Adaptive Lookahead for Answer Set Computation

Guohua Liu and Jia-Huai You
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
guohua, you@cs.ualberta.ca

Abstract

Lookahead is a well-known constraint propagation technique for DPLL-based SAT and answer set solvers. Despite its space pruning power, it can also slow down the search, due to its high overhead. In this paper, this twofold effect is analyzed. On one side, we give characterizations of the problems for which the cause for the reduction of search efficiency shows clearly. On the other we show that problem instances that lie in the phase transition regions often significantly benefit from the use of lookahead. Our analysis leads to a proposal of adaptive lookahead*, which performs lookahead according to the learned information during the search. Adaptive lookahead is implemented in one of the best-known answer set solvers,* smodels*. Our experiments show that adaptive lookahead adapts well to different search environments it is going through.*

1. Introduction

Answer set programming (ASP) has been advocated as a novel constraint programming paradigm and several implementations have been developed [6, 8, 11, 13, 18]. ASP is closely related to boolean satisfiability (SAT).

Most complete SAT/ASP solvers are variants of the DPLL search algorithm [4]. *Unit propagation*, sometimes also called *boolean constraint propagation* (BCP), is considered the most important component of a SAT solver which may drastically affect its performance. In the popular answer set solver *smodels* [18], the algorithm that corresponds to BCP is called the *expand* function.

On top of BCP/*expand*, other deductive mechanisms have been proposed. One of them is called *lookahead* [7] - before a decision on a choice point is made, for each unassigned atom, if fixing the atom's truth value leads to a contradiction, the atom gets the opposite truth value. In this way, an atom may get a truth value from the truth value propagation of already assigned atoms without going through a search process.

Lookahead, however, incurs high overhead [20]. The high pruning power, along with non-ignorable overhead, has made lookahead a somewhat controversial technique. There are two camps of constraint solvers. In one of them lookahead is employed and in the other it is not.

The (in)effectiveness of lookahead in SAT solvers were studied in [10]. The main conclusion is that lookahead does not pay off when integrated with look-back methods. This paper focuses on the effect of lookahead on answer set computation without any look-back techniques. On one hand we show that for some extremely hard problem instances, lookahead can indeed significantly improve the search. On the other hand, we provide some characterizations and identify the representative benchmarks for which lookahead downgrades the search efficiency.

Based on these observations, we propose an *adaptive lookahead* mechanism and implement it in the ASP solver *smodels*. The resulting system is called *A-smodels*, in which lookahead is carried out only when it tends to be beneficial to do so. Our experiments show that adaptive lookahead adapts well to different search environments it is going through - it performs like smodels with lookahead for the problems that benefit from the use of lookahead, and without lookahead for those problems for which the use of lookahead slows down the search.

In the direction of efficiently using lookahead in ASP solvers, it is proposed in [13] and [15] to use lookahead in a limited manner, performing lookahead on a subset instead of all of the unassigned atoms. While [15] deals with normal logic programs, [13] focuses on disjunctive logic programs.

Adaptive lookahead deals with normal logic programs. The main difference between adaptive lookahead and *limited lookahead* in [15] is that adaptive lookahead turns on/off lookahead according to the information - frequencies of conflicts and dead-ends discovered during the search. Limited lookahead on the other hand depends on the status of literals - it chooses a literal for lookahead if assuming its value leads to at least one inference. A more detailed

comparison will be given later in the paper.

The general idea of adaptive constraint propagation is also of interest in CSP research [2, 17, 12]. The approach in [12] incrementally applies arc-consistency to obtain higher consistency and [2, 17] switch between different consistency algorithms. But none of them deals with the lookahead technique. Adaptive lookahead is similar to [2] in the sense that they both use some thrashing prediction mechanism instead of the domain information as in [17].

The next section provides the background. Section 3 presents some characterizations by which we identify problems that run much slower with lookahead. The adaptive lookahead algorithm is given in Section 4. Section 5 provides experimental results. The summary and future directions are given in Section 6.

2 Answer Set Computation in Smodels

In this paper, we consider the class of *normal logic programs*. A normal logic program consists of function-free rules of the form $A \leftarrow B_1, ..., B_m, \textbf{not } C_1, ..., \textbf{not } C_n$, where A, B_i and C_i are atoms (also called *positive literals*), and **not** C_i are called *negative literals*. The head of a rule may be the special atom $\bot$ representing the atom *false*, in which case the rule serves as a constraint. We may simply call a normal program a *logic program* or just a *program*.

A logic program may contain variables. The definition of *answer set* (also called *stable model* [9]) as well as the computation of answer sets is based on the ground instantiation of a given program.

Let P be a (ground) logic program. A set of atoms M is an answer set for P if and only if M is the least model of the reduct of P w.r.t M, P^M, which is obtained by

1. deleting each rule in P that has a negative literal **not** ϕ in its body such that $\phi \in M$, and
2. deleting all negative literals in the remaining rules.

Both positive and negative literals are called *literals*. Given a program P, $Literal(P)$ denotes the set of literals appearing in P. A set of literals is *consistent* if there is no atom a such that a and **not** a are both in the set. A *partial assignment* is a consistent set of literals. $Atoms(P)$ denotes the set of distinct atoms appearing in P (excluding the special atom $\bot$). The expression $not(\textbf{not } a)$ is identified with a, and $not(a)$ is **not** a. Given a set of literals B, $B^+ = \{a \mid a \in B\}$ and $B^- = \{a \mid \textbf{not } a \in B\}$. Given a set of atoms S, we define $not(S) = \{\textbf{not } a \mid a \in S\}$.

Smodels implements the answer set semantics for normal logic programs. A *choice point* is defined as a point during search where the branching heuristic picks a literal to assign a value. In the literature, this is also referred to as *making a decision*. Before making a decision, smodels performs constraint propagation. When lookahead is not involved, constraint propagation is carried out by a function called $expand(P, A)$, where P is a program and A a partial assignment. When lookahead is employed, constraint propagation is carried out as follows (cf. Algorithms 1 and 2): for each unassigned atom x, assume a truth value for it (via the function lookahead_once), if it leads to a conflict, then x gets the opposite truth value. This process continues, repeatedly, until no atom can be fixed a truth value by lookahead. Truth values are propagated in lookahead by $expand(P, A)$, which returns a superset of A, representing the process of propagating the values of the atoms in A to some additional atoms. In lookahead_once, the function $conflict(P, A)$ returns true if $A^+ \cap A^- \neq \emptyset$ and false otherwise. We say a conflict is detected if $conflict(P, A)$ returns true.

Algorithm 1 lookahead(P, A)

```
1: repeat
2:    A' = A
3:    A := lookahead_once(P, A)
4: until A = A'
5: return A
```

Algorithm 2 lookahead_once(P,A)

```
1: B := Atoms(P) − Atoms(A)
2: B := B ∪ not (B)
3: while B ≠ ∅ do
4:    Take any literal x ∈ B
5:    A' := expand(P, A ∪ {x})
6:    B := B − A'
7:    if conflict(P,A') then
8:       return expand(P, A ∪ {not (x)})
9: return A
```

A call to $expand(P, A)$, returns a superset of A by repeatedly applying the following five propagation rules until no new literals can be deduced:[1]:

1. Add the head of a rule to A if the body is true in A;
2. If there is no rule with a as the head whose body is not false w.r.t. A, then add **not** a into A;
3. If $a \in A$, and there is exactly one rule with a as the head whose body is not false in A, then add each literal in the body of the rule into A.
4. If the head of a rule is false in A, and all the body literals are in A except one, say l, then add $not(l)$ into A.

[1]The first four rules constitute what is called the $Atleast$ function in smodels, and the last rule describes the effect of the $Atmost$ function [18].

5. If a set of atoms S is *unfounded*[2] w.r.t. A, then A becomes $A \cup not(S)$ [19].

3 Ineffectiveness of Lookahead

We give two characterizations under which the use of lookahead is totally or nearly totally wasted. The first describes the situations where no pruning is ever generated by lookahead in the course of solving some parts of a problem, which are called *easy sub-programs*. The second is called *spurious pruning*, which describes the situations where a literal added by lookahead to a partial assignment is immaterial to the rest of the search.

3.1 Easy sub-programs

Let P' and P be two ground programs, P' is called a *sub-program* of P if $P' \subseteq P$. A program P is said to be *easy* if any partial assignment can be extended to a solution.

Proposition 1 *Given a program P, if it is an easy program then lookahead is totally wasted during the process of search, in that, if $expand(P, A) = A$ then $lookahead(P, A) = A$, for any partial assignment A generated during the search.*

This is the case where lookahead does not do anything more than what *expand(P, A)* does. Proposition 1 can be proved by the definition of lookahead.

A program for solving a hard problem may contain many nontrivial easy sub-programs for which lookahead in the search for answer sets is totally wasted. An example is the pigeon-hole problem, which is to put N pigeons into M holes so that there is at most one pigeon in a hole and every pigeon must take some hole. Consider the following program taken from [16].

$$
\begin{aligned}
pos(P,H) &\leftarrow \textbf{not } negpos(P,H).\\
negpos(P,H) &\leftarrow \textbf{not } pos(P,H).\\
\bot &\leftarrow pos(P,H),\ pos(P,H'), H \neq H'.\\
\bot &\leftarrow pos(P,H),\ pos(P',H), P \neq P'.\\
hashole(P) &\leftarrow pos(P,H).\\
\bot &\leftarrow pigeon(P), \textbf{not } hashole(P).
\end{aligned}
$$

The first two rules enumerate all placements of the pigeons. The third and fourth rules state no pigeon may take more than one hole and no hole may be occupied by more than one pigeon. The last two rules guarantee that each pigeon takes some hole.

[2]Unfoundedness is to capture the atoms that should be false for the generation of minimal models. For a logic program, unfoundedness is mainly due to the positive loops in the program. E.g. if a program consists of two rules $\{a \leftarrow b.\ b \leftarrow a\}$, then $\{a, b\}$ is unfounded, given $A = \emptyset$.

The problem can be very difficult for DPLL-based solvers when $M = N - \delta$, for some positive integer δ.

To see that lookahead is totally wasted in solving some easy sub-programs, consider any partial assignment A in the course of computing an answer set by smodels. Here, let A be such that some pigeons already take holes, e.g., suppose $pos(p_i, h_j) \in A$, for some i and j. Note that when $lookahead(P, A)$ is invoked, propagations by the *expand(P, A)* function is completed, i.e., $expand(P, A) = A$. Thus, we have $hashole(p) \in A$, for all pigeon p, due to the last rule; similarly, **not** $pos(p_i, h_k) \in A$, for any hole h_k different from h_j, due to the third rule; and **not** $pos(p_m, h_j) \in A$, for any other pigeon p_m, due to the fourth rule. Now, suppose $lookahead(P, A)$ is called, which calls $expand(P, A \cup \{pos(p', h')\})$ inside $lookahead\_once(P, A)$. This is to assume $pos(p', h')$ and attempt to derive a conflict. It can be verified that whenever there are at least three holes left unoccupied, lookahead finds no conflict.

In fact, lookahead begins to detect conflicts only when there are two holes left. When one of them is assumed to be taken by one pigeon, all the other pigeons that do not have a hole will be competing for the only remaining hole (by the definition of *expand(P, A)*, the rule with $hashole(p)$ as the head makes the unassigned $pos(p, h)$ true for the only remaining h). In other words, lookahead begins to find conflicts only when most of the holes have been assigned.

In this example, the sub-programs by removing some facts about pigeons so that the resulting number of pigeons equals to the number of the holes are typical easy sub-programs the search is going through.

Besides the pigeon-hole problem, the clique coloring problem is another typical problem which has easy sub-programs. The pigeon-hole, clique coloring and parity problems are known to be exponentially difficult for any conventional resolution-based provers (including any DPLL implementation) [5]. It would be interesting to investigate if the parity problem also has easy-subprograms.

3.2 Spurious pruning

When lookahead finds a conflict, some search space is pruned. But this pruning may be immaterial to the rest of the search. Suppose, by an invocation of lookahead, a literal, say l, is added to the current partial assignment A. The addition of l may not contribute to further constraint propagations. This can be described by an equation

$$expand(P, A) \cup \{l\} = expand(P, A \cup \{l\})$$

for any partial assignment A generated during search, such that the superset returned by $expand(P, A \cup \{l\})$ is not an answer set. This is what we mean by *spurious pruning*. In

this case, lookahead is unnecessary since the decision on l can be delayed to any later choice point.

We can use the number of calls to the *expand(P, A)* function during search to measure the effectiveness of lookahead. Suppose N_{lh} and N_{nlh} denote the number of calls to *expand(P, A)* in smodels with and without lookahead respectively. We have the following proposition.

Proposition 2 *Given a program P, if all the pruning by lookahead are spurious, then $N_{nlh} \leq N_{lh}$.*

An example where the search only generates spurious pruning is the Hamiltonian cycle problem for complete graphs. The following program is taken from [16] with some minor changes in predicate names.

$$
\begin{aligned}
hc(V,U) &\leftarrow arc(V,U), \textbf{not } otherroute(V,U).\\
otherroute(V,U) &\leftarrow arc(V,U), arc(V,W), hc(V,W), U \neq W.\\
otherroute(V,U) &\leftarrow arc(V,U), arc(W,U), hc(W,U), V \neq W.\\
reach(U) &\leftarrow arc(V,U), hc(V,U), reach(V), \textbf{not } init(V).\\
reach(U) &\leftarrow arc(V,U), hc(V,U), init(V).\\
\bot &\leftarrow vertex(V), \textbf{not } reach(V).
\end{aligned}
$$

The first three rules ensure that for each node exactly one incoming and outgoing arc belong to the path. The last three rules state that the path forms a cycle which visits all nodes and returns to the initial node. The literal $hc(u, v)$ being true in a stable model means $arc(u, v)$ belongs to the Hamiltonian cycle.

Consider a complete graph on nodes $v_1, v_2, ..., v_n$, for some sufficiently large n. Suppose in the current partial assignment A, a path like $v_1 \leftarrow v_2 \leftarrow v_3$ is established. Note that **not** $hc(v_2, v_3) \in expand(P, A)$, due to the third rule. At this point, in the call $lookahead(P, A)$, $expand(P, A \cup \{hc(v_1, v_3)\})$ will derive a conflict and then **not** $hc(v_1, v_3)$ is added to A. However, it is clear that the addition of **not** $hc(v_1, v_3)$ to A is immaterial to the rest of the search, and choosing any other $hc(v_i, v_j)$ is guaranteed to lead to a solution.

4 Adaptive Lookahead

4.1 Algorithm

Adaptive lookahead is designed to avoid lookahead when its use tends to be ineffective. Two pieces of information are useful for this purpose. One is the number of conflicts (conflicts are generated by failed literals whose addition to the current partial assignment causes both a literal and its negation to be included in the partial assignment by constraint propagation. Note that conflicts do not necessarily mean backtracking is needed since the negation of the failed literal may be consistent with the current partial assignment). Another is the number of dead-ends (dead-end is a point during the search where neither a literal nor its negation is consistent with the current partial assignment. In this case backtracking is needed) detected during the search. The idea is that if after some runs, the conflicts have been rare, it is likely the search is in a space where pruning is insignificant, and likely to remain so for some time to come, so lookahead is turned off; if dead-ends have been frequently encountered after some runs, it is likely that the search has entered into a space where pruning can be significant, so lookahead is turned on.

Smodels with adaptive lookahead is called *A-smodels* (Algorithm 3). The control of lookahead is realized by manipulating two scores, $look\_score$ and $dead\_end\_counter$. The $look\_score$ is initialized to be some positive number, then deducted each time lookahead does not detect any conflict. When it becomes zero, lookahead will be turned off. The $dead\_end\_counter$ is initialized to be zero and increased each time a dead-end is encountered. Lookahead will be turned on if $dead\_end\_counter$ reaches some threshold. The counters will be re-set after each turn.

In addition to the on/off control, lookahead will never be used in the later search processes if it cannot detect any conflicts after a number of atoms have been assigned. This is because the search efficiency cannot be improved much by lookahead if the conflicts it detects only happen in late stages of the search. The lateness is measured by a ratio of number of assigned literals to all literals in the program.

The initial value for $look\_score$, the amounts of increase and decrease, and the thresholds are determined empirically. We set the amount of increase/decrease to 1, $look\_score$ to 10, the thresholds of $dead\_end\_counter$ and $ratio$ to 1 and 0.8 respectively. Note that there may not be a setting of these parameters that works universally well for all kinds of problems. The current setting is effective for the problems that we have experimented on and likely to work well for similar problems.

4.2 Comparison with limited lookahead

The idea of limited lookahead (LL) [15] is that if assuming a literal to be true does not lead to any inference, lookahead is guaranteed to be wasted. As such, lookahead is performed only on what are called *propagating literals*, the literals whose assignment leads to at least one inference.

The existence of some inferences is a condition that appears to be too weak. It is easier to incur inferences than a conflict or dead-end, so performing lookahead on propagating literals may not generate any space pruning. The

Algorithm 3 A-smodels(P, A, $look$, $shut\_down$)

```
1: reset dead_end_counter
2: A ← expand(P, A)
3: if (look) then
4:    A ← lookahead(P, A)
5:    if (no conflict detected by lookahead) then
6:       decrease look_score
7:       if (look_score = 0) then
8:          look ← false {lookahead is turned off}
9: if (conflict()) then
10:   conflict_found ← true {a conflict is found}
11:   if (a dead-end is found) then
12:      increase dead_end_counter
13:      if (!look and dead_end_counter > threshold_1
         and !shut_down) then
14:         look ← true {lookahead is turned on}
15:         reset look_score
16:   return false
17: else if (A covers Atoms(P)) then
18:   return true{A+ is a stable model}
19: else
20:   ratio ← ||A|| / ||Literal(P)||
21:   if (ratio > threshold_2 and !conflict_found)
      then
22:      shut_down ← true {shut down lookahead in later
         search}
23:   choose a new atom x to assign
24:   if (A-smodels(P, A ∪ {x}, look, shut_down)) then
25:      return true
26:   else
27:      return A-smodels(P, A ∪ {not x}, look,
         shut_down)
```

pigeon-hole problem is an illustrative example. It is known to be hard when the number of pigeons are greater than the number of holes by one. Suppose we have 10 pigeons and 9 holes, and the current partial assignment is $A = \{pos(1,1)\}$. Consider the following rule

$$\bot \leftarrow pos(2,2), pos(2,3).$$

Both $pos(2,2)$ and $pos(2,3)$ will be identified as a propagating literal, because they are unassigned, the head of the rule is $false$, and assuming either $pos(2,2)$ or $pos(2,3)$ to be true allows us to infer the other is false. But as we have commented in Subsection 3.1, lookahead on pos(2,2) or pos(2,3) does not lead to any conflict because there are more than two holes left unoccupied. Actually, the same situation arises for any $pos(i,j)$, where $2 \leq i \leq 10, 2 \leq j \leq 9$. It is easy to see that LL reduces very little overhead in this case[3]

[3]The code of the implementation of LL is unavailable at the time of this writing, so an experimental comparison cannot be performed.

The above observation is also applicable to the computation of Hamiltonian cycles on complete graphs (this is due to the second and third rules of the program given in Subsection 3.2).

Instead of identifying propagating literals, adaptive lookahead uses the information developed during the search about conflicts and dead-ends. In solving the pigeon-hole instances like the one above, adaptive lookahead will turn off lookahead after some invocations since it cannot detect any conflicts (w.r.t the current partial assignment A). For the problem of computing Hamiltonian cycles on a complete graph, lookahead is also turned off since the dead-end is rarely encountered because every choice point during the search can lead to a solution.

5 Experiments

The experiments consist of two parts. In the first part we test the general performance of adaptive lookahead. We compare A-smodels with smodels using random logic programs, which are provided as benchmarks for the first ASP solver contest [1].

The second part of the experiments serves three purposes. First, they confirm our findings of the problems where the performance is significantly deteriorated by the use of lookahead; second, they show that lookahead tends to be very effective for a number of hard problems, especially for the problems that lie in the known regions of phase transition; third, adaptive lookahead behaves as if it "knows" when to employ lookahead and when not to.

We run *smodels 2.32* with and without lookahead, and A-smodels, respectively. By default, smodels runs with lookahead. All of the experiments are run on Red Hat Linux AS release 4 with 2GHz CPU and 2GB RAM.

We shall mention that the branching heuristic in smodels is related to lookahead. The heuristic value of each atom is initialized according to how many rules they are involved in. During the computation, the heuristic value of an atom is updated according to how many other atoms whose values can be determined by the addition of the atom to the current partial assignment. With lookahead, smodels computes the heuristic value of every atom each time lookahead is invoked. Without lookahead, the heuristic value will not be computed until the atom is chosen to be assigned. So the effectiveness of lookahead should be considered on both the search procedure and the branching heuristics.

5.1 Random logic programs

There are two sets of random logic programs provided in [1]. The first is just called *random logic programs* (RLPs) and the second *random non-tight logic programs* (RNLPs).

We run A-smodels and smodels on both of them. The results are plotted in Figures 1 and 2. The graphs show the ratio of the running time of A-smodels to smodels. The instances are arranged according to the descending of this ratio.

For RLPs, we choose 110 instances randomly from the pool in [1]. A-smodels outperforms smodels on all of them except for three. The average improvement is 35% and maximum is 62%.

For RNLPs, 80 instances are chosen for the test. A-smodels is faster than smodels except for five. The average improvement is 34% and maximum is 66%. This result is different from [15], where the benefit of limited lookahead is not so obvious.

For these benchmarks, A-smodels turned on and off lookahead once, respectively.

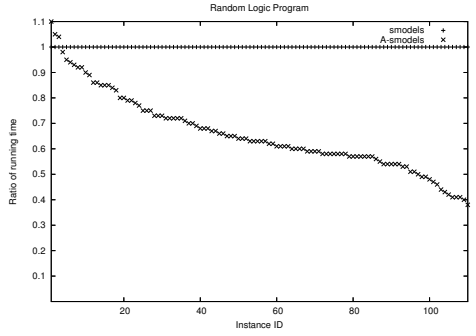

Figure 1. Random logic programs

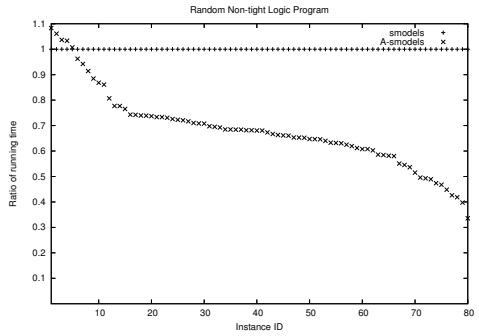

Figure 2. Random non-tight logic programs

5.2 Adaptiveness of Adaptive lookahead

5.2.1 Cases that benefit from lookahead

Graph coloring We use Culberson's flat graph generator [3] to generate graph instances. Each pair of vertexes is assigned an edge with independent identity probability p. We use the suggested value of p to sample across the "phase transition". The number of color is 3 and the number of nodes of the graph is 400. The cutoff time is 3 hours. For each measure point we generated 100 instances and the average running time is reported.

The extremely hard instances happen when p is in [0.018, 0.020]. In this region, lookahead speeds up the search drastically (Table 1). Note that smodels without lookahead cannot finish the search in the cutoff time (for $P = 0.018$, it runs for two days without completion).

3-SAT Another well-known problem with phase transition is random 3-SAT. The instances are extremely hard when the ratio $\frac{c}{a}$ is around 4.3 [14], where c is the number of clauses, and a the number of variables in the formula.

We fixed the number of variables to 300 and randomly generated conjunctive normal formulas by the ratio from 1 to 10. For each ratio, we generated 10 instances and computed the average running time. Similar to the graph coloring problem, lookahead substantially speeds up the search in the hard region (Table 2).

Blocks-world The blocks world problem is a typical planning problem. It is to rearrange a number of cubical blocks on a table from an initial configuration to a goal configuration.

The problem instances are generated as follows. For n blocks, b_1,...,b_n, the initial configuration is b_1 on the table and b_{i+1} on b_i for $i = 1...n-1$. The goal is b_n on the table, b_1 on b_n and b_{i+1} on b_i for $i = 1...n-2$. We use this setting because, under it, each block in the initial state has to be moved, so the problem turns out to be nontrivial. The minimum steps needed is $2n-2$.

The results are reported in Table 3. The upper sub-row of each row is the solvable case and the lower is the unsolvable case (similarly in Table 5).

Gripper The goal of the gripper problem is to transport balls from room $R1$ to room $R2$. To accomplish this, a robot is allowed to move from one room to the other, pick up, and put down a ball. Each gripper of the robot can hold one ball at a time.

In our experiments two kinds of settings are used. In the first, all of the balls are in $R1$ initially and $R2$ in the goal state. In the second, there are equal number of balls in $R1$ and $R2$ initially and in the goal state, balls initially in $R1$ are in $R2$ and initially in $R2$ are in $R1$.

The results are reported in Table 5, where the first two columns are the number of balls in each room initially, and the third is the number of steps allowed.

The results suggest that lookahead generates more performance gains when the instances become harder especially for the unsolvable instances.

For all of the problems in this section, A-smodels per-

forms as well as smodels with lookahead (Tables 1, 2, 3, and 5). For most instances of the graph coloring, 3-SAT, and gripper problem, lookahead is turned on and off only once. The blocks world problem is interesting. For the solvable instances, A-smodels turns off lookahead automatically and performs several times faster than smodels. Especially for the instance where the blocks number is 16, lookahead is turned off two times and on once. A drastic improvement in the performance by this action can be observed.

5.2.2 Cases that suffer from lookahead

For the problems with easy sub-programs, like pigeon-hole, lookahead can detect conflicts only near the end of the search. Employing lookahead for these problems makes the search several times slower (Table 6).

As for computing Hamiltonian cycles on complete graphs, lookahead cannot reduce the depth of the search tree while bringing on extra constraint propagations. The experimental results (Table 4) show smodels can be several hundred times slower than smodels without lookahead.

For the above two problems, A-smodels works largely as smodels without lookahead. Its performance is slightly better than smodels without lookahead for the pigeon-hole problem (Table 6). A-smodels turned off lookahead permanently after some literals are assigned during the process of solving this problem.

For the Hamiltonian cycle problem, the performance of A-smodels is in between smodels and smodels without lookahead - it is about 20 to 30 times faster than smodels and 2 to 10 times slower than smodels without lookahead (Table 4). We notice that if the parameter $look\_score$ is set to be 2 and $ratio$ 0.1, A-smodels will perform two times faster than what we reported here but it is still several times slower than smodels without lookahead. The reason for that could be the effect of lookahead on branching heuristic as we mentioned before. More investigation on this is needed.

6 Summary and Future Directions

In this paper, we show that lookahead could be a burden in, as well as an accelerator to, the DPLL search. We analyze why lookahead sometimes slows down the search and characterize the reasons as embedded easy sub-programs and spurious pruning. Based on this analysis, we propose an adaptive lookahead mechanism, by which the decision on whether lookahead is invoked or not is made dynamically upon the learned information developed during the search. It takes the advantage of lookahead while avoiding the unnecessary overhead caused by it.

A question left open is how the performance of A-smodels compares to the recent answer set solvers enhanced with look-back techniques [8, 11].

p	No Lookahead	Lookahead	A-Lookahead
.014	0.33	0.33	0.09
.015	1.44	0.33	0.15
.016	53.55	0.37	0.32
.017	97.12	0.24	0.12
.018	–	209.68	183.41
.019	–	43.30	40.15
.020	–	55.38	23.34
.021	2649.80	9.47	10.14
.022	299.61	0.96	1.06
.023	18.89	0.92	1.05
.024	4.19	0.25	0.67
.025	3.52	0.23	0.22
.026	3.26	0.26	0.22
.027	1.97	0.21	0.29
.028	2.13	0.21	0.26
.029	2.06	0.17	0.29
.030	1.25	0.19	0.22

Table 1. Graph coloring

c/a	No Lookahead	Lookahead	A-lookahead
1	0.01	0.04	0.01
2	0.02	0.05	0.02
3	0.08	0.06	0.05
4.3	6475.43	137.64	139.88
5	650.85	20.20	19.72
6	45.09	1.64	1.55
7	7.12	0.55	0.54
8	3.15	0.38	3.25
9	1.68	0.27	1.61
10	0.94	0.21	1.04

Table 2. Random 3-SAT

References

[1] http://www.asparagus.cs.uni-potsdam.de/.

[2] J. E. Borrett, E. P. Tsang, and N. R. Walsh. Adaptive constraint satisfaction: The quickest first principle. In *TR CSM-256, University of Essex*, 1995.

[3] J. Culberson. http://web.cs.ualberta.ca/joe/coloring/index.html.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Commun. ACM*, 5(7):394–397, 1962.

[5] H. Dixon and M. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Proc. AAAI'02*, pages 635–641, 2002.

[6] T. L. et al. noMoRe: a graph-based system for nonmonotonic reasoning with logic programming under answer set semantics. http://www.cs.uni-potsdam.de/ linke/nomore/.

n	s	No Lookahead	Lookahead	A-Lookahead
11	20	13.58	4.62	2.02
	19	16.52	1.56	1.56
12	22	37.49	9.38	3.36
	21	30.70	2.42	2.41
13	24	106.31	17.26	4.85
	23	62.31	3.81	3.76
14	26	165.88	30.49	7.34
	25	359.65	5.45	5.46
15	28	335.11	53.07	10.54
	27	4673.35	7.35	7.35
16	30	375.20	6276.28	14.66
	29	8585.08	10.15	10.50
17	32	1197.65	145.59	21.24
	31	701.86	16.48	16.40
18	34	–	145.48	29.28
	33	–	21.42	21.39

Table 3. Blocks-world problem

n	No Lookahead	Lookahead	A-Lookahead
30	0.46	3.62	1.07
40	1.11	16.90	3.19
50	2.25	64.92	8.57
60	3.91	183.59	19.74
70	6.31	467.28	40.18
80	9.58	986.85	74.02
90	13.77	1862.86	123.66
100	19.12	3522.24	194.24
110	25.56	6129.59	288.53
120	33.36	7255.97	412.58

Table 4. Hamiltonian cycle

R1	R2	s	No Lookahead	Lookahead	A-Lookahead
4	0	7	0.10	0.43	0.41
		6	0.09	0.17	0.17
5	0	11	8.31	77.35	64.94
		10	1824.55	189.40	97.12
6	0	11	263.75	1117.02	1084.31
		10	2345.49	490.94	487.65
3	3	11	0.70	15.80	15.98
		10	77.93	15.64	15.52
4	4	12	1749.35	419.42	412.64
		11	5463.57	175.61	175.73

Table 5. Gripper problem

p	h	No Lookahead	Lookahead	A-Lookahead
5	4	0.00	0.01	0.01
6	5	0.00	0.02	0.01
7	6	0.02	0.06	0.02
8	7	0.13	0.49	0.11
9	8	1.18	4.38	0.94
10	9	12.10	43.66	9.78
11	10	137.06	480.19	112.47
12	11	1608.41	5439.09	1343.93

Table 6. Pigeon hole

[7] J. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.

[8] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proc. IJCAI'07*, pages 386–392, 2007.

[9] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP'88*, pages 1070–1080, 1988.

[10] E. Ginuchiglia, M. maratea, and A. Tacchella. (in)effectiveness of look-ahead techniques in a modern sat solver. In *Proc. CP'03*, pages 842–846, 2003.

[11] J.Ward and JSchlipf. Answer set programming with clause learning. In *Proc. ICLP'04*, pages 302–313, 2004.

[12] E. Lamma, P. Mello, and M. Milano. An incremental consistency algorithm for adaptive constraint satisfaction. In *TR DEIS-LIA-97-002, University of Bologna*, 1997.

[13] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):1–57, 2006.

[14] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In *Proc. of AAAI'92*, pages 459–465, 1992.

[15] G. Namasivayam and M. Truszczyński. An $smodels$ system with limited lookahead computation. In *Proc. LPNMR'07*, pages 278–284, 2007.

[16] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Math. and Artificial Intelligence*, 25(3-4):241–273, 1999.

[17] H. E. Sakkout, M. G. Wallace, and E. B. Richards. An instance of adaptive constraint propagation. In *Proc. CP'96*, pages 164–178, 1996.

[18] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[19] A. van Gelder, K. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proc. PODS*, pages 221–230, 1988.

[20] J. You and G. Hou. Arc consistency + unit propagation = lookahead. In *Proc. ICLP'04*, pages 314–328, 2004.