Looking at Full Looking Ahead *

Daniel Frost and Rina Dechter
Dept. of Information and Computer Science
University of California, Irvine, CA 92717-3425 U.S.A.
{dfrost, dechter}@ics.uci.edu
Frost: (714) 824-1084
Dechter: (714) 824-6556
Fax: (714) 824-4056

Abstract

Haralick and Elliott's full looking ahead algorithm [4] was presented in the same article as forward checking, but is not as commonly used. We give experimental results which indicate that on some types of constraint satisfaction problems, full looking ahead outperforms forward checking.

We also present three new looking ahead algorithms, all variations on full looking ahead, which were designed with the goal of achieving performance equal to the better of forward checking and full looking ahead on a variety of constraint satisfaction problems. One of these new algorithms, called smart looking ahead, comes close to achieving our goal.

Keywords: Constraint Satisfaction, Algorithms, Forward Checking, Experimental Analysis

*This work was partially supported by NSF grant IRI-9157636, by the Electrical Power Research Institute (EPRI), and by grants from Toshiba of America, Xerox Northrop and Rockwell.

1 Introduction

In 1980, Haralick and Elliott [4] introduced the forward checking algorithm, as well as two variants which they called partial looking ahead and full looking ahead. All three algorithms can be used to solve constraint satisfaction problems, that is, to find one or more consistent solutions, or to prove that no consistent solution exists. Over the last 15 years, forward checking has become one of the primary algorithms in the CSP-solver's arsenal, while partial and full looking ahead have received little attention. This neglect is due, no doubt, in large part to the conclusions reached in [4]: "The checks of future with future units do not discover inconsistencies often enough to justify the large number of tests required."

In this paper we have three goals. First, we will demonstrate that the full looking ahead algorithm is more useful than usually supposed, and in fact substantially outperforms forward checking on problems with relatively tight constraints and relatively sparse constraint graphs. Second, we analyze the behavior of full looking ahead, both on problems where it is better than forward checking and on problems where it is worse than forward checking. Because we find experimentally that each algorithm is superior to the other on certain types of problems, we are interested in the possibility of automatically recognizing and using the superior heuristic for any individual problem. Our analysis shows several ways that this may be accomplished. The paper's third contribution is a set of new variants of looking ahead called truncated looking ahead, self-adjusting looking ahead, and smart looking ahead. Our purpose in developing these algorithms is to find an approach which usually performs similarly to the better of forward checking and full looking ahead, and sometimes outperforms either. Experimental evidence indicates that with the smart looking ahead algorithm we are near but not at this goal: its performance is almost always quite good, and in some cases surpasses both forward checking and full looking ahead.

2 Definitions, Algorithms, Problems

2.1 Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) consists of a set of n variables, $X_1, \ldots, X_n$; their respective value domains, $D_1, \ldots, D_n$; and a set of constraints. A *constraint* is a subset of the Cartesian product $D_{i_1} \times \ldots \times D_{i_j}$, consisting of all tuples of values for a subset $(X_{i_1}, \ldots, X_{i_j})$ of the variables which are compatible with each other. A *solution* to a CSP is an assignment of values to all the variables such that no constraint is violated; a problem with a solution is termed *satisfiable* or *consistent*. Sometimes it is desired to find all solutions; in this paper, however, we focus on the task of finding one solution, or proving that no solution exists. A *binary* CSP is one in which each of the constraints involves at most two variables; in this paper we consider only binary CSPs. A constraint satisfaction problem can be represented by a *constraint graph* which has a node for each variable and an arc connecting each pair of variables that are contained in a constraint.

2.2 Forward Checking

The essential idea of forward checking [4] is that when a variable X is instantiated with a value x from its domain, the domain of each future (uninstantiated) variable Y is examined, and if a value y is found such that $X = x$ conflicts with $Y = y$, then the value y is temporarily removed from the domain of Y. When Y is reached and has to be instantiated, the only values remaining in its domain will be those consistent with all earlier variables and their current values. Of course, if later on in the search X becomes uninstantiated, then y has to be restored to the domain of Y. It is therefore convenient to maintain not just the domain D of each variable, but a two-dimensional table of "current" domains, D'. $D'_{d,f}$ is the (possibly proper) subset of D_f which is consistent with the partial instantiation $(X_1 = x_1, X_2 = x_2, \ldots, X_d = x_d)$.

Fig. 1 has detailed pseudo-code of the forward checking algorithm. At the top level is a tree-search algorithm which determines the variable and value to be processed. It is called with an argument *Alg* which determines whether full looking ahead ("FLA") or any of the variants we present later is to be run.

Our forward checking and full looking ahead algorithms are essentially the same as the ones presented in [4], with several differences worth noting. Most immediately apparent, we do not present the algorithms recursively, primarily because this follows our implementation in C. Another significant difference has to do with our reliance on a dynamic variable ordering heuristic embodied in the *SelectNextVar()* function. This function always selects a variable with an empty current domain, if one is available. Therefore, when our version of the algorithm detects that selecting a certain value x will lead to an empty domain in the future, it does not immediately reject x (as does the procedure in [4]). Instead, the tree-depth variable d is incremented, and on the next level of the search the empty-domain variable is encountered. This variable is a dead-end, and d is then decremented. There is no difference between the two approaches in terms of consistency checks, and a negligible difference in CPU time.

Another small difference is that our algorithm does not call FULL-LOOKING-AHEAD() (called "LOOK-FUTURE" in [4]) if an empty domain has been detected.

In addition to *SelectNextVar*, our pseudo-code refers to two other functions which are not explicitly defined. *ConstraintBetween* returns a boolean value which is *true* if there exists a constraint which includes the two variables specified in the function's arguments, and *false* otherwise. This function can be implemented with a static table created before processing begins. Its use can significantly speed up processing.

The *Relation(X,x,Y,y)* function is based on one with the same name in [4]. It

```
TREE-SEARCH(Alg)
d ← 1                                   /* d is depth in search tree */
D'_{1,i} ← D_i for all i                /* initialize the first D''s */
X_1 ← SelectNextVar()
while 1 ≤ d ≤ n                         /* n = the number of variables */
      if D'_{d,d} = ∅                   /* dead-end? */
         then d ← d − 1                 /* yes: backtrack */
              if d = 1
                 then D'_{d,i} ← D_i for all i > d
                 else D'_{d,i} ← D'_{d−1,i} for all i > d
         else select x_d ∈ D'_{d,d}     /* choose a value */
              D'_{d,d} ← D'_{d,d} − {x_d}
              FORWARD-CHECKING(d)
              if Alg = "FLA" and smallest-future-domain > 0
                 then FULL-LOOKING-AHEAD(d)
              d ← d + 1                 /* go to next level in search tree */
              if d ≤ n
                 then X_d ← SelectNextVar()
                      D'_{d,i} ← D'_{d−1,i} for all i
if d = 0
   then return "No solution"
   else return (X_1 = x_1, X_2 = x_2, ..., X_n = x_n)
```

```
FORWARD-CHECKING(d)
for k ← d + 1 to N
    if ConstraintBetween(X_d, X_k)
       then for each y ∈ D'_{d,k}
                 if Relation(X_d, x_d, X_k, y) = false
                    then D'_{d,k} ← D'_{d,k} − {y}
                         if D'_{d,k} = ∅
                            then return
```

Figure 1: The forward checking algorithm.

```
FULL-LOOKING-AHEAD(d)
for k ← d + 1 to N
    for each y ∈ D'_{d,k}
        for j ← d + 1 to N
            if ConstraintBetween(X_k, X_j)
              then match ← false
                   for each z ∈ D'_{d,j}
                       if Relation(X_k, y, X_j, z) = true
                         then match ← true
                              break z loop
                   if match = false
                     then D'_{d,k} ← D'_{d,k} − {y}
                          if D'_{d,k} = ∅
                            then return
```

Figure 2: The Full Looking Ahead variant of Forward Checking.

returns *true* if the assignments of x to X and y to Y do not violate any constraint.

2.3 Full Looking Ahead

The full looking ahead algorithm is similar to forward checking, but it does more extensive processing after each variable is instantiated. The purpose of this processing is to remove more values from the domains of future variables which cannot possibly be part of a solution. This reduces the branching factor of the remaining search, and may also lead to earlier detection of a future variable that will be a dead-end.

The additional processing done by full looking ahead is a limited form of arc-consistency, in effect performing a single iteration of the "revise" procedure described in [5]. Suppose there are three future variables, X with current domain $\{a, b\}$, Y with current domain $\{a, b\}$ and Z with current domain $\{b\}$. There is an inequality constraint (as in graph coloring) between X and Y and between Y and Z. Full looking ahead will process X and reject neither of its values, since they both have a compatible value in the domain of Y. When full looking ahead processes Y, it will remove the value b because there is no allowable match for b in the domain of Z. Arc consistency would later go back and remove a from the X's domain, because it no longer has a consistent match in Y's domain, but full looking ahead does not do this.

See Fig. 2 for detailed pseudo-code of full looking ahead.

2.4 Dynamic Variable Ordering

The order in which variables are considered while solving a constraint satisfaction problem with forward checking or any looking ahead method can have a substantial impact on the amount of work required to find a solution. In a dynamic variable ordering, the order of variables is determined as the search progresses, and can vary from one branch of the search tree to another. We use a dynamic variable ordering scheme proposed in [4] and widely adopted today: always select next the variable with the smallest remaining current domain (i.e. D'). If $D'_{d,i}$ is empty for some uninstantiated variable X_i, then X_i is moved to be the next variable, and a dead-end occurs immediately. The technique for breaking ties is important, as there are often many variables with the same domain size. In our implementation we maintain the uninstantiated variables in decreasing order of the number of constraints they participate in in the original problem. In case of a tie for domain size (and for the first variable), the first variable in this ordering is selected. This scheme gives substantially better performance than picking one of the tying variables at random.

2.5 Random Problem Generator

The experiments reported in this paper were run on random instances generated using a model that takes four parameters: N, K, T and C. The problem instances are binary CSPs with N variables, each having a domain of size K. The parameter T (tightness) specifies a fraction of the K^2 value pairs in each constraint that are disallowed by the constraint. The value pairs to be disallowed by the constraint are selected randomly from a uniform distribution, but each constraint has the same fraction T of such incompatible pairs. T ranges from 0 to 1, with a low value of T, such as 1/9, termed a loose or relaxed constraint. The parameter C specifies the number of constraints in each problem, out of the $N * (N-1)/2$ possible. The specific constraints are chosen randomly from a uniform distribution.

Certain combinations of parameters generate problems of which about 50% are satisfiable; such problems are on average much more difficult than those which almost all have solutions (under-constrained) or which almost never have solutions (over-constrained) [1, 6]. Such a set of parameters is sometimes called a *cross-over point*. The experiments reported in this paper were all conducted with parameters at or very near the cross-over point.

3 Experiments with full looking ahead

Our first experiment was designed to indicate under what circumstances full looking ahead might be a worthwhile alternative to forward checking, when coupled with the dynamic variable ordering heuristic described above. We selected a diverse set of parameters for our random problem generator, all at the 50% solvable cross-

Problem Parameters				Mean CPU seconds	
K	T	N	C	FC	FLA
3	1/9	150	1112	**6.70**	54.56
3	2/9	175	546	41.23	**1.57**
3	3/9	150	244	39.26	**0.27**
6	4/36	50	710	**15.80**	40.52
6	8/36	70	452	**23.25**	36.44
6	12/36	90	344	25.78	**14.23**
6	16/36	100	246	22.52	**1.30**
9	18/81	35	293	**4.19**	9.25
9	27/81	50	253	**11.86**	16.49
9	36/81	70	234	49.55	**16.88**

Figure 3: Comparison of forward checking (FC) and full looking ahead (FLA). Each number is the mean of 500 instances; the lower mean on each row is in boldface.

over point, and with each set of parameters generated 500 problems. These problems were then solved using both forward checking and full looking ahead. The results of this experiment are reported in Fig. 3. The trend seems to be that full looking ahead is superior on problems with tighter constraints, with this trend being less pronounced as the size of the domain increases. This is not surprising, given the nature of the full looking ahead algorithm (refer to Fig. 2). For a given value of N (number of variables) and K (domain size), T (tightness) and C (number of constraints) will have an inverse relationship among sets of parameters at the cross-over point. Fewer and tighter constraints means that the test on line 4 of Fig. 2 will fail more often, and when it does succeed, it is more likely for *match* to be *false* and the value y to be removed. Thus we would expect that on problems generated with a higher value of T (e.g. 3/9, 16/36, 36/81), full looking ahead will generally have to do less work with better results.

Fig. 3 shows only one large value of N for each combination of K and T. To show that these numbers are indicative of the trend as N increases, Fig. 4 presents the results of comparing forward checking and full looking ahead over a variety of values for N, for K=3. The figure also shows consistency checks and nodes expanded in the search tree, as well as CPU time. Consistency checks (which are calls to the *Relation* subroutine in Figs. 1 and 2) are a common measure of the work done by constraint satisfaction algorithms. Because calling *Relation* is quite fast (at least in our implementation), and because full looking ahead trades extra calls to *Relation* for a smaller search space, just reporting consistency checks would be misleading. The middle boxes in each row of Fig. 4 report the number of search tree nodes, which is the number of times lines 10–14 of TREE-SEARCH are executed. This data indicates the size of the search tree, which is almost always much smaller for full

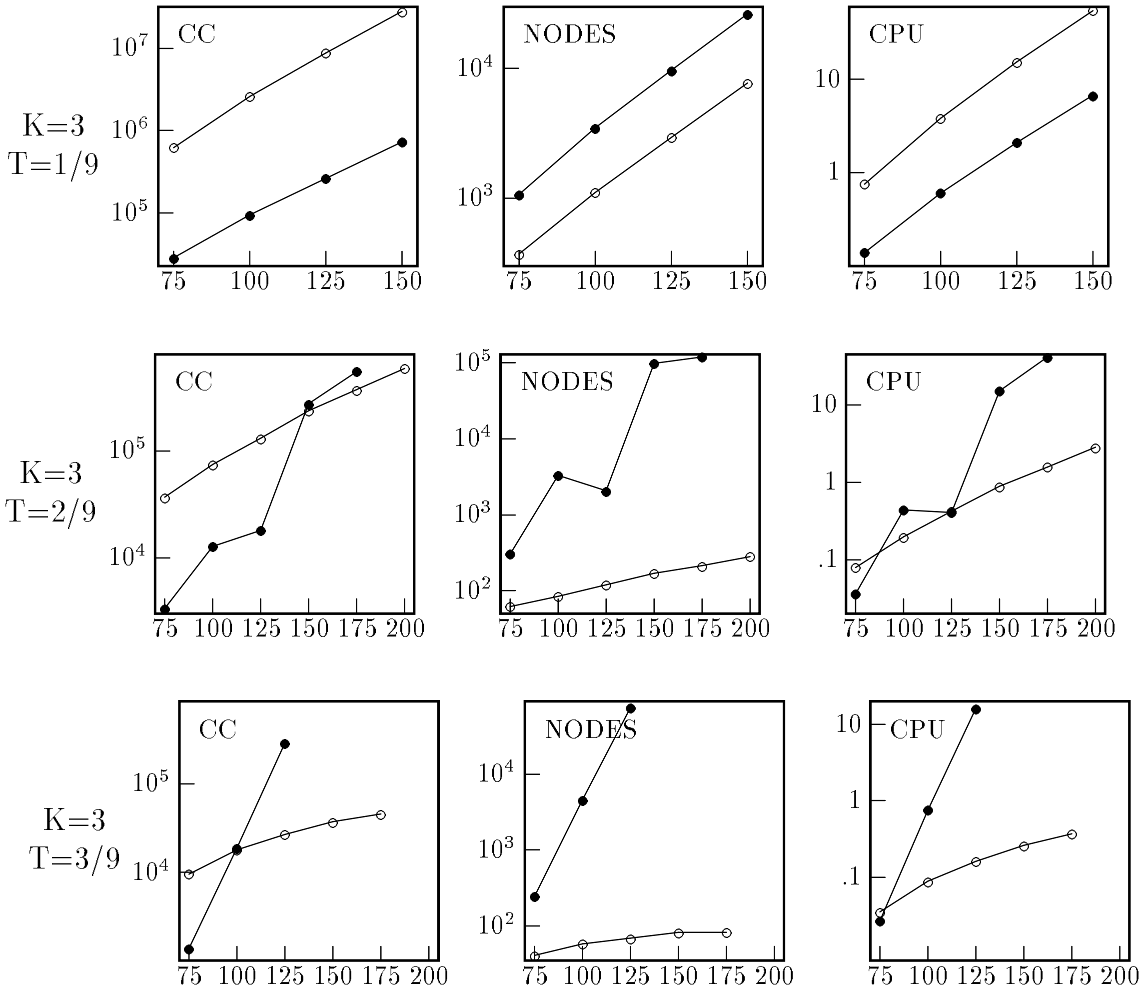

Figure 4: Comparison of forward checking (indicated by a bullet: •) and full looking ahead (a circle: ∘). Each data point is the mean of 500 instances generated with K=3, T=1/9 (first row), T=2/9 (second row) or T=3/9 (third row), and N as indicated along the x-axis. The value of C was set to put the problems at the cross-over point. The leftmost boxes on each row show consistency checks, the middle boxes show nodes expanded in the search tree, and the right most boxes show CPU time in seconds. Note that the y-axes are logarithmic.

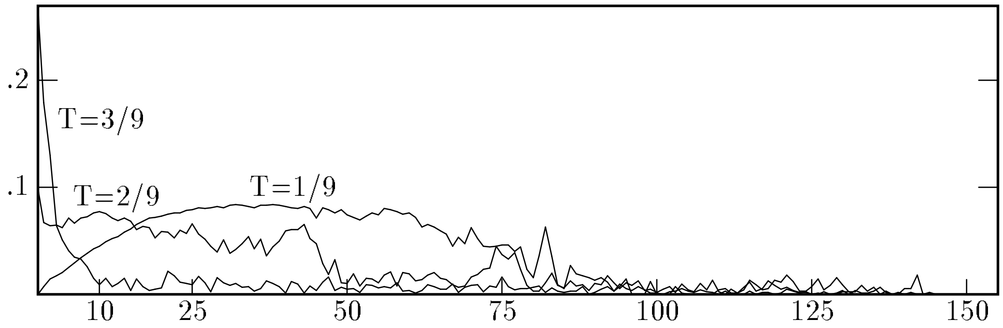

Figure 5: The fraction of future variable domain values removed by full looking ahead (along the y-axis), as a function of depth in the search tree (along the x-axis). Each line represents the mean over 500 instances, for a single set of parameters with T=3, K=1/9, 2/9 or 3/9, and N and C as in Fig. 3.

looking ahead than for forward checking.

4 Variants of full looking ahead

In this section we report on a method we used to analyze the behavior of full looking ahead. This analysis led to the development of three new algorithms. In each case, our hope was that the new algorithm would do only a small amount of work beyond that done by forward checking, while reaping most of the benefits of full looking ahead.

To help us understand why full looking ahead is better for certain classes of problem than for others, we modified our program to report what fraction of the values in the domains of future variables are removed during the full looking ahead process. This fraction is a measure of the effectiveness of full looking ahead. In terms of our pseudo-code in Fig. 2, we measured the ratio of times a value y is removed in line 11 to the number of values of y generated in line 2. We tabulated this ratio relative to the value of d, the depth in the search tree. The results are shown in Fig. 5.

Studying Fig. 5 prompted the observation that when full looking ahead is most successful (that is, for K=3/9), most of its power seems to come at shallow levels in the search tree (where d is small). When the constraints are loose (e.g. T=1/9), full looking ahead is most likely to remove values somewhat deeper in the tree, but in such cases the cost to do so does not seem to be justified.

We therefore developed a version of full looking ahead which we call truncated looking ahead (TLA). The modification is simple: the extra processing associated with full looking ahead is done only when the newly instantiated variable is at depth 10 or less in the search tree. At depth greater than 10, truncated looking ahead is

Add two lines to TREE-SEARCH():

14.1 **if** Alg = "TLA" and $d \leq 10$ and smallest-future-domain > 0
14.2 **then** FULL-LOOKING-AHEAD(d)

Figure 6: Truncated Looking Ahead

Add two lines to TREE-SEARCH():

14.1 **if** Alg = "SLA" and smallest-future-domain > 1
14.2 **then** SMART-LOOKING-AHEAD(d)

SMART-LOOKING-AHEAD(d)
Lines 1—11 are the same as FULL-LOOKING-AHEAD(). Change lines 12 and 13 to read:

12 **if** $sizeof(D'_{d,k}) = 0$ or 1
13 **then return**

Figure 7: Smart Looking Ahead

identical to forward checking. Pseudo-code for truncated looking ahead is given in Fig. 6.

Another promising variant of full looking ahead is called self-adjusting looking ahead (SALA). This algorithm starts the full looking ahead process at every level in the search tree, but stops if progress is not being made. Progress is defined as removing a sufficient fraction of the values in the domains of future variables, and is controlled by a parameter called *Credit*. See Fig. 8 for the pseudo-code. Each time a future variable is considered and no value in its domain is removed, the credit level c is decremented by one. If a value is removed, the credit level is increased by *Credit*. Determining the most effective setting for *Credit* must be done experimentally. The best results may arise from adjusting *Credit* to have different values at different levels of the search tree (our implementation of self-adjusting looking ahead does not do this). In the experiments reported below, we tried settings of 5 and 10 for *Credit*.

A third variation of full looking ahead is closely coupled with the dynamic variable ordering scheme. One advantage of performing full looking ahead is that values

```
Add two lines to TREE-SEARCH():
14.1        if Alg = "SALA" and smallest-future-domain > 0
14.2          then SELF-ADJUSTING-LOOKING-AHEAD(d)
```

```
SELF-ADJUSTING-LOOKING-AHEAD(d)
 0  c ← Credit
 1  for k ← d + 1 to N
 2      for each y ∈ D'_{d,k}
 3          for j ← d + 1 to N
 4              if ConstraintBetween(X_k, X_j)
 5                then match ← false
 6                     for each z ∈ D'_{d,j}
 7                         if Relation(X_k, y, X_j, z) = true
 8                           then match ← true
 9                                break z loop
10                     if match = false
11                       then D'_{d,k} ← D'_{d,k} − {y}
12                            if D'_{d,k} = ∅
13                              then return
14      if any value was removed from D'_{d,k}
15        then c ← c + Credit
16        else c ← c − 1
17      if c = 0
18        then return
```

Figure 8: Self Adjusting Looking Ahead. Lines 1–13 are the same as in FULL-LOOKING-AHEAD().

Add two lines to TREE-SEARCH():

14.1 **if** Alg = "PLA" and smallest-future-domain > 0

14.2 **then** PARTIAL-LOOKING-AHEAD(d)

PARTIAL-LOOKING-AHEAD(d)

The same as FULL-LOOKING-AHEAD(), except that line 3 is changed to read:

3 **for** $j \leftarrow k+1$ to N /* is $j \leftarrow d+1$ in FULL-LOOKING-AHEAD */

Figure 9: Parital Looking Ahead

in the domains of future variables are removed. This leads to another advantage: the dynamic variable ordering heuristic is more likely to find a future variable which has a very small domain size. In the absence of a dead-end, we hope to find a future variable with a domain size of 1, as instantiating this single value represents a forced choice that will have to be made eventually. The smart looking ahead (SLA) algorithm performs the full looking ahead level of consistency enforcing, but stops when the current domain of some future variable becomes 0 or 1 (see Fig. 7). Once a dead-end or single-value domain is found, the smart looking ahead procedure returns to TREE-SEARCH(), and the variable with that 0 or 1 size domain will be made the next variable in the ordering. The goal is to do enough looking ahead to effectively guide the variable ordering heuristic.

It would be remiss not to mention one of the original variants of looking ahead, called partial looking ahead [4]. This scheme checks each future variable only with other future variables later than it, thus performing directional arc consistency at each step [2]. As noted in [4], partial looking ahead makes about half the consistency checks of full looking ahead. Partial looking ahead is described in Fig. 9.

A comparison of all algorithms discussed in the paper is shown in Fig. 10. No one new variant completely realizes our goal of always being equal to the better of forward checking and full looking ahead. However, smart looking ahead comes reasonably close: it is best or near-best in three out of the 11 sets of problems, and it takes more than twice as much time as the best algorithm only for problems with K=3 and T=3/9. Also, smart look ahead has the smallest maximum average CPU time (23.37) of any of the algorithms we experimented with.

Problem Parameters				Mean CPU seconds						
K	T	N	C	FC	TLA	SALA-5	SALA-10	SLA	PLA	FLA
3	1/9	150	1112	**6.70**	10.76	20.88	28.88	12.02	54.80	54.56
3	2/9	175	546	41.23	25.22	1.70	**1.02**	**1.22**	2.68	1.57
3	3/9	150	244	39.26	13.95	14.70	**0.11**	3.38	5.41	**0.27**
3	3/9	175	281		29.12	6.93	**1.27**	10.40	38.93	**0.37**
6	4/36	50	710	**15.80**	30.60	31.93	38.09	20.86	39.04	40.52
6	8/36	70	452	**23.25**	25.29	29.76	33.95	23.37	47.59	36.44
6	12/36	90	344	25.78	13.13	**11.72**	12.74	**12.54**	28.20	14.23
6	16/36	100	246	22.52	2.61	**1.14**	**1.17**	1.75	4.14	1.30
9	18/81	35	293	**4.19**	9.25	8.41	9.21	6.14	9.04	9.25
9	27/81	50	253	**11.86**	15.38	15.13	16.33	12.34	20.10	16.49
9	36/81	70	234	49.55	12.71	22.80	20.39	**12.62**	36.81	16.88

Figure 10: Comparison of six algorithms: forward checking (FC), truncated looking ahead (TLA), self-adjusting looking ahead with *Credit* = 5 (SALA-5) and 10 (SALA-10), smart looking ahead (SLA), partial looking ahead (PLA), and full looking ahead (FLA). Each number is the mean of 500 instances. The best time in each row (or best two when there is a near-tie) is in boldface. The FC column is blank where we were unable to run all 500 instances because too much CPU time was required.

5 Discussion

The experimental results in this paper have to be considered in the context of the dynamic variable ordering heuristic we used. In particular, self-adjusting looking ahead, smart looking ahead, and partial looking ahead are all affected by the order of the future variables. In our implementation, the future variables are kept in decreasing order of degree in the original constraint graph. A different scheme might have a significant impact on these algorithms.

Another limitation of our experimental results is that we only generated problems at the cross-over point. We think it very likely that the usefulness of the algorithms considered would be much different on over- or under-constrained problems. As a general rule of thumb, on easy problems pre-processing such as is done by full looking ahead is unlikely to pay off. In fact, even forward checking may do too much work in advance. We have cast scheduling problems in the constraint satisfaction framework, and some of these problems are under-constrained (they have a large number of solutions) but have thousands of variables. On such problems the look-ahead that forward checking does is too time-consuming, and the benefits too small, as future dead-ends are rarely uncovered. It may be better do perform minimal forward checking as suggested in [3], or backtracking or backjumping with no look-ahead component. Taking a larger perspective than we adopt in the body of this paper, the ideal might be an algorithm always does the "right" amount of looking ahead, from as little as backtracking, through forward checking and full looking ahead, to full arc consistency at each instantiation.

6 Conclusion

We have presented experimental evidence that the full looking ahead algorithm developed in [4] is often preferable to forward checking, at least in conjunction with dynamic variable ordering and on hard problems with tight constraints. Three new looking ahead algorithms, truncated looking ahead, self-adjusting looking ahead, and smart looking ahead, were shown experimentally to have some desirable characteristics. In particular, smart looking ahead, which stops comparing future variables with other future variables whenever a domain size is reduced to 1 or 0, seems to be a promising method for achieving the best of both forward checking and full looking ahead.

References

[1] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the *really* hard problems are. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 331–337, 1991.

[2] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[3] M. J. Dent and R. E. Mercer. Minimal forward checking. Technical Report 374, The University of Western Ontario, Dept. of Computer Science, 1993.

[4] R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[5] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[6] David Mitchell, Bart Selman, and Hector Levesque. Hard and Easy Distributions of SAT Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, 1992.