J. King Saud Univ., Vol. 16, *Comp. & Info. Sci*., pp. 17-43 (A.H. 1424/2004)

Preprocessing Overconstrained CSPs to Locate and Resolve Conflicts

I. Shah
Department of Computer Science, College of Computer & Information Sciences
King Saud University, P.O.Box 51178, Riyadh 11543, Saudi Arabia

(Received 21 January 2002; accepted for publication 28 April 2003)

Abstract. A constraint satisfaction problem (CSP) is said to be overconstrained if it does not have a solution, and its constraints are said to be conflicting. Subsets of constraints conflicting with each other are defined here as conflict sets. The process of locating conflict sets is termed conflict location. Algorithms to locate conflict sets in an overconstrained CSP are proposed. It is shown that conflict sets make explicit all the alternative ways of resolving conflict and thus define a relaxation space. An algorithm to find an optimum relaxation, one that relaxes the minimum number of constraints to resolve the conflict, is also proposed. A pre-processing technique is proposed for overconstrained CSP, whereby, relaxation obtained from the conflict sets is used to resolve conflicts in the CSP. This is shown to improve the performance of branch and bound algorithms.

Keywords: Constraint satisfaction; overconstrained CSPs; partial constraint satisfaction.

1. Introduction

Solving a constraint satisfaction problem (CSP) involves determining an assignment of values to a set of variables that satisfies the constraints between them. Such an assignment is called the CSP's solution. A CSP is said to be overconstrained if it does not have a solution, and its constraints are said to be conflicting or inconsistent with each other. Overconstrained CSPs can arise in any application of constraints. When a CSP is overconstrained, an assignment of values, not satisfying all the constraints must be accepted as a solution. Such an assignment is called a partial solution and the process of finding it, the partial constraint satisfaction.

Traditionally over-constrained CSPs are solved by branch and bound methods [2, 11] or heuristic local search [6, 12]. Heuristic methods start with a complete assignment of values, violating an unspecified number of constraints and improve on the assignment heuristically, by choosing alternative assignments. Branch and bound methods are variations of classical backtracking that incrementally extend an assignment of values to variables while minimizing the number of constraint violations. Branch and bound methods have a source of inefficiency: they waste computational effort in extending assignments that in the end turn out to be suboptimal. This paper discusses the reasons behind this inefficiency and proposes preprocessing techniques to overcome it. An experimental evaluation of these techniques is also presented.

Branch and bound algorithm, like classical backtracking, incrementally extends an incomplete solution by assigning values to variables and testing them to check if the new value satisfies the constraints with the previously assigned variables. But unlike backtracking does not necessarily backtrack when a new value violates a constraint with a variable already assigned a value. Instead branch and bound keeps track of the best solution found so far i.e. one that violates least number of constraints, and backtracks when the number of constraint violations (along a search path), exceeds those of the best solution found that far. If it finds a solution violating lesser number of constraints than the current best solution, it replaces the current best solution with that solution. When the algorithm terminates, it returns the current best solution as the maximal solution.

While accepting constraint violations, a branch and bound algorithm makes no distinction between constraints that are conflicting and those that are not, and this leads to inefficiency. Not all the constraints of an overconstrained CSP are conflicting, and therefore responsible for the CSP being overconstrained. Leaving a constraint, which is not responsible for the conflict, unsatisfied in an evolving solution, is unnecessary and redundant. A maximal (optimal) solution should leave no such constraint unsatisfied. A branch and bound algorithm has no information about the constraints responsible for the conflict. It goes on extending an evolving solution, which leaves a constraint not responsible for conflict unsatisfied. Such an evolving solution turns out to be suboptimal but after the wasted computational effort of performing some constraint checks. One could improve the performance of branch and bound algorithms if constraints responsible for the conflict were known in advance. This information can be used in two ways: the branch and bound algorithm would immediately backtrack when an evolving solution violates a constraint not responsible for the conflict; or such constraints could be relaxed beforehand, yielding a CSP which is not overconstrained. This paper focuses on the latter approach.

The subsets of conflicting constraints of an overconstrained CSP are defined here as *conflict sets* and the process of locating them *conflict location*. A conflict set is critical, in the sense that relaxing any one constraint from it, resolves the conflict due to it. When all the conflict sets of an overconstrained CSP are known, all possible alternative ways of resolving conflict become obvious. It is easy to search among these alternatives, for an alternative that relaxes the minimum number of constraints. Such an alternative is called an *optimal relaxation*. An optimal relaxation corresponds to a maximal solution found by traditional maximal satisfaction techniques and relaxes exactly the same number of constraints that a maximal solution leaves unsatisfied.

A depth-first search algorithm to locate conflict sets is given here. Experiments study the cost of locating the conflict sets in comparison to finding a maximal solution with the branch and bound algorithms. In general locating all the conflict sets is computationally very hard but locating a subset of conflict sets is easy. We present two methods to locate a subset of conflict sets. The first method limits search by carrying out a depth-limited search. The second method searches only in the tightly connected subproblems of a given CSP. One is not guaranteed to determine an optimal relaxation with a subset of conflict sets. Our approach is to use the relaxation obtained from the subset of conflict sets to relax constraints of the overconstrained CSP, and then solve the resulting CSP using a branch and bound algorithm. The pre-processing is shown to improve the performance of branch and bound algorithm, which are able to find good near optimal solutions quickly.

Conflict location is similar in spirit to dependency directed backtracking (also called backjumping) [1,9] and various consistency techniques for CSPs [5]. All the three techniques aim at finding the cause of an inconsistency and eliminate it from its source. The latter two techniques are employed in searching a solution space. Dependency-directed backtracking limits search, by changing a variable assignment, responsible for an inconsistency and not the chronologically previous value assigned, which would lead to the same inconsistency being rediscovered repeatedly during backtrack search; a source of inefficiency. Consistency techniques preprocess a CSP and remove inconsistent combinations of values from the variables, which otherwise would lead to a `thrashing' behavior during backtracking search. Conflict location has a similar role in solving an inconsistent CSP: to identify sets of constraints that are conflicting with each other, so that some of them can be relaxed to yield a solvable CSP. Solving overconstrained CSPs has been viewed as search in a problem space for a solvable CSP [2]. Conflict location can be used in the problem space search, just as the other two techniques are used in solution space search.

Section 2 defines conflict sets and simple examples illustrate how conflict in an overconstrained CSP might be resolved optimally when all its conflict sets are known. The problem of locating conflict sets is an enumeration problem; the complexity of this problem is discussed. Algorithms for locating conflict sets, completely and partially, are presented.

Conflict sets of an overconstrained CSP define a relaxation space. Section 3 presents an algorithm to search the relaxation space and determine an optimal relaxation.

An optimal relaxation is guaranteed only when all the conflict sets are known. A relaxation obtained, optimal or suboptimal, can be applied to resolve conflicts in the given CSP. We discuss how this preprocessing might improve the performance of branch and bound algorithms.

Section 4 presents the results of the experiments. In general, it is hard to locate all the conflict sets in an overconstrained CSP compared to solving them by branch and bound techniques, but results show that it is very easy to locate conflict sets in CSPs with tight constraints. And it is even easier to determine a subset of the conflict sets for this class of CSPs. Relaxation determined from these conflict sets, when applied to the given CSP, improves the performance of the branch and bound algorithms, and good near optimal solutions can be found easily.

2. Basic Definitions

Here finite CSPs are considered. A finite constraint satisfaction problem (CSP) consists of a set of problem variables $V = (X_1,...,X_n)$, which take values from a finite domain of symbolic values D, and a set of constraints $\boldsymbol{C}$. A constraint $Ci_1,i_2,...,i_k \in \boldsymbol{C}$ is a subset of D^k, with $1 \leq k \leq n$. The constraint $Ci_1,i_2,...,i_k$ is said to be of arity k and between distinct variables $Xi_1,...,Xi_k$. Solving a CSP involves determining an n-tuple $(d_1,...,d_n)$ of values such that $d_i \in D_i$, $i=1,...,n$ and all the constraints are *satisfied*, that is, for every constraint $Ci_1,i_2,...,i_k \in \boldsymbol{C}$, $(di_1,...,di_k) \in Ci_1,i_2,...,i_k$. The n-tuple is called a *solution* of the CSP. Without loss of generality discussion is restricted to only two types of constraints: *unary* constraints and *binary* constraints. Unary constraints are of arity one and specify values a variable can take; binary constraints are of arity two, and specify the combinations of values acceptable for the two variables.

A solution n-tuple, satisfying all the constraints in $\boldsymbol{C}$, is the CSP's *complete solution* or simply a solution. A CSP that does not have a complete solution is said to be *overconstrained* or *inconsistent*. For an overconstrained CSP, an n-tuple satisfying only a subset of the constraints in $\boldsymbol{C}$, is accepted as a solution and is called a *partial solution*. The number of constraints left unsatisfied in a partial solution is the *distance* of the partial solution. A CSP's *maximal solution*, is a partial solution with minimum possible distance i.e. one that satisfies maximum number of constraints. A graphical representation of a CSP, with nodes representing variables and arcs the binary constraints, is called its *constraint network*. Fig.1, presents an example of a constraint network, for a CSP with domain $D=\{a,b,c,d,e\}$, variables $V=\{1,2,3,4,5\}$, and binary constraints $\{C_{12},C_{13},C_{15},C_{23},C_{24},C_{25},C_{34},C_{35},C_{45}\} \in \boldsymbol{C}$. Values allowed by the unary constraints are shown within the nodes, and sets of 2-tuples allowed by the binary constraints, as arc labels. A constraint C_i is a *relaxation* of a constraint C_j if C_i allows more tuples than C_j, that is, C_i is a superset of C_j. In this case, constraint C_j is also said to be *tighter* than C_i. For example, a binary constraint allowing tuples *{ab, ae, bc}*, is a relaxation of the constraint C_{13}. A constraint is said to be *totally relaxed* if it allows all possible combinations of values. The terms relaxation and total relaxation will be used interchangeably here. A constraint satisfaction problem CSP_i is a relaxation of CSP_j, if CSP_i is obtained by relaxing one or more constraints of CSP_j.

3. Conflict Location

Conflict location is the process of identifying the conflict sets in an overconstrained CSP. This section defines the conflict sets and an example is used to show how conflict might be resolved, when all the conflict sets of a CSP are known. Computational complexity of the problem of conflict location is discussed. A basic search algorithm for conflict location is given and the possible search strategies that it might be use are discussed. A depth-first conflict location algorithm is presented in detail. Lastly, we present two methods to limit the depth-first search and locate only a subset of the conflict sets, a process we call partial conflict location.

3.1 Conflict sets

Formally, *conflict set* of an overconstrained CSP *P*, is defined as a minimal, connected subset of constraints of *P*, which cannot be satisfied together. A conflict set being minimal, all its proper subsets are consistent and all its supersets are inconsistent. Thus, if $G_c = (V_c, E_c)$ is the constraint network corresponding to the conflict set with vertices V_c representing the variables, and edges E_c the binary constraints, all its sub networks $H_c = (W_i, F_i)$, with $W_i \subseteq V_c$ and $F_i \subset E_c$, are consistent. Conflict due to a conflict set may, therefore, be resolved by relaxing at least one constraint from the conflict set. The number of constraints in a conflict set i.e. $|E_c|$, is referred to as the *size* of conflict set.

Figure 1 presents examples of conflict sets. Constraint network shown is overconstrained due to three overlapping conflict sets: $\{C_{12}, C_{15}\}$, $\{C_{15}, C_{25}\}$ and $\{C_{15}, C_{45}\}$; constraint C_{15} being common to them. Relaxing at least one constraint from every conflict set of an overconstrained CSP yields a consistent CSP. Relaxing a constraint from a conflict set, in effect, replaces the conflict set by one of its consistent subsets. Relaxing constraints, that are not members of conflict sets, have no effect on the conflict; the resulting CSP will still be overconstrained. To resolve the conflict in the constraint network of Fig. 1, at least one constraint from each of the three conflict sets must be relaxed. The relaxed CSP obtained is consistent and a backtracking algorithm can find a complete solution to it. The complete solution of the relaxed CSPs, may be regarded as a partial solution to the original overconstrained CSP, with the constraints relaxed as those that the partial solution leaves unsatisfied.

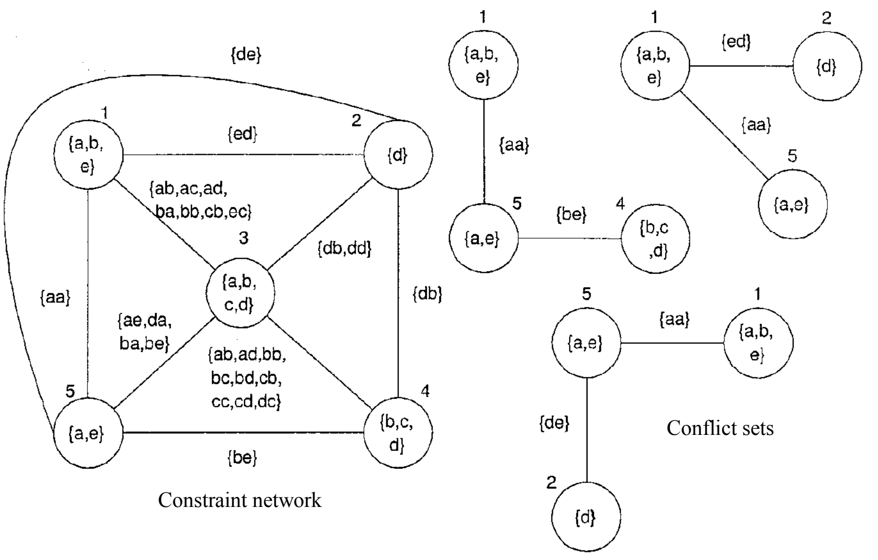


Fig. 1. Constraint network of an overconstrained CSP with three overlapping.

When all the conflict sets are known, all possible ways of resolving conflict in the overconstrained CSP become explicit. The CSP of Fig.1 has eight alternative ways of resolving conflict: relaxing C_{12} and C_{15}, relaxing C_{12}, C_{15} and C_{45}, relaxing C_{12}, C_{25} and C_{15}, and so on. Conflict sets can thus be viewed as defining a space of alternative relaxations or a relaxation space. Each alternative corresponds to a partial solution to the overconstrained CSP, with distance equal to the number of constraints it relaxes. An *optimal relaxation* is an alternative that relaxes minimum number of constraints and it corresponds to a maximal solution. (Section 4 presents an algorithm that searches the relaxation space, for an optimal relaxation.) Determining an optimal relaxation is easy if the conflict sets are disjoint: choosing exactly one constraint to relax from each conflict set is the optimal relaxation. But for overlapping conflict sets, relaxing a shared constraint resolves the conflict due to the overlapping conflict sets. So in this case if an optimal relaxation is desired, constraints must be chosen for relaxation, keeping in view that it resolves maximum number of conflicts. The conflict sets of Fig. 1 are overlapping, with the constraint C_{15} common to them; relaxing this constraint resolves all the three conflicts. A maximal solution to this problem, found by branch and bound, leaves just this constraint unsatisfied.

3.2 Conflict location problem

Locating conflict sets in a CSP is an enumeration problem which can be stated as: "Given a CSP, find all the subsets of its constraints that are conflict sets". A consistent CSP has zero conflict sets, while as an inconsistent CSP has one or more conflict sets. A brute force method to locate conflict sets will generate subsets of constraints and check if they are conflict sets. Each subset of constraints of the given CSP, is a sub-problem (also a CSP), and checking whether it is a conflict set is a decision problem:

CONFLICT-SET Given a CSP, is it the case that the CSP is overconstrained, but relaxing any single constraint from the CSP yields a consistent CSP.

CONFLICT-SET belongs to a class of problems known as critical problems. Critical integer programming asking, "Given a system $Ax \leq b$ is it true that it has no integer solution, but omitting any single inequality permits a solution.", is another example of a critical problem. A slightly better known critical problem is the critical-SAT, which asks: "Given an instance of SATISFIABILITY, is it the case that it is unsatisfiable, but deleting any single clause is enough to yield a subset that is satisfiable". Such an instance of SATISFIABILITY is known as minimally unsatisfiable formula. Complexity of critical problems goes beyond NP. These problems belong to the complexity class D^P, first introduced in [7]. D^P is a class of all languages (or problems) that are the intersection of a language (or a problem) in NP and a language (or a problem) in co-NP. (This is not the same as NP $\cap$ co-NP.) Both NP and co-NP are its subsets i.e. NP, co-NP $\subseteq D^P$. CONFLICT-SET is also in D^P. The problem query has two parts. The first part asks if the given CSP is inconsistent. This part determines the co-NP language. The second part asks if removal of any one constraint yields a consistent CSP. This involves removing $e = |\boldsymbol{C}|$, constraints, one at a time and testing if the resulting CSP is consistent, or e instances of CSP. These e instances can be combined into a single instance and determines the NP language.

Theorem 1 *CONFLICT-SET (or critical—CSP) is D^P—complete.*
Proof. Critical-SAT is a known D^P—complete problem [4]. Critical-SAT reduces to CONFLICT-SET, since SAT can be encoded as a CSP and if SAT is unsatisfiable (satisfiable) its corresponding CSP is inconsistent (consistent).

Returning to the enumeration problem of conflict location, it appears that there is no appropriate complexity class for it, and it remains an open problem [8]. One could consider its membership in the well known complexity class for enumeration problems #P [3, 4]. An enumeration problem belongs to #P if, for all its instances, and for all solutions of each instance the size of the solution is less than equal to polynomial of the instance size, and a solution for an instance can be verified in polynomial time. It is not possible to verify, for a CSP, whether one of its subsets is a conflict set in polynomial time. Therefore conflict location cannot be in #P.

3.3 Possible search strategies

Search space for the conflict location problem consists of $2^{|C|}$ possible subsets of constraints. A possible organization of this search space is the tree organization, depicted in Fig. 2 for a simple three variable CSP. The subsets appear in an order of increasing size from the root to the leaves. At the root of the tree is the null subset, the first level subsets are of size one, the second level has subsets of size two, and so on. Within a subset constraints are ordered in the increasing order of their indices. (A constraint subset may be represented by the indices of its constraints e.g. subset $C_2\ C_5\ C_6$ as (256)). A useful property of this organization is that a set S_p, at any node, is a subset of any of its children S_c. The subset S_c is said to be *subsumed* by S_p. A conflict location algorithm can utilize this property to prune search as follows: if a constraint subset at a node is found to be inconsistent, the subtree rooted at the node (i.e. all the children of the node) is bound to be inconsistent and need not be searched. This property also enables one to check the consistency of the constraint subsets incrementally. This is discussed further below.

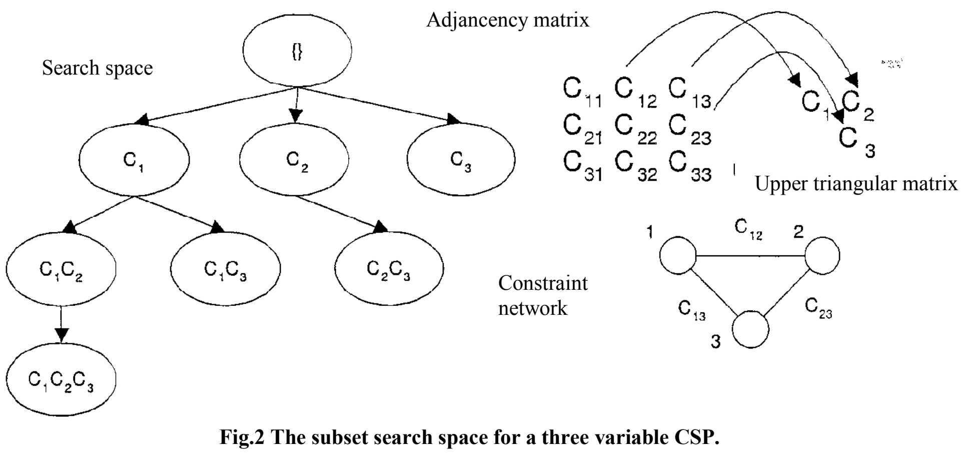


Fig.2 The subset search space for a three variable CSP.

The basic conflict location algorithm proposed, generates the subsets in the subset space, tests them for consistency, and records subsets found to be inconsistent. Since only the minimal inconsistent subsets are required, only those inconsistent subsets, that are not subsumed by any other inconsistent subset, are recorded. The children of inconsistent subset are not generated and tested; they are bound to be inconsistent. Furthermore, a subset subsumed by a recorded inconsistent subset, is not checked for consistency and its children are not generated. This prunes search further.

The order in which nodes are generated and tested for consistency is important for the overall efficiency of the algorithm. A search strategy should aim to generate and test nodes in an order, so that minimal inconsistent subsets are discovered early during search. This will ensure maximum pruning of the search space. A search strategy that discovers many non-minimal inconsistent subsets first, can be inefficient. These non-minimal subsets are found to be subsumed by minimal subsets discovered later, and checking the former subsets for consistency turns out to be a wasted effort.

Standard search strategies can be employed to search the subset space. Each strategy yields a variant of the basic conflict location algorithm and each strategy has its merits and drawbacks. Employing a breadth first strategy (bfs), subsets would be generated and tested for consistency, level by level. In the subset space shown in Fig. 2, first the root node would be expanded, generating and testing C_1, C_2 and C_3. Then from C_1, subsets C_1C_2 and C_1C_3; and from C_2, subset C_2C_3, would be generated and tested. Subset C_3 would not generate any subsets. Lastly, $C_1C_2C_3$ would be generated and tested from C_1C_2. Since subsets get generated and tested in an order of increasing size, minimal inconsistent subsets are guaranteed to be discovered first and search pruning would be maximum. Non-minimal inconsistent subsets would never be tested. Thus, if C_1 is minimally inconsistent, it would be discovered first and non-minimal subsets C_1C_2, C_1C_3, $C_1C_2C_3$ would never be generated and tested. The strategy from this point of view is efficient but breadth first strategy has an exponential space complexity, since all the subsets at a certain level (potentially exponential in number) have to be stored in a queue, before they are expanded to generate the subsets in the next level. This drawback simply makes breadth first strategy an impractical strategy.

Depth first strategy (dfs) goes deeper into the search space and in doing so, unlike breadth first strategy, may discover non-minimal inconsistent subsets. This is a source of inefficiency. For example, if the minimal inconsistent subset is C_3, subsets C_1C_3 and C_2C_3 are tested and later found to be non-minimal when C_3 is discovered. The depth first strategy has modest memory requirements compared to breadth first strategy, since only the subsets in the current search path must be stored during search. Because of its modest memory requirement, depth-first strategy and its variant depth-limited strategy, have been chosen here for experimentation. Other strategies, notably, the best first and the iterative deepening strategy, are possible. A thorough evaluation of these and other strategies can be subject of further research.

3.4 A depth first conflict location algorithm

A stack based depth first conflict location algorithm, DFS_CL(), appears in Fig. 3. In it constraint subsets are generated in a depth-first manner and tested for consistency using an incremental backjumping algorithm, which returns consistent if a solution is found, otherwise inconsistent. The function *generate_next_combination()* generates the subsets and function *check_consistency()* checks their consistency. The variable *CCombination* represents a node in the subset space. It is first initialized to the root node, corresponding to the null subset. In the inner do-while loop, *generate_next_combination()* expands a node and generates its children nodes. This function takes a subset of size d as input parameter and generates subsets of size $d+1$, by appending a constraint index at $(d+1)^{th}$ position, and incrementing this index in each iteration, until the last constraint index is reached. For example, for a CSP with ten constraints, from a parent subset *(2 5 6)* of size *d=3*, *generate_next_combination()* will generate four subsets of size *d=4*, in four iterations of the do-while loop, in the order: *2 5 6 7, 2 5 6 8, 2 5 6 9* and *2 5 6 10* . On the fifth iteration *generate_next_combination()* returns a `nil', indicating all the children of the parent subset have been generated and the do-while loop is exited.

Algorithm DFS_CL (CSP)

```
    initialize STACK;
    initialize TABLE;
    CCombination ←{ };
    Push (STACK, CCombination);
    while not empty (STACK) {
        CCombination ← pop (STACK);
        Do {
            CCombination ← generate_next_combination (CCombination);
            if (CCombination is nil) break;
            if (not subsumed by inconsistent subsets in TABLE) {
                consistent ← check_consistency (CCombination, CSP);
                if consistent
                    push (STACK, CCombination);
                else
                    insert (TABLE, CCombination);
            }
        } while (true);
    };
```

Fig.3. The DFS*_CL() algorithm.*

Constraint subsets found to be inconsistent are recorded in *TABLE*. Search is pruned, as mentioned above, by not testing a subset for consistency if it is subset of an already known inconsistent subset, stored in the *TABLE*. Depth first strategy searches deeper in the search space and in doing so may discover non-minimal inconsistent subsets. Since only the minimal inconsistent subsets are required and need be kept around, whenever a new inconsistent subset is inserted into the *TABLE*, previously stored inconsistent subsets, subsumed by the new subset are deleted from it.

Algorithm DFS_CL()'s efficiency can be improved further by exploiting the properties of the search space. Firstly, disconnected constraint sets need not be tested for consistency. A disconnected constraint subset has more than one connected components that do not share any variables, and therefore do not induce any new constraint on each other. The consistency of the disconnected subset is thus dependent on the consistency of its connected components. Its connected components being its subsets, their consistency gets checked in the upper levels of the subset space tree. It is therefore not necessary to check the consistency of the disconnected subset. Disconnected subsets' children nodes may, however, be connected and must be checked for consistency. The disconnected subsets are therefore not discarded, but are pushed on to the stack without being checked for consistency and their children are generated. A disconnected subset without children nodes is discarded. This saves consistency tests. This mechanism is implemented within the function *check_consistency()*, which treats a connected and a disconnected subset differently.

Secondly, the consistency of constraint subsets can be checked incrementally. Solutions found by backjumping algorithm for a consistent constraint subset are associated and stacked with the subset. When a subsets' children are generated and checked for consistency, the backjumping algorithm starts working from the recorded solution of the parent subset and not from the beginning. This avoids repeating constraint checks already performed in checking the consistency of the parent subset.

3.4.1 Computational complexity

The efficiency of DFS_CL() depends on the average size of the overconstrained CSP's conflict sets. If the CSP has many small-sized conflict sets, large branches of the search space tree will get pruned early during search, as many subsets generated are found to be subsumed by the smaller conflict sets already discovered. But for a CSP with larger conflict sets, not much pruning takes place, and locating conflict sets is more expensive. The worst case occurs when the overconstrained CSP has just one conflict set, with all the CSP's constraints as its members. In this extreme case, there is absolutely no pruning and DFS_CL() must search the complete subset space tree before coming to a halt with one conflict set.

The algorithm DFS_CL() searches in a subset space. At each node in the subset space, its computation is mainly checking the consistency of the subset of constraints associated with the node, through a call to *check_consistency()*. Backtracking procedure *check_consistency()*, searches in a backtracking tree, where levels correspond to variables and nodes to assignments of values to the variables. At each node of the backtracking tree, its main computational step consists of performing constraint checks for a value assigned to a variable, with values of previously instantiated variables. Checking whether values assigned to two variables satisfies the binary constraint between them is a *constraint check*. Total number of constraint checks is a standard measure of a CSP algorithm efficiency, and the computational cost of conflict location can also be expressed in terms of constraint checks. The maximum number of constraint checks done at a node in a backtracking search tree by *check_consistency()*, is *n-1* or *O(n)*, where *n* is the number of variables of the given CSP. The maximum total number of nodes backtracking search tree can have is, $\sum_{i=1}^{n} m^i$ where $m = |D|$ is the number of values a variable can take and i represents levels in the backtracking search tree, with $i=0$ for the root. Therefore, the total number of constraint checks performed by the backtracking procedure i.e. *check_consistency()*, at a node in the subset space is bounded by $\sum_{i=1}^{n} nm^{i} = O(nm^{n})$. The total number of subsets in the subset space being 2^{e}, where $e = |\boldsymbol{C}|$ is the total number of constraints, and since in the worst case DFS_CL() tests them all for consistency, the worst case time complexity of DFS_CL() is $O(nm^{n}2^{e})$.

The space complexity of DFS_CL() is determined by the space required by the *TABLE* to record conflict sets. Maximum number of conflict sets an overconstrained CSP can have, and which TABLE will have to store is $\binom{e}{\lfloor e/2 \rfloor}$. An upper bound on this figure is $O(2^{e})$.

3.5 Partial conflict location

Exhaustive search to locate all the conflict sets can be very expensive. Partial conflict location aims to locate a subset of the conflict sets of an overconstrained CSP, by limiting search to a portion of the search space. Two methods are discussed below. The first method, limits search by limiting the depth to which DFS_CL() searches, and does not locate conflict sets larger than the depth limit. The second method, confines DFS_CL()'s search for conflict sets to tightly connected subproblems of a given overconstrained CSP.

3.5.1 Depth-limited conflict location

DFS_CL() can be made to search only up to a certain depth in the subset space tree and thus locate conflict sets of size not bigger than that depth. The depth limit may be passed on as a parameter to *generate_next_combination()* and prevent it from generating subsets bigger in size than the depth limit.

For the depth-limited DFS_CL() with depth limit d, the maximum number of constraint checks performed at a node within backtracking procedure *check_consistency()* is $d-1$ or $O(d)$, since the number of variables cannot be greater than the number of constraints in it. The total number of nodes in the backtracking search tree is $\sum_{i=1}^{n} m^{i}$, where $m=|D|$, is the number of values a variable can take and i represents levels in the backtracking search tree. The total number of constraint checks is therefore bounded by $\sum_{i=1}^{d} dm^{i} = O(dm^{d})$. Since the total number of subsets tested by the depth-limited DFS_CL() is bounded from above by 2^{e}, where e = $|\boldsymbol{C}|$, an upper bound on the time complexity of depth-limited DFS_CL() is $O(dm^{d}2^{e})$.

3.5.2 Conflict location in subproblems

Another technique to partially locate conflict sets is to look for them in tightly or densely connected subproblems i.e. those with a high average degree. Since densely connected subproblems have many constraints per variable, they usually have fewer solutions and are more likely to be inconsistent, as compared to sparsely connected subproblems. This technique involves two steps. Firstly, densely connected subproblems of a given overconstrained CSP are identified; secondly conflict sets are located within each subproblem. Obviously conflict sets spanning two or more subproblems cannot be located by this technique.

A scheme based on an efficient triangulation algorithm is used to identify densely connected subproblems [10]. The algorithm transforms any graph into a chordal graph by adding edges to it, called the fill-in edges. A graph is *chordal* if every cycle of length at least four has a chord i.e. an edge joining two nonconsecutive vertices. The maximal cliques of the resulting chordal graph, with fill-in edges omitted, are the densely connected subproblems of the given CSP.

The triangulation algorithm consists of two steps: firstly, an ordering for the vertices is computed, using maximum cardinality search; secondly, edges are filled in between any two nonadjacent vertices that are connected through vertices higher up in the ordering.

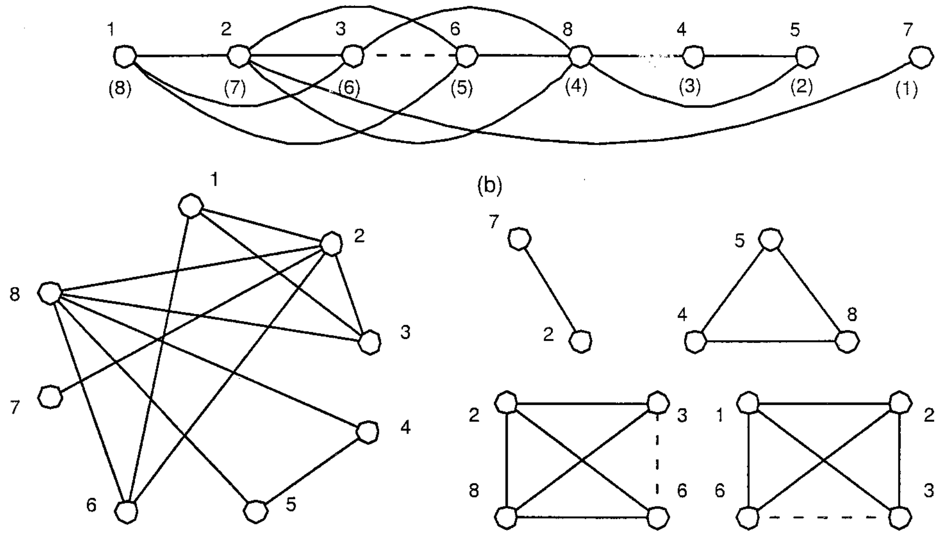


Fig. 4. (a) A constraint network. (b) the ordering produced by the maximum cardinality search and the dummy edge (shown dashed) added by the fill-in step of the trigulation algorithm. (c) the densely connected subproblems extracted as maximal cliques.

The *maximum cardinality search* orders vertices in an increasing order and assigns the next number to the vertex having the largest set of previously numbered neighbours. Vertices densely connected to each other get ordered roughly adjacent to each other. If no edges are added in the second step the original graph is chordal, otherwise the new filled graph is chordal. The maximal cliques of the constraint graph can be indexed by the rank of there highest nodes and extracted. The maximum cardinality algorithm can be implemented in *O(n+deg)*, where *n* is the number of variables and *deg* is the maximum degree. The fill-in step runs in *O(n+m')* where *m'* is the number of arcs in the resultant graph. The ordering produced by the maximum cardinality search is one of the many possible orderings and not necessarily the one that produces minimal fill-in, and the optimally densely connected subproblems.

As an example consider the CSP whose constraint network is shown in Fig.4 (a). Its variables are represented by vertices *{1, 2, 3, 4, 5, 6, 7, 8}* and the constraints by the edges between the vertices. The maximum cardinality search orders the vertices as *{1,2,3,6,8,4,5,7}*, numbering the vertices from 8 to 1. This numbering is shown in Fig. 4(b), with the ordering numbers shown in the parenthesis. The fill-in step adds the dummy edge *(3,6)* to this ordering. The fill-in edge is shown as dashed line in Fig.(b). The maximal cliques can be extracted by using a property of the fill-in graph that any vertex *V* and the vertices connected to it and numbered higher than it, form a clique. The cliques can be obtained in the increasing order of the numbered vertices, discarding a newly generated clique that is contained in a previously generated clique. In Fig.(b) clique *(7,2)*, associated with the vertex 7, which is numbered first in the ordering, is identified first. Clique *(548)* associated with vertex 5 is identified next. Clique *(48)* associated with vertex 4 is discarded since it is contained in the previous clique *(548)*. Altogether four cliques are identified and these are shown in Fig.4(c).

4. Conflict Resolution

Conflict sets of an overconstrained CSP define a space of alternative relaxations. Each relaxation in this space, is a set of constraints which when relaxed in the overconstrained CSP, alters it into a solvable CSP. An optimal relaxation is the one that relaxes a minimum number of constraints. If the conflict sets of an overconstrained CSP are non-overlapping, determining an optimal relaxation is trivial: a constraint chosen from each conflict set, constitutes an optimal relaxation. However, when the conflict sets are overlapping, a shared constraint can resolve conflicts due to all the conflict sets it belongs to. In this situation, relaxing a constraint from every conflict set, relaxes more constraints than are necessary. Such a relaxation is not optimal and determining an optimal relaxation requires some search. This section presents an algorithm, *optimum_relaxation()*, that searches the relaxation space for an optimal relaxation.

In general, it is possible for the conflict sets of an overconstrained CSP to overlap in groups or clusters, that is, the conflict sets within a group overlap each other but conflict sets belonging to two different groups do not. In this situation, optimal relaxation for each group must be determined by *optimum_relaxation()*, separately. The overall optimal relaxation will then consist of the optimal relaxations of the groups.

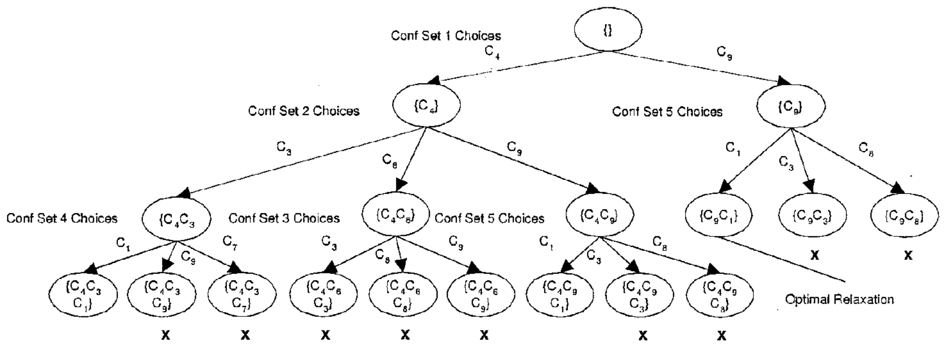


Fig. 5. The search tree traced by optimum_relaxation(), for the example discussed.

4.1 Optimum_relaxation()

Algorithm, *optimum_relaxation()*, is a depth-first branch and bound algorithm that searches the space of alternative relaxations defined by the conflict sets, for an optimal relaxation. The algorithm builds a relaxation by selecting a constraint to relax from each conflict set, until constraints have been selected from all the conflict sets. It keeps track of the smallest relaxation during search and updates it when it finds an even smaller relaxation. It does not extend a current relaxation, if it becomes clear that the current relaxation will not be any better than the smallest relaxation found so far. The algorithm reduces search further by making use of the fact that when a constraint C_i from a conflict set CS_m is relaxed, it also resolves conflict due to all other unconsidered `future' conflict sets of which C_i is a member, say, CS_n, CS_o,…. Therefore, when C_i is selected by the algorithm and included in the current relaxation, future conflict sets CS_n, CS_o,… need not be considered during search because conflict due these sets will already have been resolved. These conflict sets are, therefore, disabled and no constraint is selected from them. However, when the algorithm backtracks and selects a new constraint C_j, from the conflict set CS_m, the disabled conflict sets CS_n, CS_o,… must be restored.[1] The following example illustrates the operation of the algorithm.

[1] This has some resemblance with the forward checking algorithm for constraint satisfaction.

An overconstrained CSP with following conflict sets: (1) C_4, C_9, (2) C_3, C_6, C_9, (3) C_3, C_8, C_9, (4) C_1, C_9, C_7, (5) C_1, C_3, C_8, and (6) C_1, C_2, C_7, is considered. The search tree traced by the algorithm is shown in Fig. 5. Constraint C_4 is selected first from the first conflict set. Since it does not figure in any `future' conflict sets, none of them is disabled. Next, constraint C_3 is selected from the second conflict set; third and fifth conflict sets get disabled and no constraints will be selected from them as long as C_3 is selected. Constraint C_1 is selected next from the fourth conflict set; the sixth conflict set is disabled. At this point there are no more conflict sets left and the first complete relaxation is obtained. This is recorded as the best relaxation. The algorithm backtracks now and retracts the previous selection from the fourth conflict set i.e. constraint C_1 and selects the next alternative C_9 from it. In retracting the selection C_1 it must also restore the conflict sets for whose disablement the constraint C_1 was responsible. The sixth conflict set is therefore, enabled. As a result of the new selection no conflict sets get disabled. However, at this point, there are three constraints selected in the current relaxation; this is equal to the size of the best relaxation known and it is clear that further selections along this path will not lead to a better relaxation. Therefore, search is cut at this point. Search also gets bounded when C_7 is considered. Since there are no further constraints left to consider, the algorithm backs up one level, where the choices C_6 and C_9 are tried from the second conflict set. None of these yields a better relaxation than the one already known. Ultimately after further backing up constraint C_9 is selected from the first conflict set; this disables the second, third and fourth conflict sets. And when C_1 is selected from the fifth conflict set, a smaller relaxation is obtained; this replaces the current best relaxation and finally turns out to be the optimal relaxation.

4.1.1 Time complexity

It is easy to notice that *optimum_relaxation()*'s time complexity depends on the following characteristics: the number of conflict sets, the size of the conflict sets and the amount of overlap among the conflict sets. Large number of conflict sets and large sized conflict sets, are likely to make the search tree deeper and broader, respectively. The more the overlap among conflict sets, the more conflict sets are likely to get disabled when constraints are selected, and the smaller will be the search tree traced by the algorithm. For a certain number and size of conflict sets, the procedure *optimum_relaxation()* performs worse when the overlap between the sets is minimum. An example of such a situation is provided by the following set of conflict sets: $(C_r, \ldots, \underline{C_t}), (\underline{C_t}, \ldots, \underline{C_x}), (\underline{C_x}, \ldots, \underline{C_y}), (\underline{C_y}, \ldots, C_x)$. The set forms a `chain' in which adjacent conflict sets are linked by a single shared constraint. The procedure will search almost the whole search space to determine an optimal relaxation. Incidentally, in this example it is easy to figure out the optimum relaxation without this algorithm.

At any node in the search tree, the key step the procedure performs consists of selecting a constraint from a particular conflict set and checking if the chosen constraint figures in any unconsidered, future conflict sets. The maximum number of checks the procedure can make at a node is *t-1* or *O(t)*, where *t* is the total number of conflict sets. If *l* is the size of the conflict sets, the total number of nodes, in the search tree, at which the checks are made is bounded by $\sum_{i=1}^{t-1} l^i$. (It is assumed that no checks are done at the root and the leaves of the search tree.) Hence the worst case time complexity is bounded by $\sum_{i=1}^{t-1} tl^i = t(l^t - 1)/(l-1) = O(tl^t)$.

4.2 Partial conflict resolution

When all the conflict sets of an overconstrained CSP are known, the relaxation space is completely known and algorithm *optimum_relaxation()* is guaranteed to find an optimal relaxation. An optimal relaxation applied to the CSP, results in a solvable CSP. With only a subset of conflict sets known, the relaxation space is partially known, and an optimal relaxation is not guaranteed. Partial conflict resolution aims to apply the relaxation obtained from the subset of conflict sets, to the overconstrained CSP. The resulting CSP may still be overconstrained but this can improve subsequent search for a solution, with a branch and bound algorithm, as follows.

The branch and bound algorithms' performance depends on the upper bound. If the algorithm starts with a good upper bound (i.e. a distance closer to the maximal distance), or discovers good near-optimal bounds quickly, large amount of search space gets pruned early, resulting in a better performance. Branch and bound algorithms perform better with CSPs having loose constraints and poorly for problems with tighter constraints [2]. The former type of problems have larger sized conflict sets and many near-solutions as compared to the latter, and the branch and bound algorithm finds distances closer to the optimal quite rapidly, pruning the search space and finding a maximal solution quickly. Relaxing constraints can thus improve the performance of branch and bound algorithm by reducing the number of conflict sets and increasing the number of near-solutions in the CSP. The total number of constraints relaxed i.e. those during the preprocessing and those in the maximal solution found after the preprocessing, may not be optimal.

An analogy may again be drawn with consistency techniques [5]. Consistency techniques are used to preprocess CSPs to achieve a certain level of consistency in the CSP and improve the performance of the backtracking algorithms. Here preprocessing of overconstrained CSPs is suggested, to reduce the number of conflicts and improve the performance of branch and bound.

5. Experiments

The aim of the experiments was to compare, the methods of solving overconstrained CSPs through conflict location, with the traditional branch and bound methods. Two main experiments were carried out. In the first experiment, the cost of locating conflict sets completely and partially with DFS_CL(), was compared with the cost of finding a maximal solution with forward checking branch and bound algorithm, P-FC1 [2]. The second experiment was carried out to study the overall performance gain resulting from the pre-processing to partially resolve conflicts.

Randomly generated overconstrained CSPs with varying structural characteristics were used. The structural characteristics of problems, such as: the number of constraints in a problem, the number of domain values and satisfiability (or tightness) of the constraints, were the independent variables of the experiments. The cost was measured in terms of the total number of constraint checks. Variations in the number of conflict sets and the average size of conflict sets, with the above mentioned problem parameters were also studied.

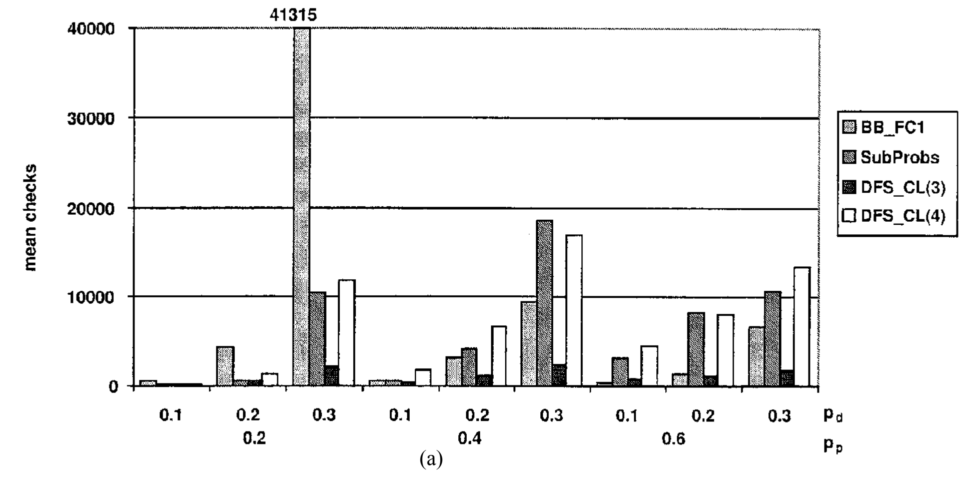
The cost of locating all the conflict sets was found to be exorbitantly high compared to the cost of determining a maximal solution with P-FC1. Experiments with partial conflict location, yielded some interesting results. Using depth-limited DFS_CL() with depth limits of three and four, it was found that the cost of locating conflict sets was a fraction of the cost of finding a maximal solution with P-FC1. This was more so for the problems with tight constraints, for which it is very hard to find a maximal solution with P-FC1. For this class of problems, determining an optimal solution after pre-processing was found to be almost guaranteed. Pre-processing to partially resolve conflicts improves the performance of branch and bound algorithms, and lowers the cost of finding an optimal solution by more than one half.

5.1 Random problems

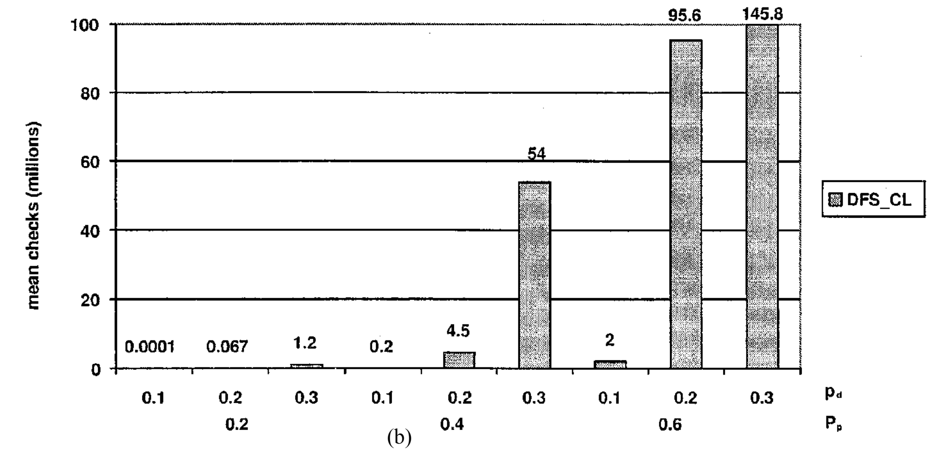
In the experiments, random overconstrained CSPs were generated using probability of inclusion method, as described in [2]. Four problem characteristics can be varied: number of variables, n; number of constraints, c; domain size, d; and satisfiablity (complement of tightness) of a constraint (i.e. the number of value pairs allowed by a constraint), p. Sets of problems were generated for the experiments and each set had a certain value for the above parameters. The number of variables was fixed at ten for all sets. The maximum domain size was fixed at ten as well. The values of d and p were varied, from one problem set to the other, by varying their respective probabilities of inclusion, p_d and p_p. Values of 0.1, 0.2 and 0.3 were used for p_d and values of 0.2, 0.4 and 0.6 were used for p_p. The probability of inclusion of a constraint, p_c, was always set equal to 0.3. (Identical values have been used in the experiments described in [2].) It was ensured that a generated problem had variables with the value of d at least equal to one, and constraints with the value of p at least equal to one. This meant that the problems generated would never have conflict sets of size one. The problem generation method also ensured that each problem generated had a connected graph.

5.2 Complete conflict location

The first experiment compared, complete and partial conflict location, with the forward checking branch and bound algorithm P-FC1. For complete conflict location sets of twenty problems were used. The parameter p_d took the values 0.1, 0.2 and 0.3 and p_p took the values 0.2, 0.4 and 0.6; nine sets were generated, one for each of the nine combination of values. The total number of constraint checks was averaged over the problems in each set.



(a)



(b)

Fig. 6. (a). The cost of finding a maximal solutions with P-FC1, and locating conflict sets partially with DFS_CL() vs. domain size p_d, and constraint satisfiability, p_p.(b). the cost of locating all the conflict sets with DFS_CL() vs. domain size, p_d and constraint satisfiability, p_p.

The results of this experiment are depicted in Fig. 6(a) and (b). The cost of locating all the conflict sets with DFS_CL()was exorbitantly expensive compared to finding a maximal solution with P-FC1. The cost of locating conflict sets increased very rapidly with increasing domain size, exactly as it does for P-FC1. However, the cost decreased rapidly with increasing tightness (or decreasing satisfiability) of the constraints. This was in total contrast with the behaviour of P-FC1, where it became costlier to find a maximal solution with increasing tightness. It was therefore relatively very easy to locate conflict sets for problems for which it was very difficult to find maximal solution using P-FC1. This can be intuitively explained as follows.

Algorithm DFS_CL() prunes large branches of the search space tree if inconsistent subsets are small in size and are discovered early. The cost of locating conflict sets is therefore less for overconstrained CSPs having smaller conflict sets compared to ones having larger conflict sets. In the latter case, DFS_CL() spends more time going deeper into the search space tree and perform more constraint checks. The likelihood of a CSP having smaller conflict sets decreases with increasing domain size and increasing constraint satisfiability, as more and more tuple values are likely to be consistent with the constraints. Therefore, CSPs with higher values of domain size and satisfiability tend to have larger conflict sets. For such problems, pruning in DFS_CL() is lesser, with the result that a higher cost is incurred in locating the conflict sets.

In the same experiment, for each problem the number of conflict sets, the average size of the conflict sets, and the distance of maximal solution found were also recorded. The averages of these variables are plotted in Fig. 7 and Fig. 8. The plot for average size of conflict sets supports the above intuitive explanation: with increasing domain size and constraint satisfiability values the average size of the conflict sets increases.

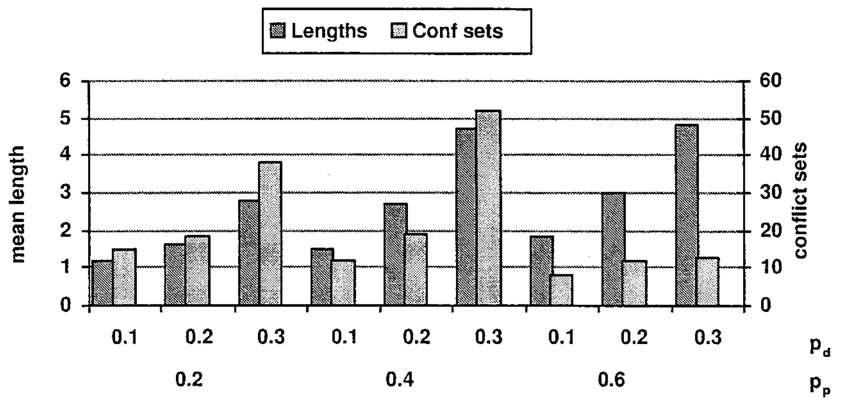
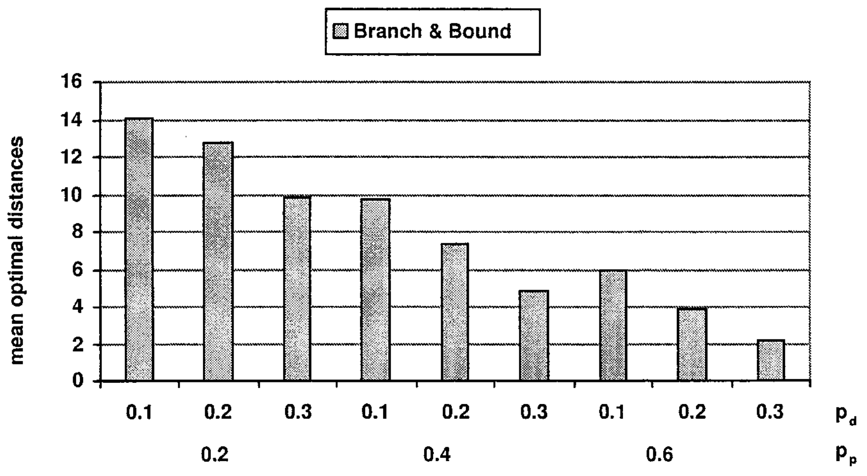


Fig. 7. Mean conflict set sizes and the mean total conflict sets vs. domain size and satisfiability.



Fgi. 8. Mean optimal distances vs. increasing domain size and satisfiability.

The total number of conflict sets for problems, increased with increasing values of domain size; but with increasing values of satisfiability, the number first increased from $p_p=0.2$ to $p_p=0.4$ and then decreased with $p_p=0.6$. This can be attributed to two reasons. Firstly, as stated above, with increasing domain size and satisfiability the possibility of a problem having smaller conflict sets decreases and larger and larger sized conflict sets become more likely. Secondly, among the 2^n subsets a set of n constraints can have, the number of middle-sized subsets, is more than the small-sized subsets or the large-sized subsets. For example, for a set of ten constraints:

$$\binom{10}{1} \leq \cdots \leq \binom{10}{5} \geq \cdots \geq \binom{10}{10}$$

Middle-sized subsets being more in number, the likelihood having more middle-sized conflict sets is proportionally more, compared to smaller or larger sized conflict sets. This explains why the number of conflict sets increase with increasing values of p_d, for each value of p_p. This explanation also seems to suggest that the total number conflict sets should start decreasing with still higher values of p_d, as large-sized conflict sets become more and more likely. This was not verified. However, decrease in the number of conflict sets was clear for the values of p_p: the number of conflict sets peaks for $p_p = 0.4$ and decreases for $p_p = 0.6$.

Optimal distances decreased (i.e. maximal solutions become better) with increasing domain size and satisfiability. Average distances of maximal solutions found by branch and bound algorithms are plotted for each problem set in Fig. 8. This is to be expected; because with increasing domain size and satifiability average conflict set size increases. Larger conflict sets tend to overlap more; and with many overlapping conflict sets it becomes possible to resolve the conflict optimally with fewer constraint relaxations and hence the lower distances. That the distances of maximal solutions decrease with increasing average size was supported by the results of an experiment in which, a set of one hundred problems was generated with $p_d=0.2$ and $p_p=0.4$. From this set, problems with a total number of conflict sets between ten and fifteen, where taken and their average conflict set size and distance distributions plotted against each other. The plot appears in Fig. 9; it shows a decrease in the mean optimal distances with increasing mean conflict set size.

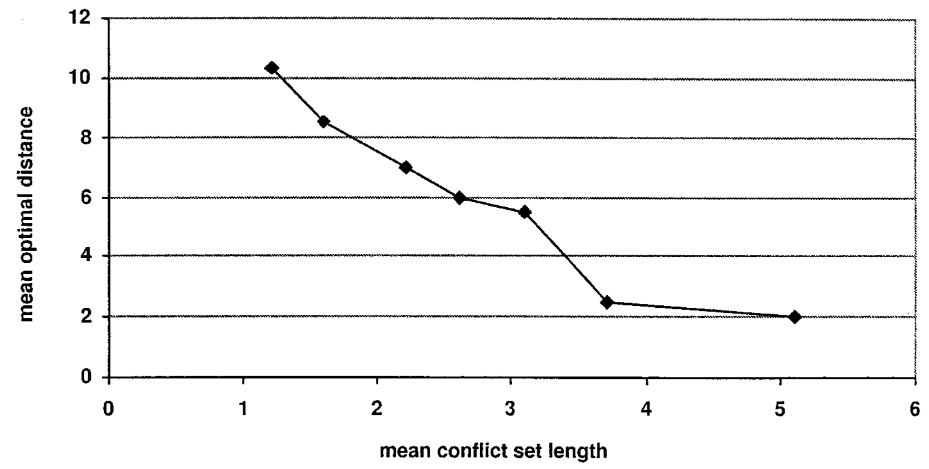


Fig. 9. Mean optimal distances vs. mean conflict set size.

5.3 Partial conflict location

For partial conflict location the first experiment was repeated, with forty problems in each set. The two methods tried were: depth-limited DFS_CL() and search in tightly connected sub-problems. The results are plotted in Fig. 6(a). The depth limits tried were three and four. For both the limits the cost of locating conflict sets increased with increasing values of p_d, for each value of p_p. The cost of locating conflict sets of size up to three was always smaller than the cost of obtaining maximal solution with P-FC1; the difference being more pronounced for $p_p = 0.2$ and $p_p=0.4$. The cost of locating conflict sets up to size four was smaller than that of P-FC1 for only $p_p=0.2$. It was noticed that the cost for both the limits decreased slightly from $p_p=0.4$ to $p_p=0.6$. This happens because with increasing satisfiability, not only do the small sized conflict sets become less likely but also smaller consistent subsets of constraints tend to have more solutions, thereby increasing the chances of backjumping algorithm finding a solution to the consistent subsets quickly.

The percentage of the total conflict sets located by DFS_CL() for both the depth limits and sub-problems search, was recorded and the plot appears in Fig. 10. Since for lower values of p_d and p_p the average conflict set length is small, a very high percentage of the total conflict sets were located by DFS_CL() under both the depth limits and through sub-problem search. The percentage of the conflict sets located decreased with increasing values of p_d and p_p, as the average conflict set length increased. The percentage of conflict sets located under depth limit four was always more than or equal to that located under depth limit three and through sub-problems search.

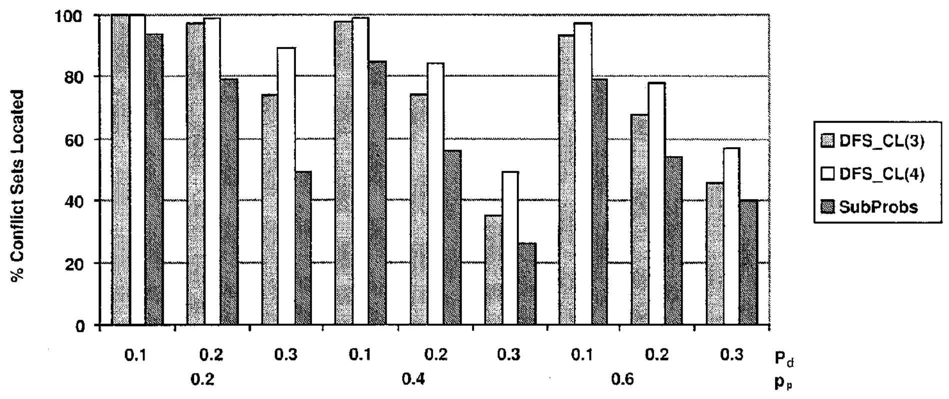


Fig.10. The percentage of the total conflict sets, located by DFS_CL() with depth limits of three and four, and by search in sub-problems, plotted against the values of domain size and satisfiability.

5.4 Preprocessing

The second experiment was to study the overall cost of solving overconstrained CSPs through conflict location. The relaxation obtained from the partially located conflict sets was applied to the overconstrained CSPs, and every constraint in the relaxation was totally relaxed in the CSP. The relaxed CSPs were then solved using P-FC1. The results of the experiment appear in Fig. 11, where the average *total* cost of solving the CSPs (i.e. the cost of locating conflict sets partially and cost of branch and bound search with P-FC1), is plotted against the domain size and constraint satisfiability. The pre-processing was found to improve the performance of P-FC1 and lower the overall cost of solving the CSP. The maximum gain was for problem sets with $p_p = 0.2$. Conflict location with depth limit of three, almost always yielded the best results. But where optimal solutions always found?

As mentioned earlier, relaxation obtained from a subset of conflict sets may not be optimal and applying it to an overconstrained CSP may relax more constraints than necessary. This does not guarantee that the number of constraints relaxed in pre-processing and those left unsatisfied in the solution found by P-FC1, equals the distance of the maximal solution. We computed, for each problem, the sum of the constraints relaxed in pre-processing and the distance of the solution obtained subsequently with P-FC1, and compared it with the distance of the maximal solution for the problem. The sum was found to equal the maximal solution distance in majority of the cases, for all the three methods, indicating that obtaining optimal solutions is highly likely. The results are plotted in Fig.12. Conflict location with depth limit of four yields the best results. The quality of the suboptimal solutions, in terms of total constraints relaxed, was not bad either. The total distance of the solutions, exceeded the optimal as follows. For depth-limited conflict location, with a depth limit of four and three, it exceeded the optimal at the most by one and two, respectively. For conflict location in the subproblems it exceeded the optimal at the most by three.

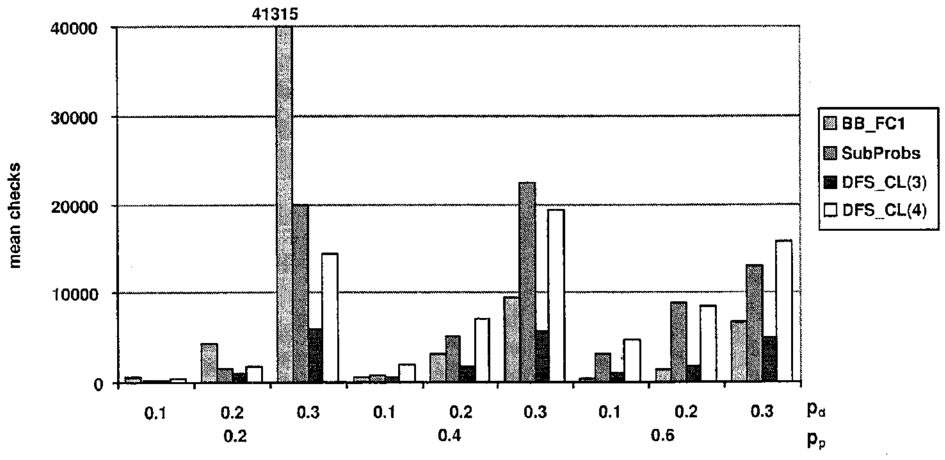


Fig. 11. The performance of P-FC1, before and after pre-processing.

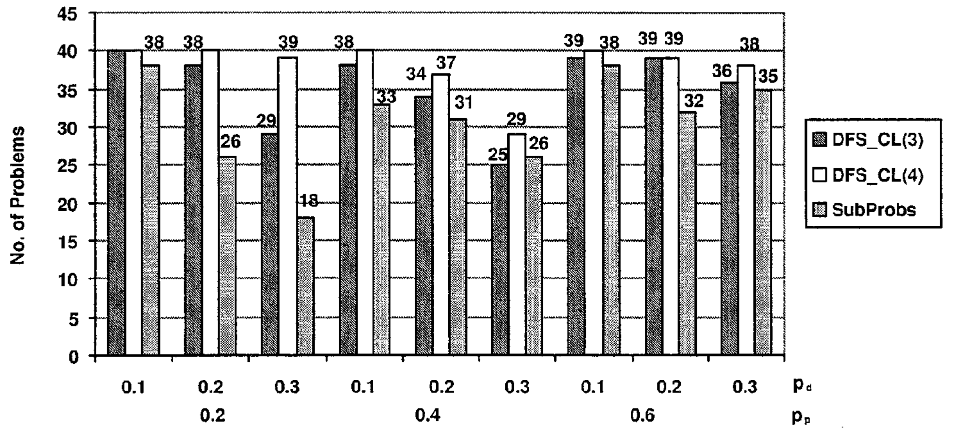


Fig. 12. The number of cases in which optimal solutions were found after preprocessing.

6. Conclusion

The reasons for a CSP being overconstrained are one or more, minimal subsets of constraints that cannot be satisfied together. These subsets were called the conflict sets. Conflict sets define a space of alternative relaxations and make explicit all possible ways of resolving conflict in the overconstrained CSP. This relaxation space can be searched for an optimal relaxation, which corresponds to a maximal solution. Algorithms were proposed to locate conflict sets and search for an optimal relaxation in the relaxation space defined by the conflict sets. There is scope for further work. One may consider and evaluate alternative search strategies for conflict location. The relation between problem structure and the difficulty of locating conflict, also needs to be looked into. Problems of a certain structure may define subclasses of easy problems.

The experimental results showed that locating all the conflict sets of an over-constrained CSP is very hard in general, but for problems with tight constraints, it is easy. Locating conflict sets of size up to four or less in these problems, is even easier. It is hard to determine a maximal solution with branch and bound algorithms for this class of CSPs. These size limited conflict sets constitute the majority of conflict sets in this class of problems and experimental results show that one is certain to find an optimum relaxation from this subset of conflict sets. The relaxation, optimal or sub-optimal, obtained from the subset of conflict sets, was used to resolve conflicts in the given overconstrained CSP. Experimental results show that this pre-processing improves the performance of branch and bound algorithms and good near optimal solutions can be found easily.

In summary, the paper presents a technique of preprocessing overconstrained CSPs, which improves the performance of branch and bound algorithms used to solve them, and is particularly suited to problems with tight constraints.

Acknowledgement. The author would like to thank Computer Science Department for its support.

References

[1] Dechter, R. "Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition." *Artificial Intelligence,* 41 (1990), 217-312.

[2] Freuder, E. C. and Wallace, R. J. "Partial Constraint Satisfaction." *Artificial Intelligence*, 58 (1992), 21-70.

[3] Garey, M. R. and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: Freeman, 1979.

[4] Johnson, D.S. "A Catalog of Complexity Classes." In: van Leeuwen, J. (Ed.), *Handbook of Theoretical Computer Science*. Vol A, Amsterdam: North-Holland (1990), 69-161.

[5] Mackworth, A. "Consistency in Networks of Relations." *Artificial Intelligence*, 8 (1977), 99-118.

[6] Minton, S., Johnston, M. D., Philips, A. B. and Laird, P. "Minimizing Conflicts: a Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems." *Artificial Intelligence*, 58 (1992), 161-205.

[7] Papadimitriou, C. H. and Yannakakis, M. "The Complexity of Facets (and some facets of complexity)." *J. Computer System Science*, 28 (1984), 244-259.

[8] Papadimitriou, C. H. Private Communication (2001).

[9] Tallman, R. M. and Sussman, G. J. "Forward Reasoning and Dependency Directed Backtracking in a System for Computer-aided Circuit Analysis." *Artificial Intelligence*, 9 (1977), 135-196.

[10] Tarjan, R. E. and Yannakakis, M. "Simple Linear-time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs and Selectively Reduce Acyclic Hypergraphs." *SIAM J. Computing*, 13, No. 3 (1984), 566-579.

[11] Wallace, R. J. "Directed ARC Consistency Preprocessing as a Strategy for Maximal Constraint Satisfaction." in M. Meyer Ed., *Constraint Processing: Lecture Notes in Computer Science*, Vol. 923, 121-138.

[12] Wallace, R. J. and Freuder, E. C. "Heuristic Methods for Over-constrained Constraint Satisfaction Problems." *Working Notes of CP'95 Workshop on Over Constrained Systems*, Cassis, France, Sept. 1995.

المعالجة السابقة لإرضاء قيود مشكلة ذات قيود مشدّدة بغية تعيين وحل الصراعات

عنايت الله شاه

قسم علوم الحاسب، كلية علوم الحاسب والمعلومات، ص.ب. ٥١١٧٨ ،
الرياض ١١٥٤٣ ، جامعة الملك سعود، المملكة العربية السعودية

(قدّم للنشر في ٢١/٠١/٢٠٠٢م؛ وقبل للنشر في ٢٨/٠٤/٢٠٠٣م)

ملخص البحث. ينظر إلى مشكلة إرضاء القيود (CSP) على أنها ذات قيود مشددة، إذا لم يكن لها حل، وكانت قيودها متصارعة. وسنقوم هنا بتسمية المجموعات الفرعية من القيود المتصارعة بعضها مع بعض، بالمجموعات المتصارعة.

إن عملية تعيين أو تحديد هذه المجموعات المتصارعة تسمى تعيين الصراع. وفي هذا البحث، نقترح خوارزميات لتعيين المجموعات المتصارعة لإرضاء قيود مشكلة ذات قيود مشددة بغية حل هذه الصراعات. وسنري هنا كيف أن هذه المجموعات المتصارعة تُظْهر كل الطرق المختلفة لحل هذه الصراعات وبالتالي تحدد فراغ التراخي (الذي يكمن فيه الحل مع مراعاة حدود القيود).

ونقترح في هذا البحث أيضا خوارزمية لإيجاد فراغ التراخي الأمثل، الذي ييسر تعيين أقل عدد ممكن من القيود لحل الصراع. وفي هذه الخوارزمية، نقترح طريقة معالجة سابقة (ابتدائية) لمشكلة إرضاء القيود المشددة (CSP) بحيث تستخدم معلومات التراخي التي نحصل عليها من المجموعات المتصارعة في الحل الأمثل لهذه المشكلة. وأثبت البحث أن هذه الطريقة تحسّن أداء خوارزميات التفرع والتقييد المستخدمة في الوصول إلى الحل المطلوب.