Constraint Satisfiability Algorithms for Interactive Student Scheduling

Ronen Feldman

Department of Mathematics and Computer Science
Bar-Ilan University
Ramat Gan, Israel

Martin Charles Golumbic

IBM Israel Scientific Center
Technion City
Haifa, Israel

Abstract

A constraint satisfiability problem consists of a set of variables, their associated domains (i.e., the set of values the variable can take) and a set of constraints on these variables. A solution to the CSP is an instantiation (or labeling) of all the variables which does not violate any of the constraints. Since constraint satisfiability problems are, in general, NP-complete, it is of interest to compare the effectiveness and efficiency of heuristic algorithms as applied, in particular, to our application.

Our research effort attempts to determine which algorithms perform best in solving the student scheduling problem (SSP) and under what conditions. We also investigate the probabilistic techniques of Nudel for finding a near optimal instantiation order for search algorithms, and develop our own modifications which can yield a significant improvement in efficiency for the SSP. Finally, we assign priorities to the constraints and investigate optimization algorithms for finding schedules which rank high with respect to the priorities. Experimental results have been collected and are reported here. Our system was developed for and used at Bar-Ilan University during the registration period, being available for students to construct their timetables.

1. Introduction

Techniques for solving constraint satisfiability problems[1] (CSPs) have received much attention in artificial intelligence, operation research and symbolic logic. Applications which may be viewed as CSPs are found in scene identification in computer vision, space and motion planning, database consistency, combinatorial optimization, and cryptarithm puzzle solving. Our research has dealt with algorithmic aspects of solving CSPs and using these methods for interactive scheduling in the Academic Planning Environment expert system project [8], a system which provides advising and assistance to university students in the planning of their program of studies.

A *constraint satisfiability problem* (CSP) is composed of a set of *variables* $V = \{v_1, ..., v_n\}$, their related *domains* $D_1, ..., D_n$ and a set of *constraints* on these variables. (The domain of a variable is a group of values to which the variable may be instantiated.) The domain sizes are $m_1, ..., m_n$, respectively, and we let m denote the maximum of the m_i. Each constraint C_j is *relevant* to a subset of variables $v_{j,1}, ..., v_{j,k_j}$ and may be regarded as containing all the tuples of this set of variables that are legal with respect to C_j, that is, $C \subseteq D_{j,1}, ..., D_{j,k_j}$. A constraint which is relevant to exactly one variable is called a *unary constraint*. Similarly, a *binary constraint* is relevant to exactly two variables.

A solution to the CSP consists of an instantiation of all the variables which does not violate any of the constraints, i.e., a consistent labeling of each variable with a value from its domain. Computationally, constraint satisfiability problems are NP-complete, which implies that there are no known polynomial time algorithms which can guarantee finding a solution. Therefore, it is of interest to compare the effectiveness and efficiency of various heuristic algorithms for solving CSPs as applied, in particular, to our application.

Because of the potentially exponential search that such algorithms will execute in order to find a solution, there are a number of techniques for improving its performance by statically or dynamically pruning the search space [1, 9, 10, 13]. For example, constraints which are relevant to small a subset of variables can be used to throw out, in a single consistency check, a much larger group of possible tuples (all the extensions of the rejected tuple). In section 3, we will formulate the most interesting and effective of these algorithms from the scheduling system point of view investigate their performance characteristics.

Our research effort attempts to determine which algorithms perform best in solving the student scheduling problem and under what conditions. Since the order in which variables are assigned values significantly affects the performance of such algorithms, we investigate the probabilistic techniques

1 Constraint satisfiability problems are also known in the literature as *consistent labeling problems*

of Nudel [12] for finding a near optimal instantiation order and develop our own modifications which can yield a significant improvement in efficiency for the SSP. This is presented in section 4.

In a practical scheduling system is generally the case that some constraints *must* be satified while others can be relaxed according to specified priorities. This implies a ranking on the quality of all valid schedules. Therefore, in section 5 we investigate optimization algorithms for student scheduling under prioritized constraints.

2. The student scheduling problem as a CSP

The student scheduling problem (SSP), which we present in this section, can be looked at as a variation of a CSP. Thus, we will be able to apply the algorithms and theory of CSPs to help in solving the SSP.

Associated to each course at a university is a set of *offerings* where "taking a course" means registering for one of its offerings. An offering consists of a number of *meetings* which the student must attend. For example, 88.101.5, the fifth offering of course 101 in department 88, might be represented by

MF 10-11	semester 1,2	lecture	prof.golumbic
Th 16-17	semester 1	recitation	mr.feldman
T 14-17	semester 2	laboratory	dr.schild

Two offerings *conflict* if their time intervals intersect. We assume here that time intervals do not include their endpoints, (e.g., W 11-12 does not conflict with W 12-13.) These conflicts constitute the *implicit constraints*. The basic unit of the SSP, called a *group*, consists of a list L_i of courses and lower and upper bounds, $\ell_i \le u_i \le |L_i|$ indicating the minimum and the maximum number of courses to be taken from the list. A group is *strict* if $\ell_i = u_i = |L_i|$. If the student wants to force the scheduler to schedule a particular course, he can do so by declaring a strict group.

The input to the SSP will be a disjoint list of groups, called the *requirements*, a timetable of offerings (as published by the university), and a set of explicit constraints and priorities to be described below. A successful output will be a set of offerings which satisfies the requirements and all implicit and explicit constraints.

We regard each course as a variable whose set of offerings is its domain. If the requirements are all strict groups, then our problem would be to find a consistent labeling of the set of all the courses included in the requirements, i.e., the union of all the course lists of the groups. But if the requirements include non-strict groups, then there will be courses that need not be instantiated in a solution to the problem. In this sense, one may either regard the SSP as a slight generalization of the regular CSP to allow for uninstantiated variables, or simply add a "dummy" offering to each domain and incorporate the number of dummy instantiations in a group with the lower and upper bound constraint checks.

Priorities and constraints: In addition to the implicit constraints defined above, we also allow certain "user defined" *explicit constraints*. In this paper we define a *priority* to be any number in the range $-P,\ldots,0,\ldots,P$. The actual *explicit constraints* are those which have priority P or $-P$ and all other priorities are simply recommendations to the scheduler. The scheduler uses those priorities to indicate which schedule will be preferred over another by the student. This is particularly important when there is no solution, and the system attempts to relax the explicit constraints or adjust the groups in order to find a good partial solution. Each of the following 2 types of objects can have a priority assigned to it:

- **Fixed time constraints** -- Each hour in the week is assigned a priority indicating how important it is for the student to learn or not to learn in this hour. Any hour with priority $-P$ is completely blocked, while any hour with priority P must be scheduled.
- **Non-fixed time constraints** -- An hour range (start hour to end hour) and a number of days in a single week may be assigned a priority. This differs from the fixed time constraints in that it leaves the actual days to be blocked or preferred unspecified.

There is one additional type of constraint which is not assigned a priority, namely the

- **Hour load constraints** -- The minimum and maximum number of hours in the week, for each semester, that the student wants to learn.

The hour load constraints together with the fixed and non-fixed time constraints constitute the all the *explicit constraints*.

3. Solution techniques for student scheduling problems

There are two opposite approaches which have been developed for solving CSPs. The first approach is called *filtering* which works by preprocessing the domains using constraint propagation in order to limit the search space and thereby reduce the time taken to find the first solution. The second approach is a *tree-search* which starts by giving some value to the first variable, then to the second and so on and if we *fail* then we *backtrack*, often in some intelligent manner. A third, *hybrid* approach, combines features of both of these techniques and includes such methods as the *forward checking algorithm*. variations which perform *full* or *partial look ahead*, and the *word-wise forward checking algorithm* (see [9, 13]).

3.1 Stages of the SSP

We can divide the procedures and algorithms for solving an SSP into four stages:

(Stage I) **Preprocessing before starting the search.** Variable (or node) consistency against the fixed time constraints and very tight non-fixed time constraints having priority $-P$ is carried out since checking these unary constraints is very cost effective, and can reduce the search space dramatically. After this preprocessing is done, these constraints may be ignored.

(Stage II) **Checks done during the search to choose the next instantiation.** In order to enhance the efficiency of how our predicates operate, we use bit masks extensively for the actual data types in our implementation. Each offering is represented by an *offering mask* indicating the semester and hours during the week in which it is active. Two offerings overlap if the ***bit-and*** between their masks is 1, i.e., there is at least one time conflict between their meetings on a certain day at the same hour. In addition, for each semester we maintain a *semester mask* of all courses that have been scheduled so far in that semester (the ***bit-and*** of all the offerings selected).

As we try to instantiate the next variable (called the *current course*), we perform the following three checks in order to make extensive cuts in the search: (1) *The overlapping check*: a ***bit-and*** check between each offering mask of the current course and the semester masks, (2) *The hour load check*: verifying that we have not yet scheduled more hours than the top limit, and (3) *Negative non-fixed time constraint checks* (with priority $-P$): Each non-fixed constraint has a hour range which is coded into a mask and a specified number n of days and is matched against the semester masks.

(Stage III) **Procedures done in each stage of the search process after a new instantiation is made.** Here there are two kinds of procedures: (1) *updating the domains of the future variables,* and sorting them into increasing order by domain size as the sizes of their domains change, and (2) *checking that we still can relax the positive priority constraints,* since it is preferable to find out as soon as possible if they cannot be relaxed.

(Stage IV) **Checks done against the final schedules.** Two final validating checks must be performed here on each candidate schedule, namely, (1) verifying that the bottom limit of the hour load constraint has not been violated, i.e., that enough hours have been scheduled, and (2) checking whether all the positive constraints have been satisfied.

3.2 Algorithms for solving the student scheduling problem

Minor modifications must be made to standard CSP algorithms since, as we noted in section 2, the SSP is slightly more general than a regular CSP. In the case of a CSP, when there is no way to satisfy a given variable, we reach a "dead end" in the search for a solution. In the case of an SSP, however, if the variable is in a non-strict group, we leave it uninstantiated and continue the search in the event that satisfying other members of the group will be sufficient to satisfy the bounds associated with that requirement.

In this section and in section 4 we assume the all priorities are P or $-P$, i.e., we deal with only the actual constraints. The default criterion for selecting the next variable to instantiate is choosing the course with the smallest domain, and was used exclusively in [8]. In section 4, however, which deals with obtaining the best instantiation order, we will see better criteria for estimating the most "difficult" variable to instantiate, which can be chosen as an option.

Comparison between different CSP algorithms when used for solving the SSP, i.e., finding one schedule, have been carried out. In these experiments, we have compared three CSP algorithms for finding one schedule:

1. regular backtracking,
2. forward checking (sorting of the variables by increasing order of their domain sizes after each stage),
3. word-wise forward checking.

In the experiment we have generated random sets of 30 courses and then tried all three algorithms with the minimum number of courses (Min) varying between 30 and 1. For each algorithm, we measured the time in seconds it took to solve the problem. We have generated about 3000 problems and have computed the average time it took for each algorithm to solve a problem for each value of Min.

From the results shown in Figure 1, we see that the word-wise algorithm clearly appears to be better than the other two and regular backtracking is the worst. If we will look at Figure 2, however, we will see that when Min is low enough (e.g., ≤ 19), then the regular backtracking is the best. This phenomenon is caused by the overhead we have in both the word-wise and the forward checking algorithms. The overhead of the word-wise algorithm is larger than that of the forward checking algorithm. So we may conclude that when Min is low it is best to use the simple backtracking algorithm because the problem is so simple we do not need such a "weapon" like the word-wise algorithm. But when the problems become more complicated, word-wise can be 5 times better than regular backtracking.

4. Instantiation Order

4.1 Heuristic techniques

It is well known that the instantiation order of the variables affects the time it takes to find a solution to a CSP. Therefore, if we can find a method which can help in determining a good order for our problem, it will save us a substantial amount of computing effort in the search. But, this method must be computationally easy enough so as not to forfeit the benefits.

Intuitively, it would appear that the variable most difficult to be instantiated ought to be instantiated first, the second most difficult next, and so forth in decreasing order of difficulty. This criterion is called "most promising to fail first" in some of the literature. The justification of such a criterion is that the sooner we fail, the sooner we can stop an unproductive search and backtrack, thus avoiding unnecessary and unfruitful checks. Haralick [9] has proved that under a suitably defined mathematical notion of "difficulty", this strategy will be optimal for the standard backtracking algorithm. Unfortunately, it is usually impossible to find such a parameter which fully captures "difficulty".

So our aim would be to select which variable is the most "difficult" to instantiate using a heuristic approach. The most obvious heuristic is to choose the variable with the smallest domain. We call this *domain-size sorting* and it is carried out a priori immediately following filtering by unary constraints (variable consistency). This well known criterion was found to be useful in [8] because of its simplicity and efficiency in providing substantial benefit over random orderings. In general, however, it is not a good estimate of the optimal order, apparently because it uses very little information about the problem itself. For example, no use is made of the relationships and interconnection between offerings.

A *dynamic* approach, determining the instantiation order as the search proceeds, was proposed for standard backtracking in [1] and analyzed in [13]. Ideally, we might wish to apply such a dynamic approach to the forward checking algorithms (as described here in section 3.2) by always choosing the uninstantiated variable whose *current* domain is smallest. As attractive as this may seem for cutting the search space, in some applications the computational complexity of constantly updating the domains in forward checking can make it infeasible.

4.3 Probabilistic analysis of the instantiation order

Nudel [12] has studied other criteria for measuring the difficulty of the variables by *estimating* the number of required consistency checks in the forward checking algorithms. In particular, he analyzed the influence of the *inter-variable* compatibility for choosing an instantiation order in arbitrary CSPs with binary constraints.

In our investigations, we have compared various strategies, including those of Nudel, as they apply to the SSP. In [4] we describe these approaches for selecting the most difficult variable, including our own approach, analyze their expected behavior and present our basic experimental findings. We will summarize this briefly here.

Nudel's investigations are based on the transition probabilities which are computed or approximated from information on the particular problem. He advocates using 2 levels of information. The first level (called *level-1*) uses the compatibility probabilities $R_{i,j}$ between the variables v_i and v_j and the initial sizes m_i of the domains. The second level (*level-2*) assumes that you also use, for each two variables, the number of pairwise compatible instantiations $I_{i,j}$ between them. The main difference between the levels is the statistical model we are adopting. For level-1 we have the binomic distribution model and for level-2 the hypergeometric distribution model. The level-2 model declares a finer partition of the CSP's than the level-1 model. (The partition is the classes of CSP's that have according to the model a common ordering scheme, i.e., all the relevant details of the problems are identical.)

If we want to use level-1 and level-2 for the general SSP when the groups are not strict, a few changes must be made since we do not schedule all the variables. This can be done using probability theory and yields a formula that is very complicated and lengthy to compute, (see [4]). Therefore, we have developed and used methods which require a much more simple calculation based on the same set of probabilities. Our *hill climb past (HC-p)* (resp., *hill climb future (HC-f)*) algorithm uses the information about the inter-relation between the candidate and the past (resp., future) variables.

An alternate approach is to use the inter-relations between the candidate and both the past variables and the future variables. The past variables are responsible for reducing the domain of the candidate, and the inter-relations between the candidate and the future variables are a good estimate for reducing the domains of the future variables' domains. Thus, our second criterion, using both of these factors, called the *inter-relations* or *IR algorithm*, and requires even less computation effort than either HC-p or HC-f.

4.4 Comparison between instantiation order algorithms – experimental results

Experiments have been performed to investigate the performance of the various methodologies presented here, comparing the algorithms for finding the optimal instantiation order of random SSP's.

4.4.1 The backtracking algorithms: Three backtracking algorithms are compared.

N The naive algorithm (increasing domain size)
HC-p The hill climb past algorithm
IR The IR algorithm.

We generated sets of 30 randomly chosen courses and then applied each of the 3 algorithms with Min varying from 1 to 30. Having gotten the instantiation order according to each one of the algorithms, we activated the regular backtracking algorithm on each one of requirements (where the courses are ordered according to the order obtained before) and the number of instantiation was counted. Figure 3 shows the average results on a set of 1218 problems.

The conclusion drawn is that the hill climb past algorithm is perhaps slightly better than the IR algorithm but both

clearly outperform the naive algorithm, i.e., **HC-p** $\simeq$ **IR** < **N**. The graph shows that there are problems where IR is better and problems where the hill climb past is better. The advantage of the IR algorithm is that it takes less time than the hill climb past algorithm.

4.4.2 The forward checking algorithms: Five forward checking algorithm are compared.

N	The naive algorithm (increasing domain size)
HC-f	The hill climb future algorithm
IR	The IR algorithm
L1	The level-1 hill climbing algorithm
L2	The level-2 hill climbing algorithm.

The same method was used as for the backtracking algorithms but the size of the sets was 20. This is because the level-1 and level-2 based algorithms are very memory consuming. From Figure 4, we can order the algorithms by descending order of performance for $13 \leq \text{Min} \leq 20$ (the best algorithm is first) as follows:

$$\textbf{HC-f} \leq \textbf{IR} < \textbf{N} < \textbf{L1} < \textbf{L2}$$

In Figure 5 we can see a magnification of the results where Min is between 1 and 12. Here the difference in performance of the algorithms is much less significant but does give a different ordering:

$$\textbf{IR} \leq \textbf{HC-f} < \textbf{L2} < \textbf{L1} \simeq \textbf{N}$$

Finally, in order to check the performance on larger sets of up to 30 courses, we ran another experiment only with the 3 algorithms IR, Hill climb future, and Naive. The results are shown in Figure 6 and suggest that the performance order of the algorithms for larger values is **HC-f** < **IR** < **N** where here Hill climb future significantly beats IR. In Figure 7 we see a magnification of the results when Min is between 1 and 16 showing a slight preference for the order **IR** < **HC-f** < **N**.

5. Getting the optimal schedule

Until this point we have presented algorithms for finding a schedule that will satisfy all the constraints, but have not dealt at all with the notion of priorities. Recall from section 2 that the fixed and non-fixed time constraints may be assigned priorities: those constraints which have priority of P or -P and *must* be satisfied (the restrictive constraints), and all others (the non-restrictive constraints) which may be relaxed. In addition, courses within a group may also be assigned a priority. The priorities will allow us to define a measure for the quality of a schedule, which we will call its *mark*. Thus, a schedule will be *valid* if it satisfies all restrictive constraints and will be *optimal* if its mark is maximum over all valid schedules.

Although our aim is to find the schedule with the highest mark, this problem is easily seen to be NP-hard. Since the brute force method of generating all possible schedules and then picking the one with the highest mark is generally impractical, we will need a more direct approach that will not pass through all valid schedules, probably at the expense of optimality.

Three groups of algorithms for finding a schedule have been investigated: (1) hill climbing algorithms, (2) option ordering algorithms, and (3) time table based algorithms. All the algorithms have been implemented and tested against a large number of typical student scheduling problems randomly generated. The results of these tests were collected and analyzed for the quality of the solution and the efficiency of search. In order to compare the schedule produced by each of the algorithms with the best schedule, the brute force algorithm was run separately in order to determine *all* the different marks for all valid schedules. The marks were then *ordered* from highest to lowest and the *closeness* of a schedule to the best schedule was measured in terms of its *rank* in the ordered set of marks. A full discussion of our optimization algorithms and techniques for relaxing constraints can be found in [3, 5].

We report here the results of experiments comparing the following hill climbing and the offering ordering algorithms for finding an optimal schedule:

1. hill climbing (with local evaluation function only)
2. hill climbing (with local and potential evaluation functions)
3. static option ordering (with the marks of the offerings fixed)
4. update option ordering (with the marks of the offerings updated whenever necessary).

We have generated about 1000 random problems and collected the marks of the schedules generated by each of the algorithms. The *quality* of each schedule is the normalization into percents where an optimal schedule gets a 100. As a measurement to the *difficultness* of a problem we have used the time it took for the brute force to find all the solutions. After having clustered the problems into groups having nearly the same total time for finding all solutions by brute force, we averaged the marks of each algorithm in each of these clusters.

In Figure 8 we see that the performance of the update offering ordering algorithm is the best. In Figure 9 we see a comparison of the average time it took for each algorithm to solve each class of problems. We see that from cost effectiveness consideration the update offering ordering algorithm is also the best. Figure 10 shows the performance of the algorithms when the only kind of constraints used is fixed time constraints, and in Figure 11 we have used only non-fixed time constraints.

6. Conclusions

In this paper, we have demonstrated the suitability of the CSP model for use in the student scheduling problem. In our research we have investigated both satisfiability and optimization aspects of the SSP and for each we have determined the algorithm with the best performance. In work reported in [5] we examine constraint overloading to find an approach for finding a minimal set of constraints that must be removed from the real constraints so that at least one solution will exist.

Several topics are suggested for future investigation. (1) All the methods for finding the best instantiation order were based on the fact that only binary constraints are involved. It is important to enhance these methods so they can deal with non-binary constraints as well. (2) Since these methods are global, i.e., estimates are made once at the preprocessing phase, it would be interesting to investigate the cost effectiveness of determining the best order after each instantiation. (3) It seems that the methods developed for the SSP may be appropriate for other kinds of problems that can be viewed as CSPs. Therefore, it may be worthwhile exploring how well these methods apply to such problems, e.g., time-table construction.

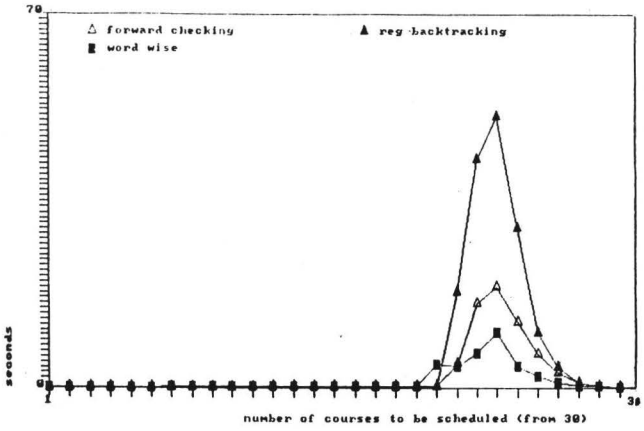

Figure 1. Comparison between CLP algorithms

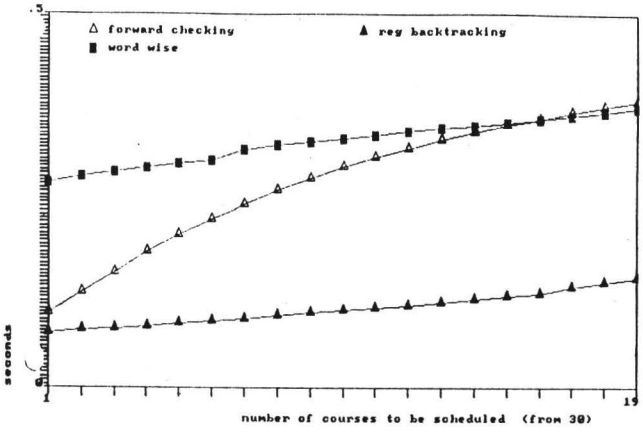

Figure 2. Comparison between CLP algorithms

References

1. J. R. Bitner and E. M. Reingold, Backtrack programming techniques, *Comm. ACM 18* (1975) 651-655.
2. M. Bruynooge, Solving combinatorial search problems by intelligent backtracking, *Infor. Process. Letters 12* (1981).
3. R. Feldman, Interactive Scheduling as a Constraint Labeling Problem, Masters thesis, Dept. of Math. and Comp. Sci., Bar-Ilan University, July 1988.
4. R. Feldman and M. C. Golumbic, Interactive Scheduling as a Constraint Satisfiability Problem, *Annals of Mathematics and Artificial Intelligence 1* (to appear 1990).
5. R. Feldman and M. C. Golumbic, Optimization algorithms for Scheduling via Constraint Satisfiability, IBM Israel Technical Report (Jan. 1989).
6. E. C. Freuder, Synthesizing constraint expressions, *Comm. ACM 21* (1978) 958-965.
7. M. C. Golumbic, Knowledge-based techniques in an academic environment, *Proc. Int'l. Conf. on Courseware and Design and Evaluation, Symposium on Artificial Intelligence and Education*, Ramat Gan, Israel, April 1986, pp. 355-362.
8. M. C. Golumbic, M. Markovich, S. Tsur and U. J. Schild, A knowledge-based expert system for student advising, *IEEE Trans. on Education E-29* (1986) 120-124.
9. Haralick, R.M. and Elliot, G.L., Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence 14* (1980) 263-313.
10. Mackworth, A.K., Consistency in networks of relations, *Artificial Intelligence 8* (1977) 99-118.
11. Montanari, U., Networks of constraints: Fundamental properties and applications to picture processing, *Information Science 7* (1974) 95-132.
12. Nudel, B., Consistent-labeling problems and their algorithms: expected-complexities and theory-based heuristics, *Artificial Intelligence 21* (1983) 135-178.
13. Purdom, P. W., Search rearrangement backtracking and polynomial average time, *Artificial Intelligence 21* (1983) 117-133.

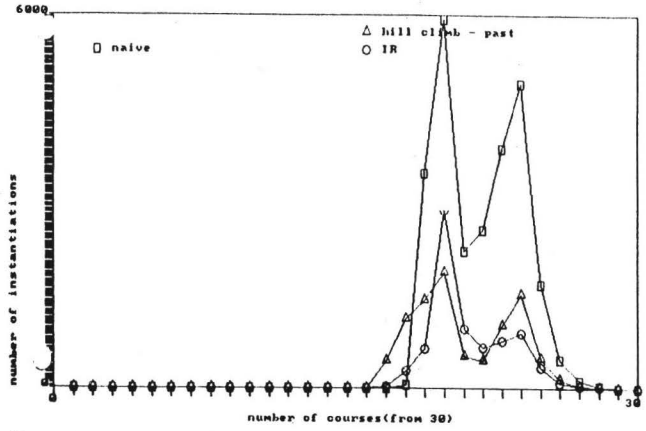

Figure 3. Ordering algorithms – regular backtracking

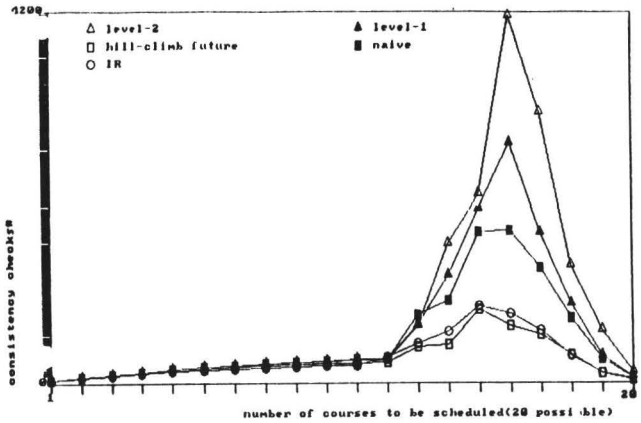

Figure 4. Ordering algorithms – forward checking

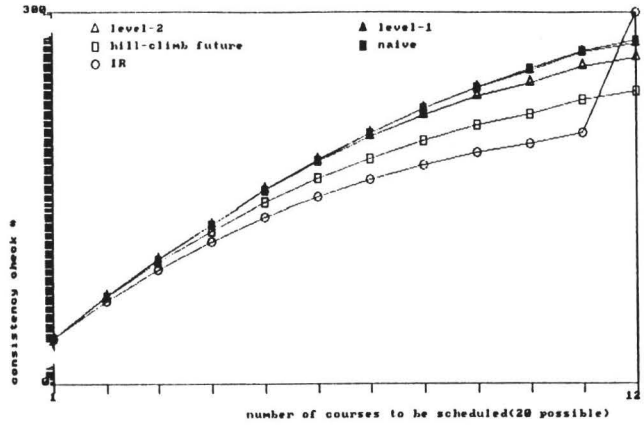

Figure 5. Ordering algorithms – forward checking

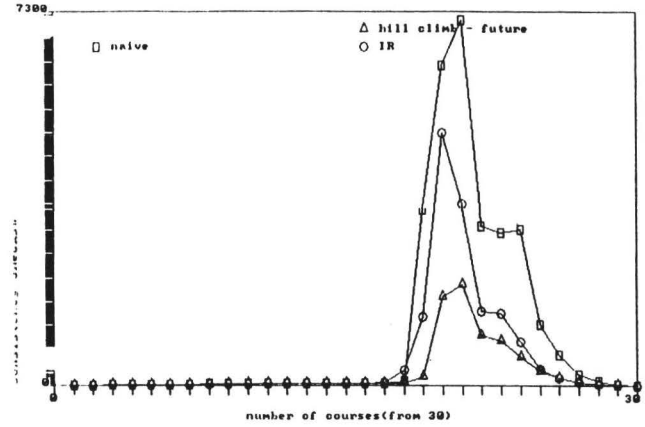

Figure 6. Ordering algorithms – forward checking

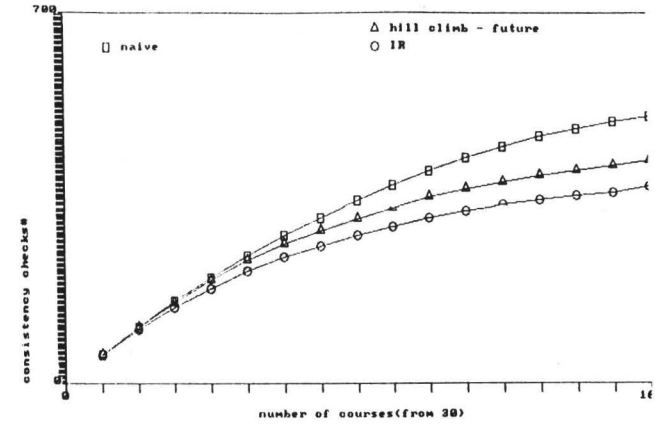

Figure 7. Ordering algorithms – forward checking

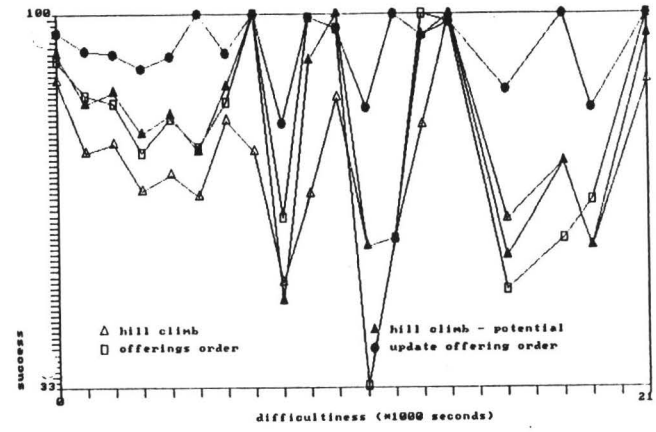

Figure 8. Finding the optimal schedule – percentage of the optimum value

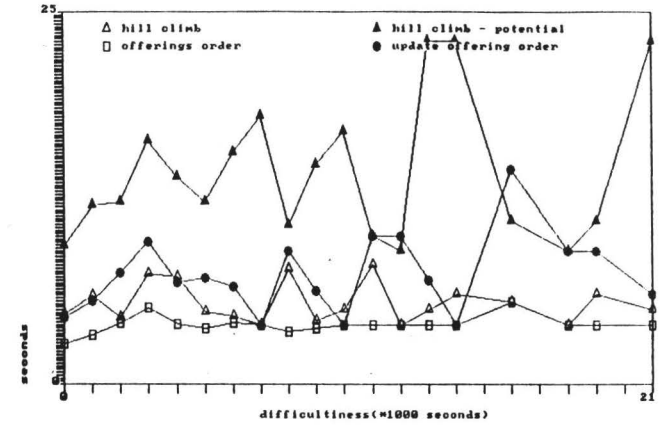

Figure 9. Finding the optimal schedule – running time

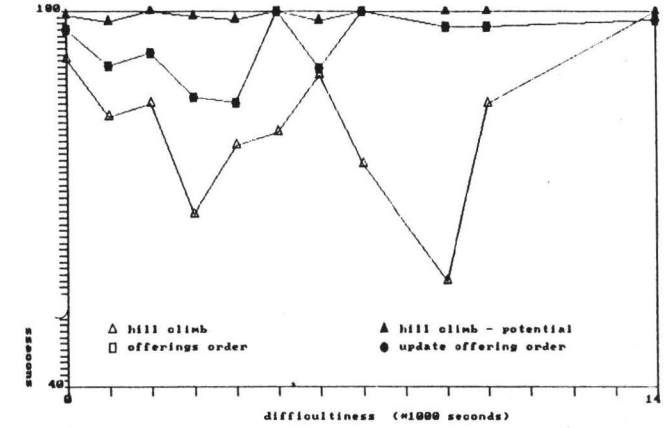

Figure 10. Optimal schedule – non-fixed constraints

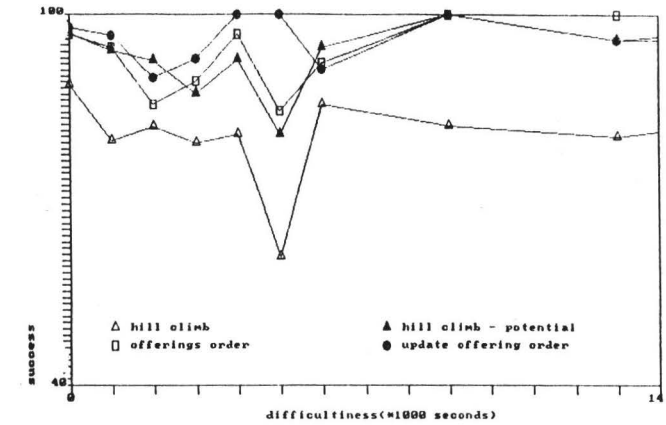

Figure 11. Optimal schedule – fixed constraints