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.

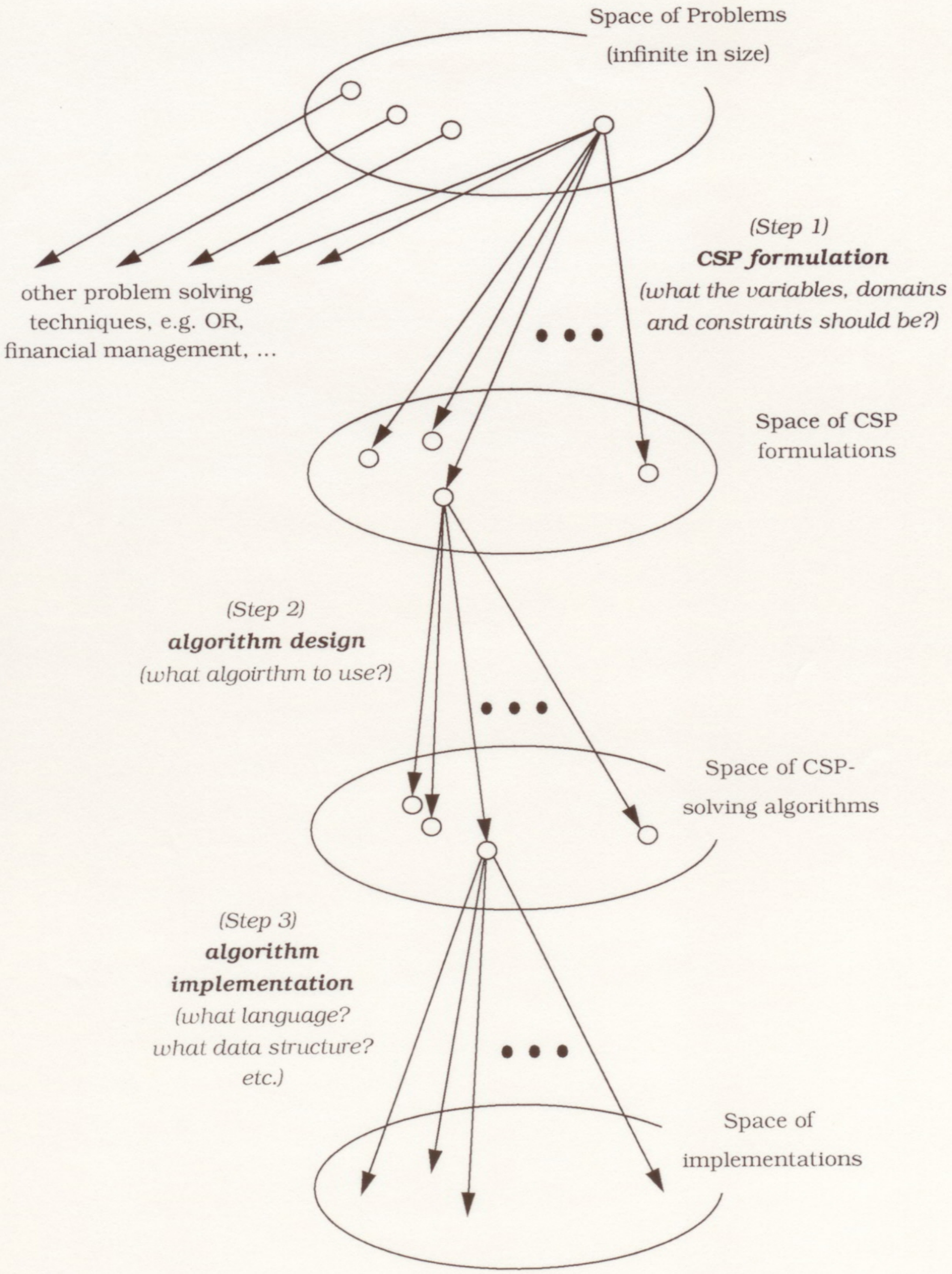


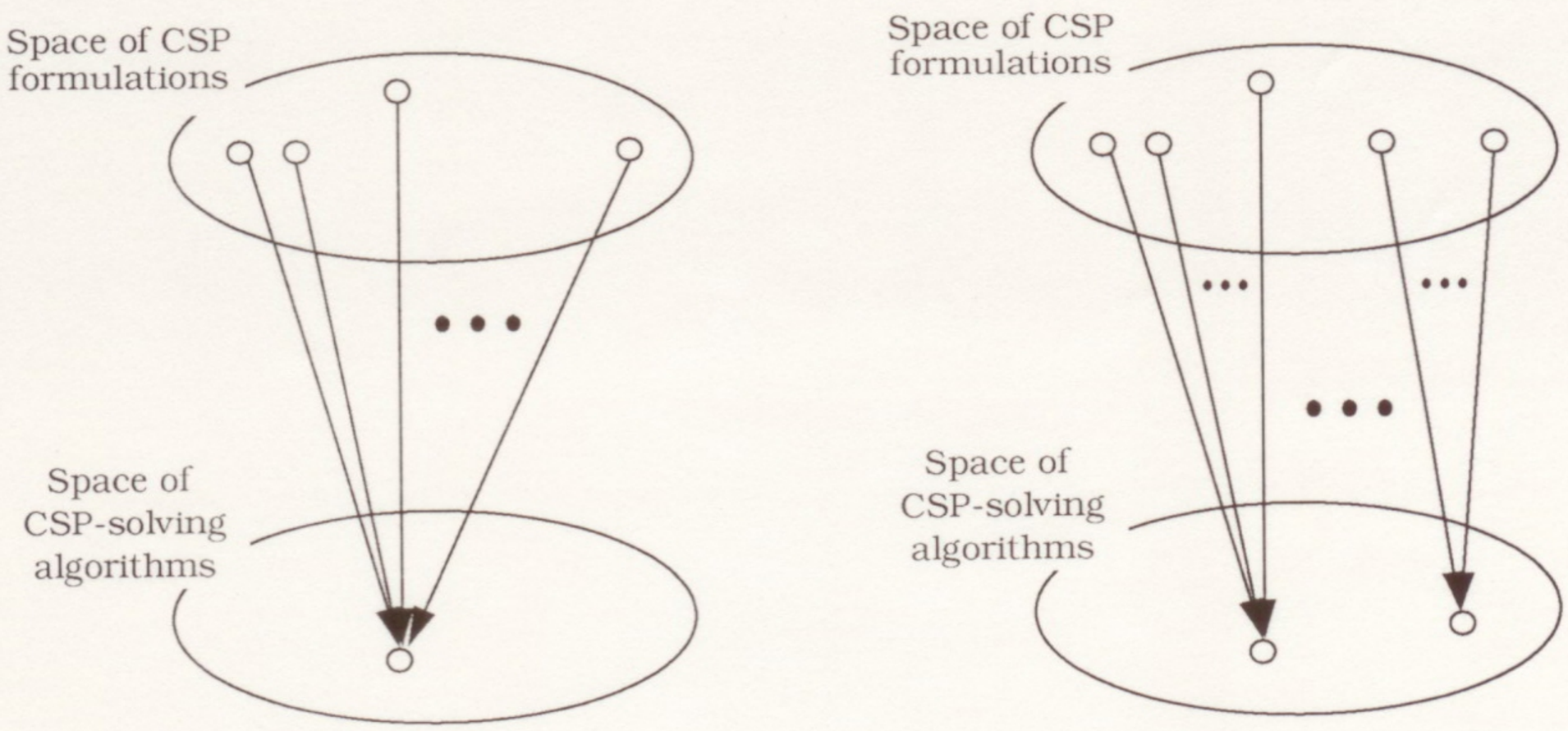
Figure 1 — Choice points in problem solving

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.