Mapping Constraint Satisfaction Problems to Algorithms and Heuristics

Edward Tsang
Alvin Kwan
Department of Computer Science
University of Essex
Colchester CO4 3SQ
tel: 0206 872774, 0206 872138
email: edward@essex.ac.uk, alvin@essex.ac.uk

Abstract

Constraint satisfaction has received great attention in recent years and a large number of algorithms have been developed. Unfortunately, from the problem solvers' point of view, it is very difficult to see when and how to use these algorithms. This paper points out the need to map constraint satisfaction problems to constraint satisfaction algorithms and heuristics, and proposes that more research should be done on how to retrieve the most efficient and effective algorithms and heuristics for a given problem. We claim that such algorithms/heuristics retrieval systems should also be valuable to guide future research.

1 Introduction

Constraint satisfaction is a general problem which appears in many places, notably scheduling. Because of its generality and importance, constraint satisfaction has received a great deal of attention in recent years. A (*finite*) *constraint satisfaction problem* (CSP) is a problem which consists of a set of variables, each of which has a finite domain from which it has to take a value, and a set of constraints restricting the values that the variables can take simultaneously. We call the assignment of a value to a variable a *label* and the simultaneous assignment of values to a (possibly empty) set of variables a *compound label*. An assignment of a value to each of the variables satisfying all the constraints is called a *solution tuple*. In some problems, all solution tuples need to be found; in some problems, finding any solution tuple would be good enough. In scheduling, some solution tuples are better than others, and one may want to find the optimal solution or near-optimal solutions according to some optimization functions.

This paper makes reference to a large number of CSP-solving algorithms and methods. Explaining each of them in detail is beyond the scope of this paper. For properties of CSPs and algorithms for constraint satisfaction, readers are referred to Tsang [1993].

Currently there is a mismatch between algorithm designers and the problem solvers: algorithm designers have invented a large number of algorithms and heuristics. But when faced with a particular CSP, a problem solver may not know which algorithm or heuristic is the most appropriate to use to tackle this problem. The fact that the efficiency of an algorithm depends on the way in which a CSP is formulated makes the problem solver's task even more difficult. After a CSP has been formulated, knowing which algorithm and heuristic to use is quite difficult. For example, *lookahead* algorithms invest their effort to propagate constraints in order to reduce the chance of backtracking. *Intelligent backtracking* algorithms invest their effort to backtrack to the culprit decisions when backtracking is needed. Given a particular CSP, how should a problem solver choose between these two groups of algorithms? Is combining the two strategies always better?

This paper advocates that in order to help the problem solvers to solve their problems, research has to be done in systematically mapping CSPs to algorithms. Furthermore, we suggest that such a mapping can be done according to the problem's characteristics.

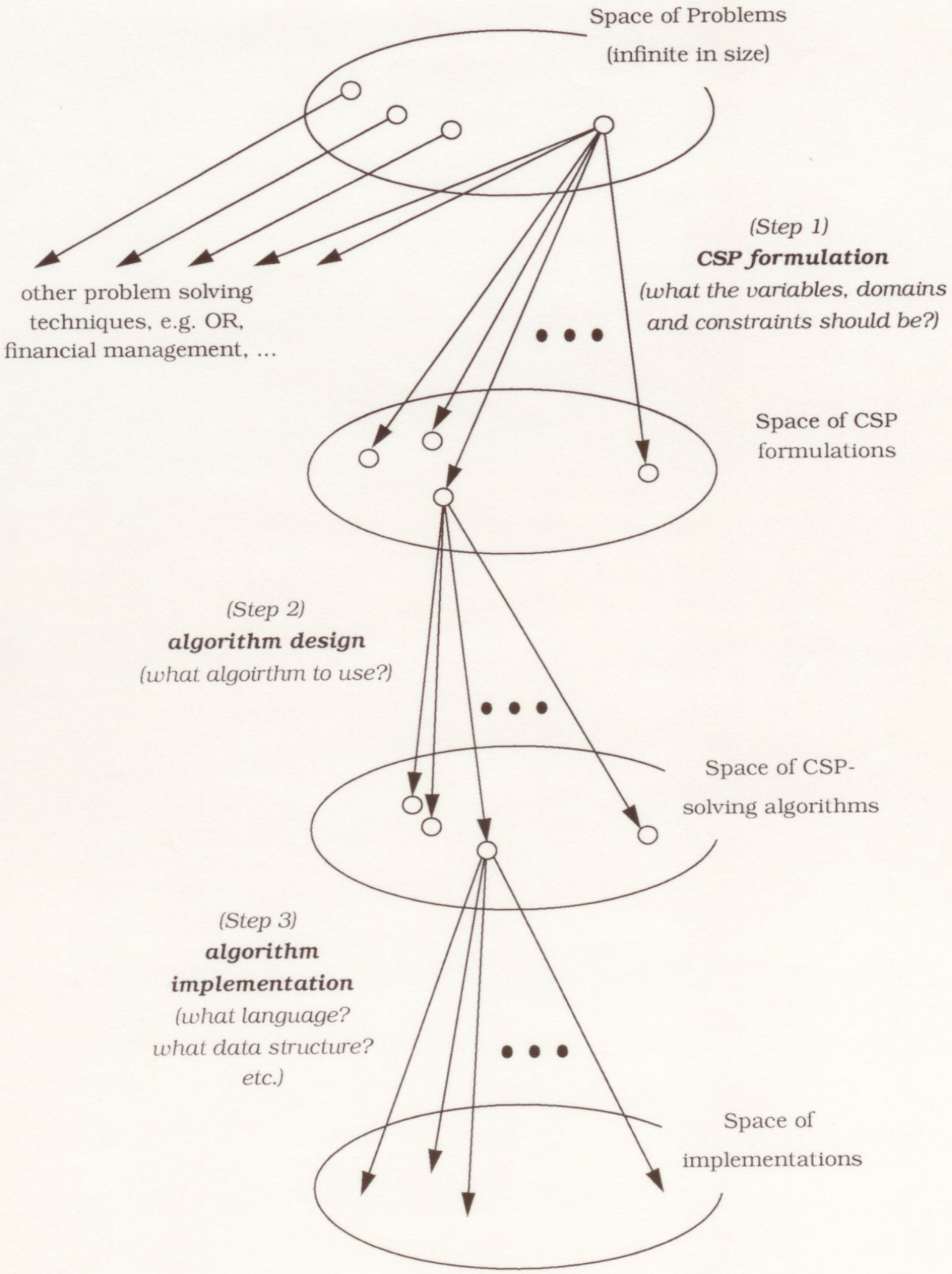


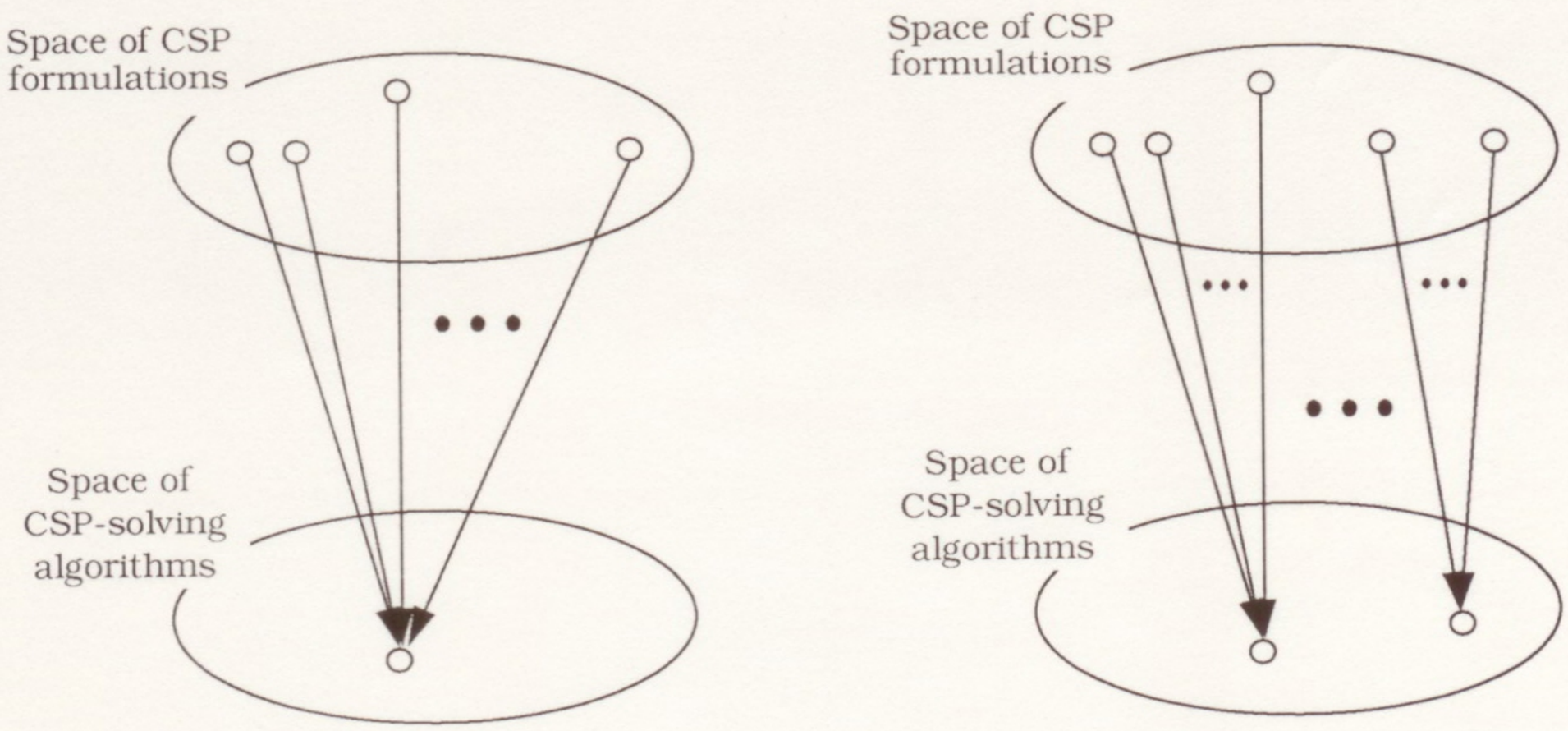
Figure 1 — Choice points in problem solving

2 The Problem Solver's Task

Given a problem, the problem solver has to search in a vast space: firstly, he/she has to search in the space of "tools" to solve it, and techniques in constraint satisfaction is only one of the many tools available. Having decided to use a constraint satisfaction approach, he/she has to decide on how to formulate the problem as a CSP, i.e. what the variables and their domains should be and how to choose among the many ways of specifying the constraints. Then he/she would have to choose an algorithm for tackling the CSP formulated — some algorithms are more efficient than others for his/her particular CSP. Finally, the algorithm has to be implemented. Figure 1 shows the space searched by the problem solver.

To build an efficient constraint satisfaction system, the problem solver often has to search more than one branch in the problem solving space. Among the steps shown in figure 1, anticipating which CSP can be solved more easily is arguably the most difficult one (apart from special cases). So problem solvers quite often have to backtrack to step (1) in figure 1. For example, one frequently asked question is whether adding redundant constraints (constraint which can be deduced from others) to a particular problem will help to solve it.

Much work in the past involved comparing algorithms against each other on randomly generated CSPs, e.g. Haralick & Elliott [1980], Nudel [1982] and Dechter & Pearl [1988]. General CSP-systems normally commit to one particular algorithm and heuristic (see figure 2). For example CHIP, Charme and PECOS mainly use the *forward checking* (FC) control strategy plus the *fail-first principle* (FFP) heuristic. This combination (call it FC+FFP) is used to tackle all the CSPs given to it as this combination has been found to be effective for many CSPs. However, like any other combinations of algorithms and heuristics, the efficiency of FC+FFP is dependent on the way in which the CSP is formulated. Therefore, users of these CSP-systems must learn how to formulate CSPs to suit their underlying strategy. Unfortunately, searching in the space of problem formulation is difficult by nature; so it is sometimes difficult to judge what is the best way to formulate the given problem as a CSP.



(a) Using one algorithm for all CSPs (all other algorithms are ignored)

(b) Mapping CSPs to different algorithms

Figure 2 — Two different ways to map problems to algorithms (refer to Step (2) in figure 1)

Moreover, by looking at the search space in figure 1, it is quite reasonable to question whether committing to a particular algorithm design (i.e. limiting the number of branches to one in step (2) in figure 1) could miss more efficient ways of solving a CSP. Indeed, there are quite a large number of algorithms in the literature which gain their efficiency by exploiting certain characteristics of CSPs. Users of the above mentioned CSP-systems are likely to miss the opportunity of applying such algorithms.[1] This will be elaborated later.

Table 1: Selected constraint satisfaction algorithms and heuristics that they might use

Sub-class		Algorithms	Abbr.
Complete algorithms			
General Algorithms		Chronological Backtracking	BT
		Iterative Broadening	IB
		Branch and Bound	B&B
Lookahead Algorithms (LA)		Forward Checking	FC
		Directional Arc-consistency Lookahead	DAC-L
		Directional Path-consistency Lookahead	DPC-L
		Arc-consistency Lookahead	AC-L
		Path-consistency Lookahead	PC-L
Gather Information While Searching Algorithms (GIWS)	Intelligent Backtracking Algorithms (INB)	BackJumping	BJ
		Graph-based BackJumping	GBJ
		Conflict-based Backjumping [Pros93]	CBJ
	other GIWS algorithms	Learning Algorithms	Le
		Backchecking, Backmarking	BC, BM
Solution Synthesis (SS)		Freuder's Solution Synthesis Algorithm	Fr
		Seidel's Invasion Algorithm	SI
		Essex Algorithms	ES
Examples of Speciallized Algorithms (which is quite impossible to exhaustively enumerate)		Tree Search Algorithm	TS
		Tree Clustering Method	TC
		AnalyseLongestPaths, AnalyseShortestPaths	ALP, ASP
Heuristics for Complete Search Algorithms			
Variables Ordering		Failed-first Principle	FFP
		Minimal Width Ordering	MWO
		Minimal Bandwidth Ordering	MBO
Values Ordering		Failed-first Principle	FFP
		Min-conflict Heuristic	MC
Stochastic Search Algorithms			
Local Search (LS)		Hill Climbing (e.g. Heuristic Repair, which uses the Min-conflict Heuristic)	HC (e.g. HR)
		Tabu Search [Glov89, 90]	TB
		Simulated Annealing [AarKor89][OttVan89]	SA
Genetic Algorithms [Gold89][Davi91]			GA
Connectionist Methods (e.g. GENET [WanTsa91] [TsaWan92])			CM

3 CSP Solving Algorithms and Heuristics, an Overview

A large number of algorithms and heuristics have been developed for CSP-solving. Table 1 lists a selection of those which property are reasonably well understood. This list is by no means exhaustive. References for those which have not been fully explained in Tsang [1993] are given in this paper. Following we shall briefly summarize the algorithms and heuristics in table 1.

Basically complete algorithms for CSP-solving can be classified into general search, *lookahead*, *gather-information-while-searching*, *solution synthesis* and algorithms which specialize on CSPs with special characteristics. Associated to each CSP is a *constraint graph*, which is a graph with each node representing a variable in the CSP and each edge between two nodes representing the fact that, in some way, the two variables represented by those nodes are involved in some common constraints. Many CSP-solving algorithms exploit the topology of the CSP's constraint graphs.

The most commonly used general algorithms is probably *chronological backtracking* (BT), which is an uninformed search. *Iterative broadening* (IB) attempts to spread its search effort over different branches in order to increase the chance of finding the first solution more quickly than BT. *Branch and Bound* (B&B) is a general algorithm for finding optimal solutions.

Lookahead algorithms attempt to recognize the need for backtracking at an earlier stage by propagating constraints as soon as each label is committed to. Algorithms used for propagating constraints are described as *problem reduction* algorithms. Although problem reduction alone is normally insufficient for solving a CSP, it is useful as a *pre-processing* (PP) step and/or in lookahead algorithms. Different lookahead algorithms use different problem reduction algorithms, and some lookahead algorithms invest more effort in problem reduction than others.

Gather-information-while-searching algorithms include *intelligent-backtracking* algorithms (INB) and algorithms that *learn*. Better known intelligent backtracking algorithms include *BackJumping*, *Graph-based BackJumping* and *Conflict-based BackJumping*.[2] General learning algorithms often incorporate *truth maintenance systems* (TMS) [SmiKel88] [DeKl89].

Solution syntheses algorithms attempt to constructively compose solution tuples instead of searching for them. This is normally done by considering compound labels for larger and larger sets of variables. To improve efficiency, *Freuder's solution synthesis algorithms* propagate constraints, *Seidel's invasion algorithm* exploit the topology of the constraint graph, and the *Essex algorithms* are designed to make use of parallel machine architecture.

Some algorithms attempt to exploit the topology of constraint graphs. By doing so, some of them may be able to contain the combinatorial explosion problem in CSP-solving. For example, if the constraint graph of a CSP forms a tree, then the *tree search algorithm* (TS) can be applied to solve it in polynomial time without any need of backtracking. If the constraint graph can be partitioned into disconnected sub-graphs, then the CSP can be divided into sub-problems which can be solved independently, i.e. by applying the *divide and conquer* (D&C) strategy.

The above control strategies may be helped by heuristics. The most useful type of general heuristics for CSP-solving are heuristics for ordering the variables and values in a search. Among the variable ordering heuristics, the *minimal width ordering* (MWO) heuristic attempts to reduce the need for backtracking; the *minimal bandwidth ordering* (MBO) heuristic attempts to reduce the distance of backtracking when backtracking is needed; the *fail-first principle* (FFP) attempts to help the algorithms to recognize situations in which backtracking is necessary so that backtracking can take place at an earlier state. Value ordering heuristics such as the FFP and the *min-conflict* (MC) heuristics attempt to label variables with values which are most likely to succeed first so as to reduce the chance of backtracking.

Because of the combinatorial explosion problem, many CSPs cannot be solved by *complete* algorithms. Stochastic search methods sacrifice completeness for tractability. Stochastic search methods are search methods which contain some elements of randomness, and the search steps available at one state depends on the outcome of previous steps. Being incomplete, the usefulness of a stochastic search method in CSP-solving is evaluated by its

---

1. Similar view is held by Minton [1993] although he took on a different research direction. He attempted to learn which heuristics to use when faced with problems with similar characteristics are to be tackled repeatedly.
2. Recently Ginsberg [1993] presented a Dynamic Backtracking Algorithm, which properties are still under investigation.

Table 2: Relating CSP characteristics to algorithms and heuristics

Problem specification and characteristics		Algorithms and heuristics											
		Complete algorithms				Heuristics				Stochastic search			Other
		BT IB B&B	LA	INB	SS	FFP	M W O	M B O	MC	HC TB SA	GA	CM	
Solution(s) required	any solution required	most of these algorithms can cope with these problems, with some exceptions (see text)			×	✔	✔	✔	✔	✔			
	all solutions required				✔	✔	✔	✔	×	×			
	optimal solution required	most of these algorithms (except B&B) have to find all solutions, then compare them according to the objective function				variable ordering heuristics can help algorithms to find all solutions; value ordering is less useful (see text)				✔ useful when near-optimal solutions are acceptable			
Time available	not critical	✔ all the above algorithms are applicable				✔	✔	✔	✔	there is less incentive to use them			
	limited time	limited by the combinatorial explosion probem				✔ good heuristics may speed up complete algorithms				✔	✔	✔	
	real time / near real time	× most of these algorithms cannot solve useful problems of this category				heuristics may help, but unlikely to speed up complete algorithms dramatically				✔	×	✔	
Problem Tightness	very loosely constrained	✔	×		×	application not solely determined by the tightness of the problem				different algorithms may suit different types of problems			
	very tightly constrained	×	✔		✔								
Characteristics of the CSP's primal graph (PG)	the PG is unconnected									these characteristics may or may not be exploited by the above stochastic methods			D&C
	the PG is a tree						✔						TS (PP)
	degrees of nodes vary significantly			✔			✔						
	PG has small bandwidth			×				✔					
Other problem characteristics	some variables and values are more constrained than others		✔ likely to be useful			✔			✔				

Keys: ✔ means the algorithm/heuristic should be considered; × means it should not (e.g. inapplicable or ineffective); blank means the problem specification or characteristic is not a deciding factor; PP = Pre-processing

speed and its successful rate in finding solutions. Some of the best known stochastic search methods are *hill climbing* (HC), *tabu-search* (TB), *simulated annealing* (SA), *genetic algorithms* (GA) and *connectionist methods* (CM).

4 Relating CSP Characteristics to Algorithms and Heuristics

We believe that whether an algorithm or a heuristic is efficient or not depends on the characteristics of the given CSP. Therefore, we believe that it is worth finding out the *domain* of each algorithm and heuristic; i.e. what characteristics must a CSP have if a particular algorithm or heuristic is applicable to it or is efficient in solving it? Table 2 relates characteristics of the CSPs to algorithms and heuristics. Table 2 suggests that some algorithms are more useful in certain types of problems. By describing algorithms by their categories, table 2 inevitably over-generalizes certain observations.[3] Detailed justifications behind each entry of this table would take up too much space here. Many of the entries are based on analysis in the literature and analysis by the authors. Following we shall justify some of the entries.

In general, BT and all LA and INB algorithms are useful for finding first or all solutions. When optimal solutions are required, these algorithms have to find all solution tuples and compare them according to the objective function. IB is designed for finding the first solution, and B&B is designed for finding optimal solutions.[4] Solution synthesis algorithms in general are only cost effective for finding all solutions.

Most variable ordering heuristics can be used by complete search methods. The motivation behind ordering the values in a search is to increase the chance of finding a solution at an earlier attempt (by searching the branch which is most likely to succeed first). When all solutions are required, value ordering heuristics are unlikely to be useful (except for algorithms that learn). When optimal solutions are required and the objective function is relevant to the number of constraints violated, then MC may help B&B to find tight bounds.

Stochastic search methods are incomplete in nature. Therefore, they are in general not suitable for finding all solutions. These methods are used for optimization problems in general. When used for finding single solutions in CSPs, they normally start with one or more sets of compound labels (with one assignment for each variable in the CSP), and then attempt to reduce the number of constraints being violated by them.

For many real life problems, one has no choice but to give up completeness for speed. This is where stochastic search methods come in. HC, TB and SA have wide domains of application. Their success relies mainly on appropriate representations and good hill climbing heuristics. The success of a tabu search also relies on the way in which the tabu list is manipulated. Since there is no limit on how complicated such manipulation is, tabu search has the potential to perform very well (though the same can be said about some other general methods such as production systems). GAs normally require a substantial number of iterations, hence a nontrivial (but tractable) amount of time to be effective. Therefore it is not suitable for real time or near real time applications. A GA's performance depends on the representation and operators that it uses as well as the setting of parameters such as the population size and mutation rate. By taking advantage of parallel architectures, connectionist approaches (such as GENET) could possibly provide a means to solve CSPs in real time or near real time, though its potential is yet to be analysed. All these issues could affect the choice of stochastic search methods under any particular circumstances.

One of the most important characteristics of a CSP is its tightness, which is only a relative measure. It can be measured by the number of solutions over the grand product of the domain sizes (which is the total number of combinations in assigning values to all variables). Although this ratio is normally unknown in reality, it is possible to estimate it. When a problem is loosely constrained, it can be solved easily by many algorithms. When a problem is extremely tightly constrained or over-constrained (i.e. no solution exists), many combinations of assignments are illegal. This in fact may help LA algorithms to search efficiently (by pruning off large parts of the search space). SS algorithms are also more suitable for tightly constrained problems because there would be fewer legal compound labels to store.

Whether INB algorithms are effective or not depends more on the topology of the given CSP than its tightness.

---

3. In particular, the three "general algorithms" are quite different in their application domains. A table which gives finer mapping has been compiled by the authors, and will be reported in the near future.

4. The authors have experimented combining LA algorithms with B&B for CSPs for which optimal solutions are required and the objective function is to find the assignments to the maximum number of variables without violating any constraints.

The same applies to heuristics for variables ordering. If some variables are constrained by only a small number of other variables, then INB algorithms are more likely to be useful. If variables can be ordered so that all variables which constrain each other are closely placed to each other, then effort spent on finding the culprit may not be justifiable.

As mentioned in the previous section, when a CSP possesses certain characteristics, it could be solved efficiently by specialized algorithms (such as D&C and TS). Characteristics such as the type of variables, domains and constraints, can also be exploited. For simplicity, this has not been elaborated in table 2.

5 The Way Forward

Table 2 defines the domain of individual algorithms and heuristics — where they are applicable or particularly useful. It only gives an outline of the algorithms and heuristics which could be useful for a CSP with particular characteristics. A ✔ suggests that the corresponding algorithm or heuristic is relevant to a CSP with a particular characteristic; it does not, however, suggest that this algorithm or heuristic must the most efficient or effective one for problems with such characteristics. (Whether an algorithm or heuristic is efficient or effective or not often depends on more than one factor.) For example, table 2 does not suggest what we should do with a CSP which is tightly constrained, with small bandwidth in the constraint graph in which the degrees of the nodes vary significantly. Should we use FC or INB algorithms? Should we use the MBO or the MWO heuristic?

What we really would like to do is to identify a set of problem characteristics which can be used to index the problem to relevant algorithms and heuristics. Ideally, one would like to be able to use them to retrieve the most efficient algorithm(s) and effective heuristic(s) for that particular type of problems. Figure 3 shows an example decision tree.

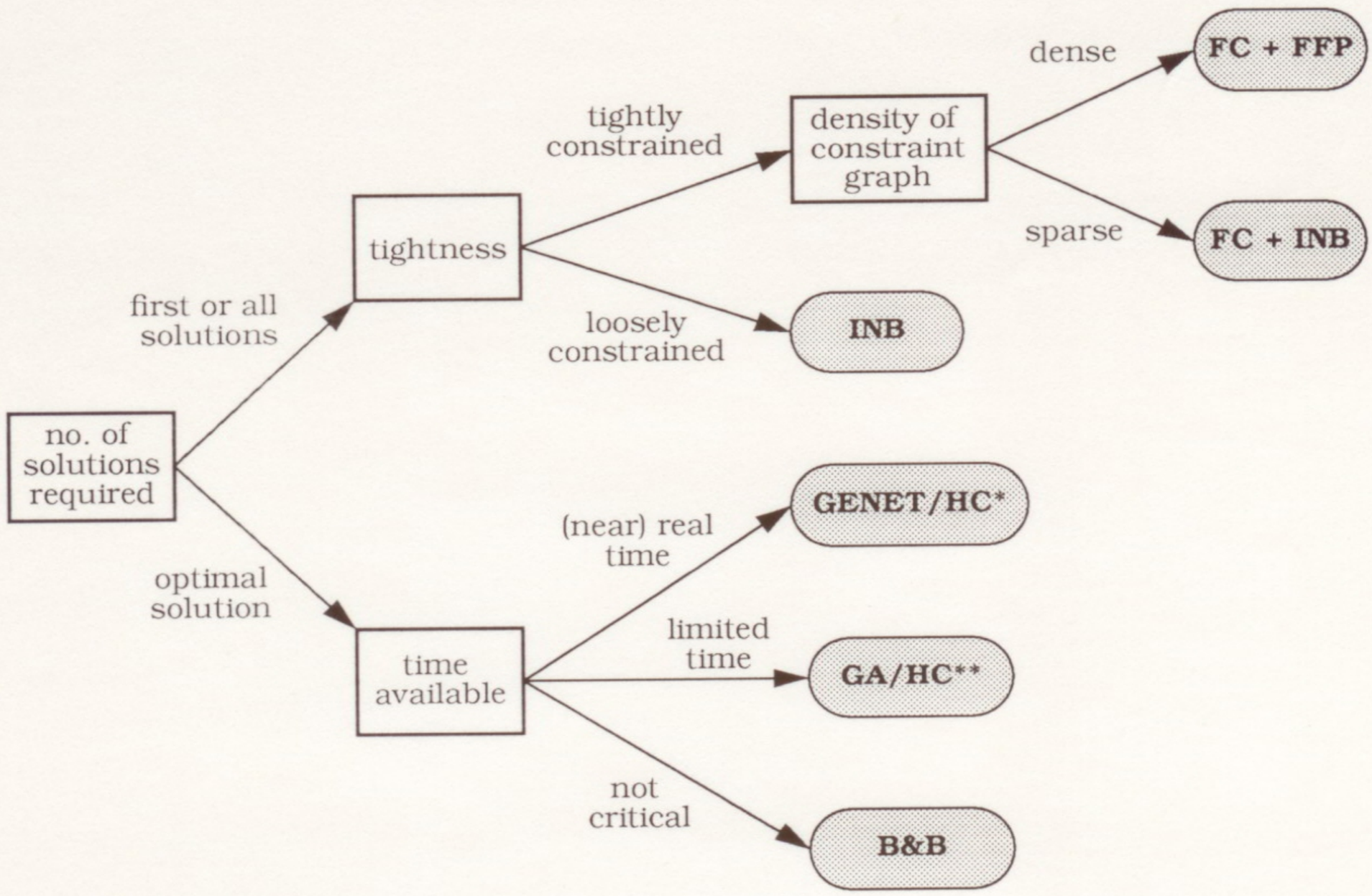


Figure 3 — Example of a crude decision tree (* whether HC is used depends on factors such as the availability of suitable heuristics for HC; ** whether GA is used depends on the available of suitable representation and appropriate GA operators for the particular problem; for simplicity in presentation, additional branches are not created in this decision tree)

The decision tree shown in figure 3 is very simple and is only used to illustrate the form of an indexing system that one could build. The tree itself represents a plausible but naive indexing strategy. Basically this decision tree suggests that if the CSP requires the first solution or all solutions to be found, then we should look at its tightness. If the CSP is relatively tightly constrained, then we should look at the number of edges in the constraint graph. If the constraint graph is dense, then this decision tree suggests to use the forward checking algorithm plus the fail-first principle (FC+FFP). A possible justification for this is that constraint propagation is more likely to have some effect in reducing a CSP with such characteristics. If the number of edges in the constraint graph is small, then it is worth spending some time to analyse the culprit when backtracking is needed, hence the indexing to a hybrid forward checking and intelligent backtracking algorithm (FC + INB). If the CSP is loosely constrained, then it is not worth spending time to propagate constraints when a label is committed to, but it is still worth trying to identify the culprits when backtracking is needed. If an optimal solution is required for the CSP, then this indexing strategy would look at the time available.[5] GENET is built for real time or near real time applications. GA is more applicable when it is given more time than a few seconds. Both GENET and GA could be replaced by HC should useful hill climbing strategies or heuristics be available.[6]

One important fact that we would like to point out is that there is no need to find the *optimal* decision tree before it can be used in practice. As long as the tree indexes correctly to efficient algorithms for CSPs characterised under each branch, it will help us to solve the problems more efficiently than applying a single algorithm for all problems (which most existing constraint satisfaction systems do).

For a problem characteristic to qualify for the indexing purpose, it should satisfy the following criteria:

(1) it matters — whether a problem possesses such characteristic or not *does* matter in what algorithm are likely to be able to solve this problem efficiently;
(2) it must be reasonably easy to recognize — the marginal gain in being able to apply a more efficient algorithm because of the presence of this characteristic must out-weight the effort spent on recognizing it; that means this characteristic must not be too difficult to recognize.

Based on point 2, any decision tree can be refined to allow more detailed classification of CSPs. However, the gain in being able to index efficient algorithms must out-weight the cost of indexing. Whether the characteristics in Table 2 meet these criteria is to be examined but outside the scope of this paper. Besides, the order in which the characteristics are placed in the decision tree matters. Given a fixed set of characteristics to be considered, certain orderings may lead to a tree which allows faster indexing.

A decision tree is also useful for guiding future research: it highlights CSPs for which no existing algorithm is efficient; so new algorithms for CSPs with certain characteristics are worth developed. Besides, any new algorithms or heuristics developed must find their positions in some leaf nodes in some useful decision trees if they were to be claimed useful. Each leaf node in a decision tree depicts a type of CSPs. To evaluate the efficiency of a new algorithm, we must compare it with algorithms which are useful for the same types of CSPs for which this new algorithm is designed. Besides, comparison must be made on CSPs of the intended types only. So if algorithm X is designed for tackling CSPs which have property P, one should only compare with other algorithms which are designed for CSPs with property P. If we evaluate algorithm X on randomly generated CSPs, then we must exclude those which do not have the property P.

Our discussion so far still leaves the question of how to formulate CSPs (Step (1) in figure 1) unanswered. Work must be done to develop guidelines to formulate CSPs. But a decision tree can at least provide targets for CSP formulation: given a decision tree, the problem solver can compare different formulations based on their characteristics which appear in the decision tree: efficient algorithms may be available for some leaf nodes of the decision tree but not for others.

6 Conclusion

In this paper, we have pointed out the gap between CSP algorithm designers and CSP solvers. We have advocated that research must be done to bridge this gap. We suggest that problem solvers may use decision trees for indexing algorithms for solving their individual problems and this mapping can be done according to the problems'

5. Of course there is no reason why one should not look at the "time available" as well when the first or all solutions are required. This is not included here because we would like to keep the tree simple for presentation purpose.
6. Again, there is no reason why simulated annealing and tabu search should not replace HC under certain circumstances.

characteristics. Such decision trees should also help algorithm designers to identify useful research areas and to evaluate any algorithms or heuristics that they may design. We have laid the foundation of such work by (a) listing and classifying some of the better documented algorithms; and (b) producing a crude mapping from CSP characteristics to algorithms and heuristics. A lot more work needs to be done to refine the above mapping and to develop useful decision trees.

References

[AarKor89] Aarts, E. & Korst, J., *Simulated Annealing and Boltzmann Machines*, John Wiley & Sons, 1989

[Davi91] Davis, L. (eds.), *Handbook of genetic algorithms*, Van Nostrand Reinhold, 1991

[DecPea88a] Dechter, R. & Pearl, J., *Network-based heuristics for constraint-satisfaction problems*, Artificial Intelligence, Vol.34, 1988, 1-38 (A slightly revised version appears in Kanal, L. & Kumar, V. (eds.), *Search and Artificial Intelligence*, Springer-Verlag, 1988, 370-425)

[DeKl89] de Kleer, J., *A comparison of ATMS and CSP techniques*, Proceedings International Joint Conference on AI, 1989, 290-296

[Gins93] Ginsberg, M., *Dynamic Backtracking*, Journal of Artificial Intelligence Research, Vol.1, 1993, 25-46

[Glov89] Glover, F., *Tabu search Part I*, Operations Research Society of America (ORSA) Journal on Computing, Vol.1, 1989, 109-206

[Glov90] Glover, F., *Tabu search Part II*, Operations Research Society of America (ORSA) Journal on Computing 2, 1990, 4-32

[Gold89] Goldberg, D.E., *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley, 1989

[HarEll80] Haralick, R.M. & Elliott, G.L., *Increasing tree search efficiency for constraint satisfaction problems*, Artificial Intelligence, Vol.14, 1980, 263-313

[Mint93] Minton, S., *An analytic learning system for specializing heuristics*, Proc., 13th International Joint Conference on AI, 1993, 922-928

[Nude82] Nudel, B.A. *Consistent-labeling problems and their algorithms*, Proceedings National Conference on Artificial Intelligence (AAAI), 1982, 128-132

[OttVan89] Otten, R.H.J.M. & van Ginneken, L.P.P.P., *The annealing algorithm*, Kluwer Academic, 1989

[SmiKel88] Smith, B. & Kelleher, G. (ed), *Reason maintenance systems and their applications*, Ellis Horwood, 1988

[Pros93] Prosser, P., *Hybrid algorithms for the constraint satisfaction problem*, Computational Intelligence, Vol.9, No.3, 1993, 268-299

[Tsan93] Tsang, E.P.K., *Foundations of constraint satisfaction*, Academic Press, 1993

[TsaWan92] Tsang, E.P.K. & Wang, C.J., *A generic neural network approach for constraint satisfaction problems*, in Taylor, J.G. (ed.), *Neural network applications*, Springer-Verlag, 1992, 12-22

[WanTsa91] Wang, C.J. & Tsang, E.P.K., *Solving constraint satisfaction problems using neural-networks*, Proceedings, IEE Second International Conference on Artificial Neural Networks, 1991, 295-299

Appendix A: Abbreviations used in this paper

AC-L	Arc-consistency Lookahead
ALP	AnalyseLongestPaths Algorithm
ASP	AnalyseShortestPaths Algorithm
B&B	Branch and Bound
BC	Backchecking
BJ	BackJumping
BM	Backmarking
BT	Chronological Backtracking
CBJ	Conflict-based Backjumping
CM	Connectionist Methods
D&C	Divide and Conquer
DAC-L	Directional Arc-consistency Lookahead
DPC-L	Directional Path-consistency Lookahead
ES	Essex Solution Synthesis Algorithm
FC	Forward Checking
FFP	Failed-first principle
Fr	Freuder's Solution Synthesis Algorithm
GA	Genetic Algorithms
GBJ	Graph-based BackJumping
GIWS	Gather-information-while-searching Algorithms
HC	Hill Climbing
HR	Heuristic Repair Method
IB	Iterative Broadening
INB	Intelligent Backtracking
LA	Lookahead Algorithms
Le	Learning Algorithm
LS	Local Search
MBO	Minimal Bandwidth Ordering
MC	Min-conflict Heuristic
MWO	Minimal Width Ordering
PC-L	Path-consistency lookahead
PP	Pre-processing
PR	Problem Reduction
PG	Primal Graph
SA	Simulated Annealing
SI	Seidel's Invasion Algorithm
SS	Solution Synthesis
TB	Tabu Search
TC	Tree Clustering Method
TMS	Truth Maintenance Systems
TS	Tree Search Algorithm