decision variable is guaranteed to be another gc-variable. After backtracking to the previous decision level, the decision function of SATCP again chooses that gc-variable and as the deduce function executes, the next alternative solution for that gc-variable is generated.

Notice that, if the first assigned gc-variable (at decision level 1) fails to generate any solution, the conflict analyzer returns decision level 0 as the backtracking level, in which case satCP declares the problem unsatisfiable.

- If the $conflict\_flag$ is not set to any value, then it indicates that, conflict occurs due to a forced implication (i.e the conflicting variable has an antecedent clause), then we analyze this type of conflict by the existing conflict analyzing mechanism of zchaff. As described in Algorithm 8, The conflict analysis proceeds by repeatedly performing resolution, until
  - an asserting clause is found. The conflict analyzer learns the asserting clause, returns the asserting level as backtracking level *blevel*. The backtracker backtracks up to *blevel* and unassigns every assignment up to the decision variable of the level $blevel + 1$. After backtracking the learned clause becomes an unit clause. The first UIP literal[4] is unit propagated from the learned clause.
  - or a clause *cl* is found in which the last assigned literal *lit* (see Algorithm 8) has no antecedent clause. It indicates that the value variable assignment is generated by a solution of the gc-variable at the current decision level. This assignment of *lit* is responsible for the current conflict. Thus, we need to obtain another alternative solution for the last assigned gc-variable. The conflict analyzer returns the current decision level as the backtracking level. The backtracker of satCP backtracks up to the current decision level. After backtracking the decision function of satCP again decides on the same gc-variable, and the next alternative solution for that gc-variable is generated.

6.4 GC-variables and Search Engines

6.4.1 Creating models and search engines in satCP

As mentioned earlier, CSP problems are modeled in gecode by creating a subclass of a built-in class named Space and specifying the model inside that subclass (see Subsection 6.2.1). The solution for a CSP model are searched by creating search engines [45].

From the CP perspective, every global constraint v_{g_i} in a SAT(gc) instance Π is an independent CSP problem. So, before starting executing satCP, for every gc-variable $v_{g_i} \in \Pi$ we create a subclass of the class Space, which models the global constraint $v_{g_i} \in \Pi$ as an independent CSP model (for instance, as shown in Figure 6.1). At the very beginning of the execution of satCP, for each of the CSP models it creates a search engine globally.

6.4.2 Searching for CSP solutions

satCP uses the Branch and Bound (BAB) search engine, which is a built-in search engine of gecode [45].

The BAB search engine in gecode has a public method, named *next*() [45], which provides the next alternative solution for the CSP model to which it is attached. The *next*() method returns *null* when there is no more alternative solution exists for the attached model.

Whenever a gc-variable v_{g_i} is assigned, satCP executes the *next*() method of the search engine attached to the model of v_{g_i} to get the next alternative solution for that v_{g_i}. When the attached search engine does not find any alternative solution for v_{g_i}, it returns *null* and expires. If that v_{g_i} is assigned again, satCP creates a new search engine (as the previous search engine for v_{g_i} has expired) at the local scope which searches for alternative solutions for v_{g_i}. Here, one point is worth mentioning. Every time an alternative solution needs to be generated for such a v_{g_i} (once failed), a local search engine needs to be created. So, for getting the i^{th} alternative solution for such v_{g_i}, $i-1$ solutions need to be generated.

[4]The one and only literal assigned at the current decision level in the asserting clause.

Chapter 7

Experiments

In this chapter we will present the experiments that we have done with SATCP, a prototype implementation of SAT(gc). We use four benchmark problems. These are - the Latin square problem, the magic square problem, the ferry planning problem with numerical constraints, and the planning problem of block stacking with numerical constraints. In solving the Latin square problem, SATCP is more efficient than SAT. The magic square problem has given us some important insights about the integration of SAT and CP. These insights have motivated us to design the two planning problems as mentioned above. In this connection, these two benchmarks have demonstrated new usability of SAT(gc).

The experiments are conducted in a UNIX server assembled with 2.10 GHZ Intel Core 2 Duo CPU (T5600) and 4GB of RAM. For all the experiments, the cut off time is 15 minutes, that is, we manually terminated the execution of SATCP for any instance which did not give output within 15 minutes.

In the rest of this chapter, we will describe these benchmarks, their encodings in SATCP, experimental results, and an analysis of these results. But before that, let us briefly discuss how SAT(gc) instances are represented in SATCP.

7.1 Encoding of SAT(gc) Formula

7.1.1 SAT instance representation in zchaff

ZCHAFF accepts SAT problem instances encoded following the DIMACS CNF format [9]. In the DIMACS CNF format, a SAT instance starts with a problem line, which has the following format

$$p \text{ CNF } var\_num \ cls\_num$$

where p signifies that the line is the problem line, by CNF it signifies that the format of that problem is in the CNF format, var_num and cls_num are two integers signifying the number of variables and the number clauses in that problem instance, respectively. The clauses appear right after the problem line. The variables that form the clauses are numbered from 1 to var_num. The non-negated version of variable v is presented by v and the negated version of variable v is presented as $-v$. A clause is presented by a sequence of numbers, each separated by a space. The termination of a clause is indicated by the sentinel 0. Termination of SAT formula is indicated also by the sentinel 0. For example, the DIMACS CNF format of the SAT problem

$$(p_1 \vee p_2 \vee \neg p_4) \wedge (p_4) \wedge (p_2 \vee \neg p_3)$$

is represented by

p CNF 4 3
1 3 -4 0
4 0
2 -3 0
0

7.1.2 Representation in SAT(gc)

As we adopt ZCHAFF as the SAT engine in our implementation, we will use the DIMACS CNF format in our prototype implementation, where

- zero or more global constraints are incorporated into a SAT instance;
- global constraints are represented by integer numbers starting from $var\_num + 1$; and
- the correspondence between the value variables and their corresponding domain values is maintained in such a way that the mapping between a domain value and its corresponding value variable can be easily performed.

Table 7.1 shows a sample SAT(gc) instance Π_0 on the left column, with 5 variables, 4 clauses, and 1 global constraint which is

$$gc(x_1 : \{10, 12\}, x_2 : \{10, 12\}).$$

Our prototype implementation takes a formula such as Π_0 and converts it to the DIMACS CNF format.

Π_0	Converted to flat CNF
p cnf_gc 5 4	p cnf_gc 6 4
1 2 0	1 2 0
1 -3 4 0	1 -3 4 0
$gc(x_1 : \{10, 12\}, x_2 : \{10, 12\})$ 0	6 0
3 5 0	3 5 0
0	0

Table 7.1: A SAT(gc) instance.

The right column of Table 7.1 shows the converted CNF formula of Π_0 in the DIMACS CNF format. Notice that, the global constraint

$$gc(x_1 : \{10, 12\}, x_2 : \{10, 12\})$$

is encoded by the propositional variable 6 ($var\_num + 1$).

As a domain value of a CSP variable corresponds to a value variable, we need to store the correspondence between the domain values and the value variables. We call this storage a *mapping dictionary*. In a mapping dictionary, the correspondence between a value variable and its corresponding domain value is presented by a pair of numbers separated by "/" as *value_variable/domain_value*. The number on the left side of "/" is a value variable and the number on the right side of "/" is its corresponding domain value. Table 7.2 shows the mapping dictionary for the example in Table 7.1. The two rows of Table 7.2 show the correspondence between domain values and value variables of the CSP variables x_1 and x_2.

x_1	1/10 2/12
x_2	3/10 4/12

Table 7.2: Mapping dictionary.

7.2 The Latin Square Problem

The Latin square can be defined as follows:

> *A Latin square of order n is an n by n array of n numbers (symbols) in which every row and columns must contain distinct numbers (symbols).*

Figure 7.1 shows an example of Latin square of order 3.

A	B	C
C	A	B
B	C	A

Figure 7.1: A Latin square of order 3.

The problem of finding a Latin square of order n is called the *Latin square problem* of order n.

7.2.1 Encoding of the Latin square problem in SAT(gc)

The SAT encoding of Latin square of order n requires n^3 propositional variables. Each propositional variable p_{ijk} denotes that a number k is assigned to the cell belonging to the i^{th} row and j^{th} column, where $1 \leq i, j, k \leq n$ [27].

Like SAT encoding, to encode our SAT(gc) instance Π_{latin} of Latin square of order n, we use n^3 propositional variables. An obvious way to encode this problem in SAT(gc) is to encode it by using both SAT clauses and the *allDiff* global constraint. We encode the constraint "no two numbers are assigned to the same cell" by clauses of propositional literals and the constraint "every number must appear exactly once in a row and in a column" is encoded by using n *allDiff* global constraints as follows:

- Constraint 1: No two numbers are assigned to the same cell.

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \bigwedge_{k=1}^{n} \bigwedge_{l=k+1}^{n} (\neg p_{ijk} \vee \neg p_{ijl})$$

- Constraint 2: No row and column has the same number repeating on them. This constraint assigns different numbers in different cells of a row and column. We encode this constraint using n calls to the *allDiff* global constraint as follows:

$$\bigwedge_{k=1}^{n} allDiff(x_1^k, x_2^k, \ldots, x_n^k)$$

where

$$Dom(x_1^k) = Dom(x_2^k) = \cdots = Dom(x_n^k) = \{1 \ldots n\}$$

The assignment $x_i^k = j$ (where $1 \leq i, j, k \leq n$) asserts that the number k is placed on the i^{th} row and j^{th} column of the square. Thus each of the *allDiff* global constraints assigns a particular number k in n different cells of the n by n Latin square (see Subsection 7.2.3).

7.2.2 Experimental results

We have run the Latin square problem on ZCHAFF (with SAT encoding) and on our prototype

n	Running Time for ZCHAFF (in seconds)	Running Time for SAT(gc) (in seconds)
3	0	0
4	0	0
5	0	0
6	0.01	0.01
7	0.02	0.01
8	1.42	0.30
9	48.20	10.02
10	—	169.51

Table 7.3: Experiments with Latin square.

implementation system, SATCP (with SAT(gc) encoding Π_{latin}) for instances of different sizes, starting from order of 3 to order of 10. The running time for ZCHAFF and for SATCP are shown in Table 7.3, where we can see that, for smaller instances, the prototype system runs in similar speed as ZCHAFF. But for larger instances, the former clearly runs faster than the latter. For example, for instance with order 10, SATCP solves it within 170 seconds, while ZCHAFF was unable to determine the satisfiability within 15 minutes. This is why for Latin square problem, SATCP is more efficient than ZCHAFF.

The next subsection analyzes this performance of the prototype implementation of the SAT(gc) framework, by taking a small SAT(gc) instance of Latin square as an example.

7.2.3 Solving Latin square of order three

In this subsection we present an analysis of the performance of SATCP by examining some of the details for Latin square of order 3, with three numbers n_1,n_2 and n_3.

For encoding this problem in SATCP, we require 27 (3^3) propositional variables starting from 1. Let the propositional variables have the following interpretation:

$$1: n_1 \textit{ is on cell } (1,1),\ 2: n_2 \textit{ is on cell } (1,1),\ 3: n_3 \textit{ is on cell } (1,1)$$
$$4: n_1 \textit{ is on cell } (1,2),\ 5: n_2 \textit{ is on cell } (1,2),\ 6: n_3 \textit{ is on cell } (1,2)$$
$$7: n_1 \textit{ is on cell } (1,3),\ 8: n_2 \textit{ is on cell } (1,3),\ 9: n_3 \textit{ is on cell } (1,3)$$
$$10: n_1 \textit{ is on cell } (2,1),\ 11: n_2 \textit{ is on cell } (2,1),\ 12: n_3 \textit{ is on cell } (3,1)$$
$$13: n_1 \textit{ is on cell } (2,2),\ 14: n_2 \textit{ is on cell } (2,2),\ 15: n_3 \textit{ is on cell } (3,2)$$
$$16: n_1 \textit{ is on cell } (2,3),\ 17: n_2 \textit{ is on cell } (2,3),\ 18: n_3 \textit{ is on cell } (3,3)$$
$$19: n_1 \textit{ is on cell } (3,1), 20: n_2 \textit{ is on cell } (3,1), 21: n_3 \textit{ is on cell } (3,1)$$
$$22: n_1 \textit{ is on cell } (3,2),\ 23: n_2 \textit{ is on cell } (3,2),\ 24: n_3 \textit{ is on cell } (3,2)$$
$$25: n_1 \textit{ is on cell } (3,3),\ 26: n_2 \textit{ is on cell } (3,3),\ 27: n_3 \textit{ is on cell } (3,3)$$

As shown in Table 7.4, we encode this problem with CNF clauses of propositional variables (to encode "No two numbers are assigned to the same cell") and with 3 $allDiff$ global constraints (to encode "No row and column has the same number repeating on them"). Column 3 of Table 7.4 shows the converted flat CNF of the SAT(gc) formula, where the three *allDiff* global constraints are encoded as propositional variables 28, 29 and 30 respectively. That is, 28, 29 and 30 are the three gc-variables in this Π_{latin} instance of order 3. Column 4 shows the mapping dictionary for the problem.

Figure 7.2 shows the implication graph for the Π_{latin} instance in Table 7.4. In Figure 7.2, the dashed box attached to the gc-variables shows the consistent value variables of the

Constraint1	Constraint 2	CNF Clauses	Mapping Dictionary
-1 -2 0	$allDiff_1(x_1^1, x_2^1, x_3^1)$	-1 -2 0	x_1^1 1/1 2/4 3/7
-1 -3 0		-1 -3 0	x_2^1 1/10 2/13 3/16
-2 -3 0		-2 -3 0	x_3^1 1/19 2/22 3/25
-4 -5 0	$allDiff_2(x_1^2, x_2^2, x_3^2)$	-4 -5 0	x_1^2 1/2 2/5 3/8
⋮		⋮	x_2^2 1/11 2/14 3/17
-25 -26 0		-25 -26 0	x_3^2 1/20 2/23 3/26
-25 -27 0	$allDiff_3(x_1^3, x_2^3, x_3^3)$	-25 -27 0	x_1^3 1/3 2/6 3/9
-26 -27 0		-26 -27 0	x_2^3 1/12 2/15 3/18
		28 0	x_3^3 1/21 2/24 3/27
		29 0	
		30 0	

Table 7.4: Coding details of Π_{latin} of order 3.

CSP variables of the corresponding global constraints after DP operation being performed. The implications generated by the invocation to GECODE, are shown by green arrows. These generated implications make more implications - both by the unit propagation and by applying the CA operation. Implications generation by unit propagation are shown by blue arrows, while the generation of implications by CA operation are shown by red arrows.

For the Π_{latin} instance specified in Table 7.4, *gc_heuristic* first chooses the gc-variable 28 and assigns it to true. Note that as no assignment is made by *gc_preprocess*(), no change occurs in the domains of x_1^1, x_2^1 and x_3^1 (as reflected in the dashed box attached to gc-literal 28). The call to the GECODE generates 3 implications for this global constraint (green arrows). This 3 implications again generate more implications both by unit propagation (blue arrows) and by CA operation (red arrows). At this point, no more literals can be implied. Then *gc_heuristic* chooses the next gc-variable 29 and makes a positive assignment on it. Before calling the constraint solver for a solution for 29, DP operations are performed on its CSP variables. As a result, one value from each of its CSP variables domain are dropped (indicated on the attached box of 29). After the call to GECODE for $allDiff_2(x_1^2, x_2^2, x_3^2)$ returns, as before some more literal assignments are generated. DP operations are also performed on the CSP variables of $allDiff_3(x_1^3, x_2^3, x_3^3)$ (30). After all the possible implications are made, then the gc-variable 30 is decided at decision level 3. Notice that, DP operations are performed on the CSP variables of the gc-variable 30. As a result, the domains of its CSP variables become singleton (indicated in the attached box of 30). The invocation of GECODE for this global constraint returns a solution, and the corresponding value literals are assigned (green arrow). At this point, all the literals are assigned and thus the instance Π_{latin} becomes satisfiable.

Figure 7.3 shows how three calls for the three different global constraints solve Latin square of order 3. The left, middle and right squares show the status of the Latin square, after the value variables $(1, 13, 25)$, $(5, 17, 20)$ and $(9, 12, 24)$ are assigned (which are induced by the solutions of $allDiff_1$, $allDiff_2$ and $allDiff_3$ respectively). These solutions map different numbers in different cell position in the following way:

- $(1, 13, 25) \mapsto$ (n_1 is on cell (1,1),n_1 is on cell (2,2), n_1 is on cell (3,3)) (Figure 7.3 left)
- $(5, 17, 20) \mapsto$ (n_2 is on cell (1,2),n_2 is on cell (2,3), n_2 is on cell (3,1)) (Figure 7.3 middle)
- $(9, 12, 24) \mapsto$ (n_3 is on cell (1,3),n_3 is on cell (2,1), n_3 is on cell (3,2)) (Figure 7.3 right)

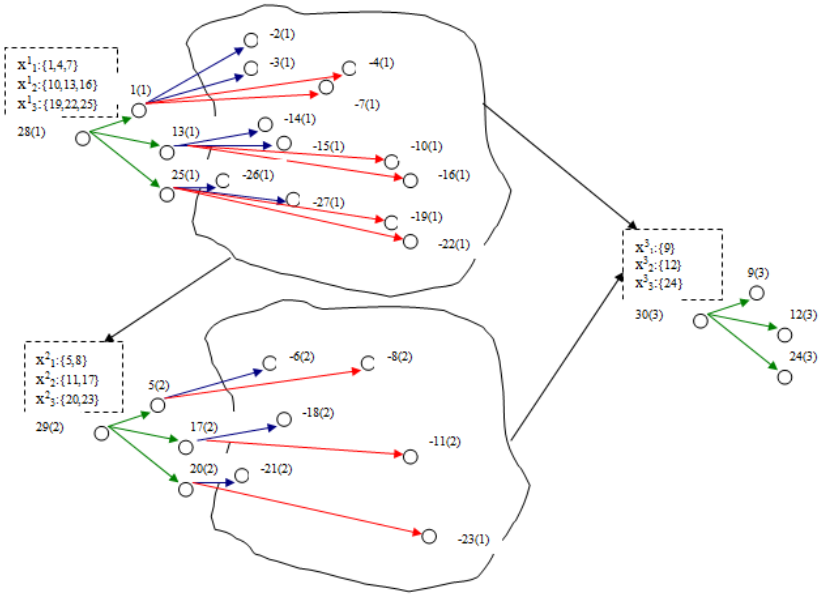

Figure 7.2: Implication graph for 3x3 Latin square for the SAT(gc) encoding.

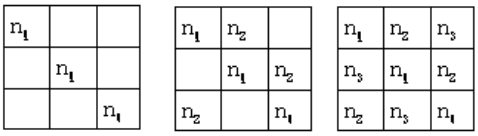

Figure 7.3: Solving 3X3 Latin square of by prototype of SAT(gc).

7.2.4 Performance analysis

For the Latin square problem SATCP runs faster than ZCHAFF. To encode Latin square in SAT, total number of clauses required are $O(n^4)$ [27]. The Π_{latin} instance has $O(n^3)$ clauses. Thus, by the use of global constraints, one can write more compact representations in SAT(gc) than in SAT. This plus the efficient propagators for $allDiff$ global constraint as implemented in GECODE seems to be the main reasons for the better performance of our implementation of SATCP.

A call to the global constraint assigns a particular number to n different cells of the square. Each call is solving a part of the problem, which could be solved after searching through $O(n^4) - O(n^3)/n$ SAT search space.

7.3 The Magic Square Problem

A normal magic square can be defined as follows:

> *A magic square of order n is an arrangement of n^2 numbers, usually distinct integers, in a square, such that the sum of n numbers in rows, columns, and in*

both diagonals are equal to a constant number.

A normal magic square contains the integers from 1 to n^2. The constant sum over rows, columns and diagonals is called the *magic sum*, which is known to be $n(n^2 + 1)/2$. The problem of finding a normal magic square is called the *normal magic square problem.*

There are variations of magic square, based on different conditions to be satisfied. But in this thesis, magic square refers to normal magic square.

Figure 7.4 shows a magic square of order 3. The magic sum in this case is 15.

4	9	2
3	5	7
8	1	6

Figure 7.4: A normal magic square of order 3.

7.3.1 Encoding of magic square in SAT(gc)

As mentioned in Chapter 5, due to the Boolean nature of SAT, the size of encoding of numerical constraints in SAT can be exponential. As the problem of normal magic square involves numerical constraints, to the best of our knowledge, no direct SAT encoding for the problem is described in the SAT literature.

Here we will present two different encodings of normal magic square problem in SAT(gc), their experimental results and some analysis. The first encoding encodes the normal magic square problem as a monolithic constraint. We call this encoding *monolithic* because the encoding consists of a single CSP constraint composed of a number of global constraints. Thus, to solve the magic square problem in SAT(gc) becomes a single call to GECODE. This will be contrasted with the second encoding where the monolithic constraint is decomposed into a collection of global constraints and SAT clauses. We call this encoding *decomposed*.

7.3.2 Monolithic SAT(gc) encoding

In this approach of encoding in SAT(gc), we encode the whole magic square problem by a monolithic constraint *magic_square* as follows:

$$magic\_square(x_{11}, x_{12} \dots x_{ij} \dots x_{nn})$$

where $Dom(x_{11}) = Dom(x_{12}) = \cdots = Dom(x_{nn}) = \{1 \dots n^2\}$. The assignment $x_{ij} = k$ represents that the number k is placed at the j^{th} column of i^{th} row (where $1 \leq i, j \leq n$ and $1 \leq k \leq n^2$).

For encoding the monolithic magic square problem of order n in SAT(gc), we need n^4 propositional variable p_{ijk}, where p_{ijk} means that, the number k is assigned in to the cell (i, j), with $1 \leq i, j \leq n$ and $1 \leq k \leq n^2$. These propositional variables are the value variables which correspond to the domain values of the CSP variables $x_{11}, x_{12} \dots x_{ij} \dots x_{nn}$.

In GECODE, we implement $magic\_square(x_{11}, x_{12} \dots x_{ij} \dots x_{nn})$ by using a *allDiff* global constraint on the variables $x_{11} \dots x_{nn}$ and $2n+2$ *sum* global constraints, where n sum global constraints map the sum constraint across the n rows, the other n sum global constraints map the sum constraints across the n columns, and 2 addition sum global constants are for constraints are for mapping the sum constraint across left to right diagonal and right to left diagonal, respectively. Table 7.5 shows the monolithic SAT(gc) instance for n by n normal magic square and its GECODE implementation. Notice that, as the whole normal magic square problem is encoded using one monolithic constraint, all the constraints, CSP variables on the right column of Table 7.5 are put into a single constraint store.

Monolithic SAT(gc) Encoding of magic square	GECODE implementation
$magic\_square(x_{11}, \ldots, x_{nn})$ 0	(Mapping distinct symbols constraint) $allDiff(x_{11}, \ldots, x_{nn})$ - (Mapping sum across rows) $sum(x_{11}, \ldots, x_{1n})$ $sum(x_{21}, \ldots, x_{2n})$ $\vdots$ $sum(x_{n1}, \ldots, x_{nn})$ - (Mapping sum across columns) $sum(x_{11}, \ldots, x_{n1})$ $sum(x_{12}, \ldots, x_{n2})$ $\vdots$ $sum(x_{1n}, \ldots, x_{nn})$ - (Mapping sum across diagonals) $sum(x_{11}, \ldots, x_{nn})$ $sum(x_{n1}, \ldots, x_{1n})$

Table 7.5: SAT(gc) encoding of magic square with a monolithic constraint.

7.3.3 Experiments with monolithic encoding

Table 7.6 shows the experimental results for the SAT(gc) encoding of normal magic square with monolithic constraint. By this encoding we are able to solve the magic square problem up to size 7 by 7 by SATCP within 15 minutes time.

n	*time in (seconds)*
3	0.00058
4	0.02
5	0.61
6	0.04
7	3.92
8	—

Table 7.6: Experiments with magic square under monolithic constraint .

7.3.4 Comparison to SAT with weight constraints

It is worth commenting on a related experiment on *answer set programming* (ASP) [4, 16, 17], for the same magic square problem. An efficient solver in ASP is CLASP,[1] which can be applied as a SAT solver; actually, it is considered a top-notch SAT solver, as it won the SAT/UNSAT category in the 2009 SAT Competition. One difference between CLASP and other sate-of-the-art SAT solvers is that CLASP supports weight constraints which can be used to encode aggregates and numerical constraints [23]. In fact, almost all ASP systems support aggregates of certain kind [25, 38, 47]. However, weight constraints in ASP are not implemented in the same way as global constraints in CP.

Recently in [50], the authors considered adding (evaluable) function in weight constraint programs, and translated such a weight constraint program to an equivalent instance of

[1] http://www.cs.uni-potsdam.de/clasp/

CSP. The solution of that CSP instance corresponds exactly to answer sets of the original program. A prototype system extending FASP[2] is implemented for the computation of answer set for weight logic programs with functions. With that system, the authors carried out some experiments with benchmarks including the normal magic square problem. They show two encodings of the normal magic square problem in weight constraint programs. In one version, to encode the constraint "no number is repeated in any two cells of the square", they use a $allDiff$ global constraint. In another version, they used equivalent ASP rules to encode the all-different constraint. In general, the version that was encoded with the all-different global constraint runs faster than the other one. For the same problem, they also compared the running time of CLASP with FASP. With the version encoded with all-different global constraint, FASP runs much faster, sometimes orders of magnitude faster, than CLASP. For example, FASP solves the normal magic square problem (encoded with $allDiff$) of order 7 in 1.63 seconds, while CLASP solves the same instance (encoded with aggregates) in 450.58 seconds.

The experimental results with the monolithic encoding of normal magic square running on SATCP confirms the results of [50], as with this encoding SATCP runs much faster than CLASP. It also demonstrates that, the propagators of global constraints from CSP are more efficient than the aggregates from the logic programming framework. Finally, one methodology for SAT(gc) is that one can always define a CSP constraint to solve part of a given program, and in our implementation SATCP such a user-defined CSP constraint can be invoked like a global constraint.

7.3.5 Decomposed SAT(gc) encoding

In this encoding of magic square, we encode the normal magic square problem in SAT(gc) by decomposing the *magic_square* constraint (described in the previous subsection) into CNF clauses and global constraints. To encode the constraint that, "all the numbers in the square must be distinct", we use CNF encoding. And to encode the constraints that "the total sum across a column, row or diagonal must be equal to the magic sum", we use the *sum* global constraints, i.e., to encode each of the sum constraints for each of the rows, columns and diagonals we use a *sum* global constraint.

Like the monolithic SAT(gc) encoding, the decomposed SAT(gc) encoding of the normal magic square problem of order n requires n^4 propositional variable, where a variable p_{ijk} means that, the number k is assigned in to the cell (i, j), with $1 \leq i, j \leq n$ and $1 \leq k \leq n^2$.

The decomposed SAT(gc) encoding contains both CNF clauses and global constraints, as follows:

- No number is repeated in any two cells of the square:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \bigwedge_{k=1}^{n^2} \bigwedge_{l=k+1}^{n^2} (\neg p_{ijk} \vee \neg p_{ijl})$$

- The sum of the numbers in any row equals the magic sum M:

$$\bigwedge_{r=1}^{n} sum(x_{r1}, x_{r2}, \ldots, x_{rn}, M),$$

  where $Dom(x_{r1}) = Dom(x_{r2}) = \cdots = Dom(x_{rn}) = \{1, \ldots, n^2\}$. The assignment $x_{rj} = k$ asserts that, in order to satisfy the *sum* constraint across row r the number k must be placed on j^{th} column of the r^{th} row of the square.

[2] http://www.cse.ust.hk/fasp/

- The sum of the numbers in any column equals the magic sum M:

$$\bigwedge_{c=1}^{n} sum(x_{1c}, x_{2c}, \ldots x_{nc}, M)$$

where $Dom(x_{1c}) = Dom(x_{2c}) = \cdots = Dom(x_{nc}) = \{1, \ldots, n^2\}$. The assignment $x_{ic} = k$ asserts that, in order to satisfy the *sum* constraint across column c the number k must be placed on c^{th} column of the i^{th} row of the square.

- The sum of the numbers in the left to right diagonal equals the magic sum M:

$$sum(\bigcup_{r=1}^{n}\bigcup_{s=1}^{n} x_{rs}, M)$$

where $r = s$ and $Dom(x_{rs}) = \{1, \ldots, n^2\}$. The assignment $x_{rs} = k$ asserts that, in order to satisfy the *sum* constraint across left to right diagonal the number k must be placed on s^{th} column of the r^{th} row of the square.

- The sum of the numbers in the right to left diagonal is equal to the magic sum M:

$$sum(\bigcup_{r=1}^{n}\bigcup_{s=n}^{1} x_{rs}, M)$$

where $r = s$ and $Dom(x_{rs}) = \{1, \ldots, n^2\}$. The assignment $x_{rs} = k$ asserts that, in order to satisfy the *sum* constraint across right to left diagonal the number k must be placed on s^{th} column of the r^{th} row of the square.

7.3.6 Experiments with decomposed encoding and analysis

SATCP solved Π_{magic} of order 3 in 3.17 sec. But for Π_{magic} instances of higher order, it failed to generate a solution within 15 minutes.

In the decomposed encoding of normal magic square, we have used separate *sum* global constraints for modeling the sum constraints across each of the rows, columns and diagonals. Thus, at the GECODE end, we model each of these global constraints as separate CSPs. When SATCP executes on a decomposed magic square instance, these global constraints are put into separate constraint stores and are treated as independent and unrelated constraints. But, notice that these *sum* global constraints are related, as they are sharing CSP variables with each other. Assigning a CSP variable x to a value by any of the global constraints effects all CSP variables of the global constraints those have x on their scope. Though we are performing DP operation, after an assignment on a CSP variable x is made by a global constraint, the propagator for other related global constraint (those have x on their scope) are not getting notified of that assignment, as the *sum* global constraints are put on different constraint store. Therefore, the result of DP operation cannot be propagated to other related CSP variables.

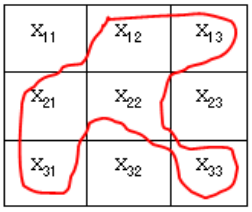

Figure 7.5: Dependency of variables in magic square problem

For the decomposed encoding, $sum_{r1}(x_{11}, x_{12}, x_{13})$ is related with $sum_{c1}(x_{11}, x_{21}, x_{31})$, $sum_{c2}(x_{12}, x_{22}, x_{32})$, $sum_{c3}(x_{13}, x_{23}, x_{33})$, $sum_{diag_{lr}}(x_{11}, x_{22}, x_{33})$ and $sum_{diag_{rl}}(x_{13}, x_{22}, x_{31})$ as $sum_{r1}(x_{11}, x_{12}, x_{13})$ is sharing exactly one CSP variable with each of them. The red area in Figure 7.5 shows which variables are effected when x_{11} is assigned to a value by sum_{r1}. During the execution of SATCP on this decomposed magic square instance, when x_{11} is assigned to a value by sum_{r1}, that assignment is only propagated on $Dom(x_{12})$ and $Dom(x_{13})$ (by the propagator of sum_{r1}). This assignment of x_{11} cannot be propagated on $Dom(x_{22})$ and $Dom(x_{33})$, $Dom(x_{21})$ and $Dom(x_{31})$, because the propagators for $sum_{diag_{lr}}$ and sum_{c1} are not notified about this assignment of x_{11} as they locate on different constraint stores.

The effect of putting related global constraints and variables in separate global constraint is drastic. The invocation to GECODE for solving a *sum* constraint returns solution without propagating the solution into other constraint stores, where the related *sum* constraints are stored. It results in an exponential number of enumerations of CSP solutions via a series of backtracking between the related global constraints. The problem of solving the normal magic square problem with decomposed encoding becomes the problem of enumerating solutions of global constraints. For example, the magic square of order 3 is solved after 711 enumerations of solutions of the 8 *sum* global constraints. With the increment of the problem size, the number of enumerations also increases exponentially. Aside this, as the problem size grows, more time is needed to search for an alternative solution for a global constraint. For these reasons, our prototype implementation of SAT(gc) provides no solution within a reasonable amount of time for the decomposed SAT(gc) encoding for the orders greater than 3.

7.4 Planning by SAT Solvers

SAT solvers can be used to solve the planning problem competitively. Usually domain independent planning is specified in a special language, called STRIPS [43]. A STRIPS problem specification includes a statement of the initial condition, possible actions and goals to achieve. The actions are specified in terms of their preconditions and effects. Before solving a planning problem specified in STRIPS using a SAT solver, a STRIPS specification is converted into an equivalent SAT instance. If the equivalent SAT planning instance is satisfiable, then the specified goal state can be reached from the initial state, by performing a series of actions. The automated planning community has developed a few SAT based planners, among which the most well-known SAT-based planner is SatPlan [22], which takes a STRIPS problem specification, converts it to a SAT formula, determines the satisfiability and extracts the plan from a propositional variable assignment. In our experiments, we have used SatPlan to obtain the converted SAT formulas from the STRIPS specifications of selected planning problems.

SatPlan encodes the planning problem by generating a planning graph. The planning graph is a layered graph[3] in which the layers of vertices form an alternating sequence of literals and operators:

$$(L1, O1, L2, O2, L3, O3, \ldots, L_k, O_k, L_{k+1})$$

The edges are defined as follows. To each operator $o_i \in O_i$, a directed edge is made from each $l_i \in L_i$ that is a precondition of o_i. To each literal $l_i \in L_i$, an edge is made from each operator $o_{i-1} \in O_{i-1}$ that has l_i as an effect [24].

The planning graph generated by SatPlan contains all the goal literals at the k^{th} layer (highest layer). SatPlan then generates clauses that asserts the following facts [22]:

- Goal state at layer k and initial state at layer 0.

[3]A layered graph is a graph that has its vertices partitioned into a sequence of layers, and its edges are only permitted to connect vertices between successive layers

- If a fluent holds at layer k, the disjunction of actions that have that fluent as an effect hold at layer $k-1$.
- Actions at each level imply their preconditions.
- Actions with conflicting effects or preconditions are mutually exclusive and encoded as negative binary clauses.
- Fluents that are inferred to be mutually exclusive are encoded as negative binary clauses.

7.5 Ferry Planning with Numerical Constraints

In this section, we will describe how we have solved the ferry planning problem with numerical constraints in SATCP, followed by some experimental results.

7.5.1 Problem specification

The original ferry planning problem can be described as the follows:

> *There are a number of cargoes and a ferry, each at a location at the beginning. The ferry can transport one cargo at a time. The goal is to transport all cargoes to their destinations.*

Our prototype system of SAT(gc) can be used to solve an enhanced version of the ferry planning problem with numerical constraints, which would present a challenge to the state-of-the-art SAT-based planners such as SatPlan. For example, for the ferry problem specified above, we can enhance the usability of the ferry, by making it capable of transporting multiple cargoes at the same time, with the consideration of their weights and volumes and the weight and volume carrying capacity of the ferry. In this thesis, to demonstrate the usability of SAT(gc), we choose to deal with a simple ferry planning problem, as follows:

> *There are n cargoes and a ferry at an initial location. The ferry can transport multiple cargoes at the same time, but the number of cargoes it can transport is limited to its weight and volume carrying capacity. The goal is to transport all the cargoes to their destinations, by taking a group of cargoes, whose weight and volume does not exceed the weight and volume carrying capacity of the ferry, at each turn of the ferry.*

To further simplify the problem, we assume the destinations are unique. It is clear that the ability to solve this problem by a SAT-based planner can be extended to the general case of planning with multiple original locations for cargoes and the ferry and different destinations of cargoes.

```
(define (problem FERRY_PROBLEM)
    (:domain FERRY3)
    (:objects C1, C2, C3, LOC1, LOC2)
    (:INIT (CARGO C1) (CARGO C2) (CARGO C3) (LOCATION LOC1)
    (LOCATION LOC2) (FERRY-AT LOC1) (ORIGIN C1 LOC1)
    (ORIGIN C2 LOC1) (ORIGIN C3 LOC1) (DESTINATION C1 LOC2)
    (NEXT-TO LOC1 LOC2) (DESTINATION C2 LOC2)
    (DESTINATION C3 LOC2))
    (:goal (AND (SERVED C1) (SERVED C2) (SERVED C3)
    (FERRY-AT LOC1))))
```

```
(:action board
    :parameters (?loc-source ?c)
    :precondition(and (location ?loc-source) (cargo ?c) (ferry-at ?loc-source)
                (origin ?c ?loc-source))
    :effect(and
        (boarded ?c)
        (in-ferry ?c ?loc-source)))
```

```
(:action unboard
    :parameters (?loc-destination ?c)
    :precondition (and (location ?loc-destination) (cargo ?c)
    (destination ?c ?loc-destination) (boarded ?c) (ferry-at ?loc-destination))
    :effect (and
        (served ?c)
        (not (boarded ?c))
        (not (in-ferry ?c ?loc-destination) )))
```

```
(:action move-to
    :parameters (?x ?y)
    :precondition (and (location ?x) (location ?y) (next-to ?x ?y) (ferry-at ?x))
    :effect( and
        (ferry-at ?y)
        ( not (ferry-at ?x))))
```

```
(:action move-back
    :parameters (?x ?y)
    :precondition (and (location ?x) (location ?y) (next-to ?y ?x) (ferry-at ?x))
    :effect( and
        (ferry-at ?y)
        ( not (ferry-at ?x))) ) )
```

Table 7.7: STRIPS specification for the ferry problem.

We use a monolithic numerical constraint (composed of three sets of sum global constraints depicted in Table 7.8) to create the groups of the cargoes to be transported for each turn of the ferry. The solution of this monolithic constraint directs the planning by determining which group of cargoes to be transported in which turn. The details of the encoding are discussed in the next subsection.

7.5.2 Encoding in SAT(gc)

To encode the ferry planning problem with numerical constraints in SAT(gc), first we generate a CNF problem instance which is encoded from a STRIPS specification such as the one described in Table 7.7.

For a STRIPS specification such as the one described in Table 7.7, SatPlan generates a CNF problem instance which states that, the ferry boards all the cargoes available at

the source location, moves to the destination location and unboards all the cargoes and then moves back to the source location. As we intend to give ferry the ability to generate a plan based on the weights and volumes of the cargoes and enables the ferry to make multiple moves between source and destinations, we need to extend this SAT instance. We use the phrase *a turn* to describe each movement of ferry between the source location and destination location. So, in each turn the ferry transports a subset of cargoes from the set of n cargoes, which satisfies the weight and volume constraints. To make a SAT(gc) instance with numerical constraints from this SAT instance, we apply the following steps:

- A group of cargoes are transported in a turn of the ferry. In the SAT instance (such as those, which generated from STRIPS of Table 7.7), we only encode one turn of the ferry. If there are m groups of cargoes, then that SAT instance is required to be replicated m times (where each of the replicated part serves as one group of cargoes). We extend that SAT instance by replicating it $n-1$ times. Extending that SAT instance by replicating it $n-1$ times guarantees that, all of the cargoes are served for all possible number of groups. Of course, this assumes that each single cargo fits the ferry.

- The grouping of cargoes for each turn is encoded by a monolithic constraint

  $$group(x_{11}, x_{21} \ldots x_{n1} \ldots x_{ij} \ldots x_{nn})$$

  where $Dom(x_{ij}) = \{0, 1\}$. The assignment $x_{ij} = 1$ asserts that the cargo i is served at turn j and the assignment $x_{ij} = 0$ asserts that the cargo i is not served at turn j.

  We introduce n^2 propositional variables to form the value variables[4] which correspond to the domain values of these CSP variables. Let us write these value variables by r_{ij} (where $1 \leq i, j \leq n$). We assign r_{ij} to true when x_{ij} is assigned to value 1 and assign r_{ij} to false when x_{ij} is assigned to 0.

- In the j^{th} part of the extended CNF instance, we add $\neg r_{ij}$ to the clause which contains the unboarding literal for i^{th} cargo at j^{th} turn. We also add another clause r_{ij} $\neg u_{ij}$ (u_{ij} represents the unboarding of the i^{th} cargo at the goal layer of the j^{th} turn) to disable the generation of plan for serving i^{th} cargo at the j^{th} turn, in the case when solution to *group* contains the assignment $x_{ij} = 0$.

- The GECODE implementation of this monolithic constraint *group* is described in Table 7.8. Three different sets of *sum* global constraints are used. The first set models the weight capacity constraint, the second set models the volume capacity constraints and the third set models the constraint that one cargo is served exactly once. W_f and V_f denote the maximum weight and volume respectively that the ferry can carry in one turn. *weights* and *volumes* are two arrays which hold the weights and volumes of the cargoes $c_1, c_2 \ldots, c_{n-1}, c_n$ respectively.

  The solution of this monolithic constraint makes groups of cargoes, where each group i is taken at each turn (by the i^{th} part of the extension).

[4]Note that the total number of domain values in the CSP variables $x_{11} \ldots x_{ij} \ldots x_{nn}$ are $2n^2$. But as the domains are binary, we can represent each of these domain values by using one propositional variable.

group constraint	GECODE *implementation*
$group(x_{11}, \ldots, x_{ij}, \ldots x_{nn})$ 0	(Constraining weights for each turn) $sum(x_{11}, \ldots, x_{n1}, weights, W_f)$ $sum(x_{12}, \ldots, x_{n2}, weights, W_f)$ $\vdots$ $sum(x_{1n}, \ldots, x_{nn}, weights, W_f)$ (Constraining volumes for each turn) $sum(x_{11}, \ldots, x_{n1}, volumes, V_f)$ $sum(x_{12}, \ldots, x_{n2}, volumes, V_f)$ $\vdots$ $sum(x_{1n}, \ldots, x_{nn}, volumes, V_f)$ (Constraining that one cargo is served exactly once) $sum(x_{11}, \ldots, x_{1n}, 1)$ $sum(x_{12}, \ldots, x_{2n}, 1)$ $\vdots$ $sum(x_{n1}, \ldots, x_{nn}, 1)$

Table 7.8: GECODE model for grouping cargoes.

7.5.3 Solving ferry planning with numerical constraints

At the very beginning, *gc_heuristic* chooses the the literal corresponding to the monolithic constraint *group*. GECODE is invoked to solve the *group* constraint. As the solution of *group* is returned, if x_{ij} is assigned to 1 (the i^{th} cargo is served in the j^{th} turn), then it creates an implication r_{ij} and by the assignment of r_{ij} to 1 (Figure 7.6), the unboarding clause for the i^{th} cargo at the j^{th} turn becomes a unit clause.

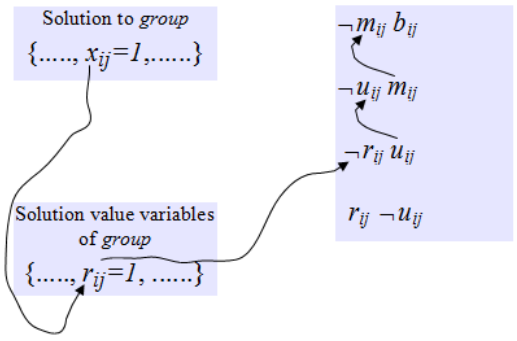

Figure 7.6: Determination of the plan for serving i^{th} cargo in j^{th} turn

At this point, the planning for serving i^{th} cargo at j^{th} turn is generated by the chaining of actions (via unit propagation) from the current layer back to the initial layer. The selection of the i^{th} cargo at this j^{th} turn is propagated back up to the initial layer for the j^{th} turn. As shown in Figure 7.6, at the goal layer of the j^{th} turn, the unboarding literal u_{ij} is assigned to true. This creates unit propagation and as a result, at the middle layer, move-to action literal, m_{ij}, is assigned to true (as the clause becomes unit), which enables the move-to action for cargo i at turn j. And during the BCP of m_{ij} at the initial layer of turn j, boarding action literal b_{ij} is assigned to true. This assignment of b_{ij} enables the boarding events of the i^{th} cargo at the j^{th} turn.

In case x_{ij} is assigned 0 (i^{th} cargo is not served in j^{th} turn), r_{ij} is assigned to false. By the last clause depicted in Figure 7.6, this assignment of r_{ij} prohibits the generation of the plan to serve the i^{th} cargo at the j^{th} turn.

In this way, a plan for serving the cargoes is generated.

7.5.4 Experimental results

We have run three instances with different numbers of cargoes for this ferry planning problem in SATCP. We are able to solve those problems under 0.05 seconds. The results are shown in Table 7.9.

n	*weights* $(w_1, w_2 \dots w_n)$	*volumes* $(v_1, v_2 \dots v_n)$	W_f	V_f	*served cargoes* (by turn)	*time* (in sec)
3	(23,45,18)	(150,180,100)	50	250	turn1:c1,c3 turn2:c2 turn3:	0.01
6	(23,45,53, 51,53,18)	(150,180,120, 200,190,200)	100	400	turn1:c1,c2 turn2:c3,c6 turn3:c4 turn4:c5 turn5: turn6:	0.03
9	(12,25,16, 15,18,25, 25,12,36)	(150,180,120, 100,190,170, 160,150,140)	100	600	turn1:c1,c2,c3,c4 turn2:c5,c6,c7 turn3:c8,c9 turn4: turn5: turn6: turn7: turn8: turn9:	0.04

Table 7.9: Experiments with the ferry problem.

7.6 Planning of Block Stacking with Numerical Constraints

The planning problem of block stacking with numerical constraints can be described as follows:

> In a table, there are n ($n > 1$) stacks of blocks, each having m_i number of blocks, where $1 \leq i \leq n$. Let $block_{ij}$ be the j^{th} ($1 \leq j \leq m_i$) block of i^{th} ($1 \leq i \leq n$) stack. In the initial configuration in every stack i the first block $block_{i1}$ is placed on the table. If $m_i > 1$, then $block_{ij}$ is placed on $block_{i(j-1)}$. Every block $block_{ij}$ has a weight w_{ij}. We have to generate a plan of actions for building a new stack of blocks by taking exactly one block from each of the initial n stacks in such a way that
>
> - The total weight of the selected blocks should be equal to a certain total weight W_{max}. That is, if block $j^1, j^2 \dots j^n$ are selected respectively from stacks $1, 2 \dots n$, then $w_{1j^1} + w_{2j^2} + \dots + w_{ij^i} + \dots + w_{nj^n} = W_{max}$ (Constraint1).
> - Block selected from the i^{th} stack must be placed over block selected from the $(i-1)^{th}$ stack (Constraint2).

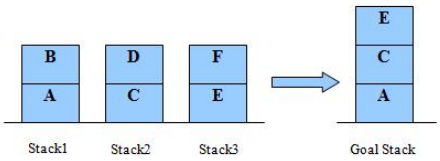

Figure 7.7: Example of block stacking with numerical constraint.

Figure 7.7 shows an example of block stacking with 3 stacks. The left side of the arrow depicts the initial configuration of the stacks. In stack1, block B is placed on block A, in stack2, block D is placed on block C and in stack3, block F is placed on block E. Suppose the weights of blocks A, B , C, D, E and F are w_A (5 kg), w_B (4 kg), w_C (3 kg), w_D (6 kg), w_E (7 kg) and w_F (3 kg) respectively. W_{max} is 15 kg. This planning problem requires to provide a solution in such a way that blocks selected form stack1, stack2 and stack3 must be equal to 15kg (W_{max}). By this constraint, three blocks A, C and E are selected from stack1, stack2 and stack3, respectively, as $w_A + w_C + w_E = (5+3+7)kg = 15kg$ (satisfying constraint 1) and E (from stack3) is placed over C (from stack2) and C is placed over A (from stack1) (satisfying constraint 2).

7.6.1 Encoding in SAT(gc)

There are two different parts involved in solving this block stacking planning problem with numerical constraints. First, we need to determine which blocks to select to form the goal stack by solving the weight constraint. Then we need to gather these selected blocks and make the goal stack in such a way that block j^i is placed on block $j^{(i-1)}$. To encode such a problem in SAT(gc),

1. we first need to generate the action-based CNF problem instance from a STRIPS specification as given in Table 7.10.

   In the action based CNF encoding of planning problem of block stacking such as the one generated from STRIPS specification in Table 7.10, actions are encoded as Boolean variables. For each action a and layer l, we have a Boolean variable a^l representing that action a is enabled at the layer l. Every a^l has a corresponding *not* literal. The literal not_a^l represents that action a is not enabled at layer l. Two ideas are central to this encoding:

   - If the action a is enabled at the layer l, then the other actions which generate the preconditions of action a, must be enabled in the layers $l' < l$. If action a needs to be enabled for achieving the goal state, then action a must be enabled in any of the layers. For a given layer l, all the necessary actions to enable action a at the layer l is chained from layer $l-1$ to the layer initial layer by using the literal a^l.
   - If the action a needs to be enabled to reach the goal state and it cannot be enabled at the layer l, then action a must be enabled in any other layers $l' < l$. This requirement is encoded from layer l to the initial layer by using not_a^l via chaining.

```
(define (problem block-stacking)
    (:domain BLOCKS)
    (:objects A B C D E F)
    (:INIT (ON B A) (ON D C) (ON F E) (ONTABLE A) (ONTABLE C) (ONTABLE E)
        (CLEAR B) (CLEAR D) (CLEAR F) (HANDEMPTY))
    (:goal (AND (ON C A) (ON E C) (CLEAR E) (ONTABLE A))))
```

```
(:action pick-up
    :parameters (?x)
    :precondition (and (clear ?x) (ontable ?x) (handempty))
    :effect
    (and (not (ontable ?x))
    (not (clear ?x))
    (not (handempty))
    (holding ?x)))
```

```
(:action put-down
    :parameters (?x)
    :precondition (holding ?x)
    :effect
    (and (not (holding ?x))
    (clear ?x)
    (handempty)
    (ontable ?x)))
```

```
(:action stack
    :parameters (?x ?y)
    :precondition (and (holding ?x) (clear ?y))
    :effect
    (and (not (holding ?x))
    (not (clear ?y))
    (clear ?x)
    (handempty)
    (on ?x ?y)))
```

```
(:action unstack
    :parameters (?x ?y)
    :precondition (and (on ?x ?y) (clear ?x) (handempty))
    :effect
    (and (holding ?x)
    (clear ?y)
    (not (clear ?x))
    (not (handempty))
    (not (on ?x ?y)))))
```

Table 7.10: STRIPS specification for block stacking.

To illustrate how to encode the above two requirements, we assume a part of the initial state and a part of the goal state of a planning problem of block stacking. In the initial configuration, at the top of i'^{th} stack, suppose $block_{i'j'}$ is placed on $block_{i'j''}$ and at the top of i^{th} stack, $block_{ij}$ is placed on the $block_{ij0}$. In the goal state, as part of the goal stack, $block_{i'j'}$ is placed on $block_{ij}$, where $i' = i+1$. Let $ST^l_{ij\_i'j'}$ represent "stack the block $block_{i'j'}$ on $block_{ij}$ at layer l", and $U^l_{i'j''\_i'j'}$ represent that "unstack $block_{i'j'}$, which resides on $block_{i'j''}$ at layer l". To assert the goal that,"$block_{i'j'}$ is placed on $block_{ij}$", the CNF encoding has the following clauses:

(a) Stacking of $block_{i'j'}$ on $block_{ij}$ at the goal layer or any of the previous layers.

$$ST^{l\_goal}_{ij\_i'j'} \vee not\_ST^{l\_goal}_{ij\_i'j'}$$
$$\neg not\_ST^{l\_goal}_{ij\_i'j'} \vee ST^{l\_goal-1}_{ij\_i'j'} \vee not\_ST^{l\_goal-1}_{ij\_i'j'}$$
$$\neg not\_ST^{l\_goal-1}_{ij\_i'j'} \vee ST^{l\_goal-2}_{ij\_i'j'} \vee not\_ST^{l\_goal-2}_{ij\_i'j'}$$
$$\vdots$$
$$\neg not\_ST^{2}_{ij\_i'j'} \vee ST^{1}_{ij\_i'j'}$$

(b) If the j'^{th} block from the i'^{th} stack is stacked in the j^{th} block from the i^{th} stack at the layer l, then it must be unstacked from i'^{th} stack at any of the previous layer.

$$\neg ST^{l}_{ij\_i'j'} \vee U^{l-1}_{i'j''\_i'j'} \vee not\_U^{l-1}_{i'j''\_i'j'}$$
$$\neg not\_U^{l-1}_{i'j''\_i'j'} \vee U^{l-2}_{i'j''\_i'j'} \vee not\_U^{l-2}_{i'j''\_i'j'}$$
$$\vdots$$

(c) If $block_{i'j'}$ is stacked on $block_{ij}$ at the layer l, then it is not stacked in any of the previous layers.

$$\neg ST^{l}_{ij\_i'j'} \vee \neg not\_ST^{l}_{ij\_i'j'}$$

The above set of clauses pertains to $block_{i'j'}$. The full CNF encoding has similar clauses for each of the blocks which form the goal stack. In the above encoding other actions (like pick-up and put-down) are not shown, but they are chained in similar manner.

2. After generating the SAT instance as above, we need to introduce a *sum* global constraint into it. The question of which block j^i to select from the i^{th} stack to build the goal stack is determined by the solution of the *sum* global constraint. So, the solution of the global constraint is used to determine the part of the initial condition and which, in turn, determines the goal state. The *sum* global constraint is modeled as follows:

$$sum(stack_1, stack_2, \ldots, stack_n, W_{max})$$

where weights $stack_1, stack_2, \ldots, stack_n$ are the CSP variables and $Dom(stack_i) = \{w_{i1}, w_{i2}, \ldots w_{ij}, \ldots w_{im_i}\}$. The assignment $stack_i = w_{ij}$ asserts that the j^{th} block form the i^{th} stack is selected for building the goal stack. We also introduce $n*m$ (where $m = \bigwedge_{i=1}^{n} m_i$) propositional variables, which are the value variables corresponding to the domain values of $stack_i$. We define $Val\_Var(stack_i) = \{r_{i1}, r_{i2}, \ldots r_{ij}, \ldots r_{im_i}\}$. In the CSP solution of the *sum* global constraint, if $stack_i$ is assigned to w_{ij}, then r_{ij} is assigned to true, otherwise r_{ij} is assigned to false.

3. These value variables are introduced to the SAT instance, in order to associate the selection literals with stacking action literals (to achieve the goal stack) at the goal layer of that CNF encoding. For every pair of blocks, $block_{ij}$ and $block_{i'j'}$ (where $i' = i + 1$), we add

$$\neg r_{ij} \vee \neg r_{i'j'}$$

to stacking action (goal action) clauses (at goal layer) as follows: action of the $j'th$ block from $i'th$ stack into j^{th} blocks of i^{th} stack

$$(i)\ \neg r_{ij} \vee \neg r_{i'j'} \vee ST^{l\_goal}_{ij\_i'j'} \vee not\_ST^{l\_goal}_{ij\_i'j'}$$

Clause set (i) ensures that if $block_{ij}$ and $block_{i'j'}$ are selected, then $block_{i'j'}$ is stacked on $block_{ij}$ at the goal layer or any of the previous layers.

In addition, we must ensure that no blocks, which are not selected by the sum constraint, are placed in the goal stack. If the block $block_{1j}$ (where $j > 1$) is selected from stack1, $block_{1j}$ has at least one block placed under $block_{1j}$ which is not selected. We must exclude all the blocks that are placed under $block_{1j}$ from being a part of the goal stack. To do this, we need to add some more clauses. We add the following clauses for each of the blocks $block_{1j}$ $(j > 1)$ as follows:

$$(ii) \quad -r_{1j} \vee PD_{1j}^{l\_goal-1} \vee \cdots \vee PD_{1j}^{1}$$

where PD_{1j}^{l} represents that $block_{1j}$ is put on the table at layer l. Clause set (ii) ensures that, if $block_{1j}$ is selected for the building the goal stack, which is not currently residing on the table (i.e residing on other block), it must be unstacked from stack1 and put-down on the table.

This encoding ensures that, whenever j'^{th} block of i'^{th} stack and j^{th} block of i^{th} stack satisfies the *sum* constraint, a plan for putting $block_{i'j'}$ on $block_{ij}$ (where $i' = i+1$) to form the goal stack is generated.

In the following subsection, we will describe the solving process of this planning problem of block stacking by assuming the example shown in Figure 7.7.

7.6.2 Solving block stacking with numerical constraints

To solve the planning problem of block stacking shown in Figure 7.7 by SATCP, first we need to generate a SAT instance of the STRIPS specification shown on Table 7.10. This SAT planning instance is converted into a SAT(gc) planning instance by introducing the global constraint

$$sum(stack_1, stack_2, stack_3, W_{max})$$

where

$$Dom(stack_1) = \{w_A, w_B\}, Dom(stack_2) = \{w_C, w_D\}, Dom(stack_3) = \{w_E, w_F\}.$$

Let

$$Val\_Var(stack_1) = \{r_A, r_B\}, Dom(stack_2) = \{r_C, r_D\}, Dom(stack_3) = \{r_E, r_F\}$$

where r_A corresponds to w_A, r_B corresponds to w_B, and so on.

In this SAT(gc) planning instance the solution of this *sum* global constraint determines which blocks must be taken from each of the stacks to build the goal stack. So, as described in the previous section in the goal layer of the SAT(gc) instance, we have (now each line represents the disjunction of the literals appearing in it)

$$\begin{array}{llll}
(c1)\ \neg r_E & \neg r_C & ST_{C_E}^{l_goal} & not_ST_{C_E}^{l_goal} \\
(c2)\ \neg r_E & \neg r_D & ST_{D_E}^{l_goal} & not_ST_{D_E}^{l_goal} \\
\mathbf{(c3)\ \neg r_F} & \mathbf{\neg r_C} & \mathbf{ST_{C_F}^{l_goal}} & \mathbf{not_ST_{C_F}^{l_goal}} \\
(c4)\ \neg r_F & \neg r_D & ST_{D_F}^{l_goal} & not_ST_{D_F}^{l_goal} \\
\mathbf{(c5)\ \neg r_C} & \mathbf{\neg r_A} & \mathbf{ST_{A_C}^{l_goal}} & \mathbf{not_ST_{A_C}^{l_goal}} \\
(c6)\ \neg r_C & \neg r_B & ST_{B_C}^{l_goal} & not_ST_{D_E}^{l_goal} \\
(c7)\ \neg r_D & \neg r_A & ST_{A_D}^{l_goal} & not_ST_{A_D}^{l_goal} \\
(c8)\ \neg r_D & \neg r_B & ST_{B_D}^{l_goal} & not_ST_{B_D}^{l_goal} \\
and & & & \\
(c9)\ \neg r_B & PD_B^{l_goal} & \ldots PD_B^1 &
\end{array}$$

By *gc_heuristic*, SATCP first chooses the *sum* global constraint and as a result GECODE is invoked. Suppose the invocation returns the following CSP solution:

$$stack_1 = w_A,\ stack_2 = w_C,\ stack_3 = w_F$$

SATCP then assigns r_A, r_C and r_F to be true. It also applies the CA operation. After applying the CA operation, among all these 9 clauses only (c3) and (c5) (both in boldface) remain unsatisfied. All the other clauses are satisfied and stacking action literals for the blocks on those clause are not assigned. So, these blocks will not be the part of the goal stack.

As (c3) remains unsatisfied, one of $ST_{C\_F}^{l\_goal}$ and $not\_ST_{C\_F}^{l\_goal}$ will later be assigned. Similarly, as (c5) remains unsatisfied, one of $ST_{A\_C}^{l\_goal}$ and $not\_ST_{A\_C}^{l\_goal}$ will be assigned. These assignments create other required action literal assignments at the other layers and the plan for building the goal stack with F, C and A is generated.

7.6.3 Results

We have solved three instances with different number of blocks and stacks for the planning problem of block stacking with numerical constraint by the SAT(gc) prototype. We solve those problems under 0.30 seconds. The result is shown in Table 7.11.

Stacks	Blocks /Stack	Initial Configuration	Weights	W_{max}	Goal Stack	Time (sec)
2	2	stack1:(ON B A) stack2:(ON D C)	$w_A = 4, w_B = 5$ $w_C = 5, w_D = 7$	10	(ON C B)	0.03
2	3	stack1:(ON B A) (ON C B) stack2:(ON E D) (ON F E)	$w_A = 5, w_B = 6$ $w_C = 7, w_D = 3$ $w_E = 5, w_F = 8$	11	(ON E B)	0.18
3	2	stack1:(ON B A) stack2:(ON D C) stack3:(ON F E)	$w_A = 3, w_B = 4$ $w_C = 2, w_D = 4$ $w_E = 5, w_F = 2$	7	(ON E C) (ON C A)	0.29

Table 7.11: Experiments with block stacking with numerical constraints.

Chapter 8

Summary and Future Work

8.1 Summary

In this thesis, we have devised an algorithm for the framework SAT(gc), which incorporates CSP style constraint solving into SAT solving, by integrating a DPLL based SAT solver and a generic constraint solver. We also have provided argument on the correctness of the algorithm. To demonstrate the feasibility of the SAT(gc) framework we have implemented a prototype system of SAT(gc), namely SATCP. We also have performed some experiments with four benchmarks problems on SATCP.

The SAT(gc) framework allows part of a SAT problem to be encoded in terms of global constraints. The generic constraint solver is used as a black box by the SAT solving engine to obtain solutions of global constraints when needed. The returned CSP solutions are converted into their corresponding Boolean literals, and these literals are used in the searching process of the SAT solving engine to determine satisfiability. In case of a conflict generated by the solution of a global constraint or by a failed global constraint, the framework handles those conflicts properly.

The SAT(gc) framework is closely related with the SMT framework. We have identified some similarities and dissimilarities between them. While problem representation is quite similar in both frameworks, the manners in which deduction and conflict analysis are performed are quite different.

To develop SATCP, we have integrated ZCHAFF, as the DPLL based SAT solving engine and GECODE, as the generic constraint solver. In SATCP, we have introduced a heuristic, named *gc_heuristic*, which puts highest priority on the gc-variables for making decisions.

We have performed experiments on SATCP with four benchmark problems. Two of the benchmarks, Latin Square Problem and Magic Square Problem, are taken from the puzzle problem domain. Rest of the two, ferry planning problem with numerical constraint and block stacking planning problem with numerical constraint, are taken from the planning domain.

We have achieved better efficiency from SATCP over ZCHAFF for the Latin Square Problem, as part of the problem is encoded compactly with *allDiff* global constraints which have specialized and efficient propagators implemented in GECODE. While we attempted to solve the Magic Square Problem, we have obtained an important insight about the drastic consequence of decomposing large constraints into pieces and putting them into separate constraint stores, in which case constraint propagation is not effective for global constraints that share CSP variables. As a result, though we solved the magic square of order 3, we failed to solve larger instances of magic square problem in a reasonable amount of time. Our attempt to solve this problem in SAT(gc) demonstrates that we can incorporate numerical constraints, such as *sum* in SAT solving, which has no direct encoding in Boolean logic.

SAT solvers are competitive in solving the planning problem. But for the planning problems that require numerical constraints, the SAT solving is not the best choice. The

SAT(gc) framework is able to solve this type of problems easily. We have designed two new planning problems with numerical constraints, namely the ferry Planning problem with numerical constraints and block stacking planning problem with numerical constraints. We have solved this two new problems using SATCP. This demonstrates the usability of the SAT(gc) framework to solve new type of problems, which require both SAT and CSP style problem representation, in order to get solved efficiently.

8.2 Future Work

In this thesis, the conflict analysis involving gc-variables is conservative, in the sense that whenever a conflict analysis occurs due to a gc-variable basically chronological backtracking is performed without learning. In this connection, an interesting question arises as how non-chronological backtracking with learning can be performed for the conflicts involving gc-variables.

The problem with the decomposed magic square problem raises another interesting point in regards to implementation. A more rigorous and successful implementation can be done by modifying a constraint solver, which will enable constraint propagation between separate constraint stores. In SATCP, for getting the i^{th} alternative solution for a failed gc-variable, we need to generate $i-1$ alternative solutions from the search engine created in the local scope. This is clearly undesirable as the same work is repeated carried out. This is a major weakness of the current implementation of SATCP. To avoid this, it appears that the constraint solver needs to be modified so that the search engines that are created at the beginning of the execution of SATCP (at global scope) for each of the global constraints, can be utilized for the failed gc-variables.

Another interesting point worthy of an investigation is potential applications. That is, what type of industrial benchmarks can be devised for the SAT(gc) framework so that the full power of SAT(gc) can be utilized. A possible direction is to design benchmarks which subsume planning and scheduling (with numerical constraints) at the same time.

Bibliography

[1] SMT-COMP'06: 2nd satisfiability modulo theories competition, 2006. http://www.csl.sri.com/users/demoura/smt-comp/.

[2] Global constraint catalog, 2011. http://www.emn.fr/z-info/sdemasse/gccat/.

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

[4] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2006.

[5] F. Benhamou, N. Jussien, and B. O Sullivan. *Trends in Constraint Programming*. ISTE Ltd, 2007.

[6] Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Propagating conjunctions of alldifferent constraints. In *Proc. AAAI'10*, 2010.

[7] L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys*, 38(4), 2006.

[8] M. Buro and H. Kleine-Buning. Report on a sat competition. Technical Report Technical Report, University of Paderborn, 1992.

[9] DIMACS center. Satisfiability suggested format, 1993. http://personnel.univ-reunion.fr/fred/Enseignement/CalculComplex/SAT/DIMACS.pdf.

[10] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems*, 24(3):305–317, 2005.

[11] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Assoc. Computing Machinery*, 7(3):201–215, 1960.

[12] L. de Moura and H. Ruess. An experimental evaluation of ground decision procedures. In *Proc. CAV'04*, LNCS 3114. Springer, 2004.

[13] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[14] J.W. Freeman. *Improvements to propositional patisfiability search Algorithm*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[15] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *Proc. of the 25th International Conference on Logic Programming*, pages 235–249, 2009.

[16] M. Gelfond. Answer sets. In *Handbook of Knowledge Representation*. Elsevier, 2008.

[17] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP*, pages 1070–1080, 1988.

[18] E. Goldberg and Y. Novikov. BerkMin – a fast and robust SAT solver. In *Design Automation and Test in Europe (DATE2002)*, pages 142–149, 2002.

[19] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.

[20] G. Hou. Relating constraint propagation techniques in CSP with Answer Set Programming. Master's thesis, University of Alberta, 2004.

[21] R.G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[22] H. Kautz, B. Selman, and J. Homann. SatPlan: Planning as satisfiability. In *Abstracts of the 5th International Planning Competition*, 2006.

[23] D. B. Kemp and J. S. Peter. Semantics of logic programs with aggregates. In *Logic Programming, Proceedings of the 1991 International Symposium*, pages 387–404, 1991.

[24] S.M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[25] G. Liu and J. You. Lparse programs revisited: semantics and representation of aggregates. In *Proc. ICLP'08*, pages 347–361, 2008.

[26] A. Lońpez-Ortiz, C-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proc. IJCAI '03*, pages 245–250, 2003.

[27] I. Lynce. *Propositional Satisfiability: Techniques, Algorithms and Applications*. PhD thesis, Instituto Superior Tcnico, Universidade Tcnica de Lisboa, 2005.

[28] M. Logman M. Davis and D. Loveland. A machine program for theorem proving. *CACM*, 5:394–397, 1962.

[29] S. Malik. The quest for efficient sat solvers. Distinguished Lecture Series, The Gaschnig/Oakley Memorial Lecture, 2004.

[30] S. Malik and L. Zhang. zChaff: a state of the art SAT solver, 2011. http://www.princeton.edu/~chaff/zchaff.htm.

[31] J. Marques-Silva. The impact of branching heuristics in propositional satisfiabilityalgorithms. In *Proc. 9th Portuguese Conference on Artificial Intelligence*, 1999.

[32] J. Marques-Silva. Practical applications of boolean satisfiability. In *Proc. 9th International Workshop on Discrete Event Systems*, pages 74–80, 2008.

[33] J. Marques-Silva and K.A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Proc. IEEE International Conference on Tools with Artificial Intelligence*, pages 220–227, 1996.

[34] K. Marriott and P. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[35] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. 38th Annual Design Automation Conference*, pages 530–535, 2001.

[36] M. Nielsen. Parallel search in gecode. Master's thesis, KTH Royal Institute of Technology, 2006.

[37] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: from an abstract davis-putnam-logemann-loveland procedure to dpll(t). *Journal of the ACM*, 53(6):937–977, 2006.

[38] N. Pelov. *Semantics of Logic Programs with Aggregates*. PhD thesis, Ketholieke Universiteit Leuven, 2004.

[39] P. Prosser. Hybrid algorithm for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[40] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. AAAI*, pages 362–367, 1994.

[41] F.S. Roberts. Center for discrete mathematics and theoretical computer science, 2011. http://www.dimacs.rutgers.edu/.

[42] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.

[43] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[44] C. Schulte, M. Lagerkvist, and G. Tack. Gecode : Generic constraint development environment, 2011. http://www.gecode.org/.

[45] C. Schulte, G. Tack, and M. K. Lagerkvist. *Modeling and Programming with Gecode*. 2008.

[46] R. Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.

[47] T.C. Son and E. Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7:355–375, 2007.

[48] W.-J. van Hoeve and I. Katriel. Global constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 7. Elsevier, 2006.

[49] M. Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.

[50] W. Yisong, J. You, F. Lin, L. Yuan, and M. Zhang. Weight constraint programs with evaluable functions. *Annals of Mathematics and Artificial Intelligence*, 60(3-4):341–380, 2010.

[51] H. Zhang and M. Stickel. An efficient algorithm for unit-propagation. In *Proc. International Symposium on Artificial Intelligence and Mathematics*, 1996.

[52] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. The 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285, 2001.

[53] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Proc. CAV'02*, LNCS 2404, pages 17–36. Springer, 2002.