Using Criticalities as a Heuristic for Answer Set Programming

Orkunt Sabuncu[1], Ferda N. Alpaslan[1], and Varol Akman[2]

[1] Department of Computer Engineering
Middle East Technical University,
06531 Ankara, Turkey
{orkunt,alpaslan}@ceng.metu.edu.tr
[2] Department of Computer Engineering
Bilkent University,
06800 Ankara, Turkey
akman@cs.bilkent.edu.tr

Abstract. Answer Set Programming is a new paradigm based on logic programming. The main component of answer set programming is a system that finds the answer sets of logic programs. During the computation of an answer set, systems are faced with choice points where they have to select a literal and assign it a truth value. Generally, systems utilize some heuristics to choose new literals at the choice points. The heuristic used is one of the key factors for the performance of the system.
A new heuristic for answer set programming has been developed. This heuristic is inspired by hierarchical planning. The notion of criticality, which was introduced for generating abstraction hierarchies in hierarchical planning, is used in this heuristic. The resulting system (CSMODELS) uses this new heuristic in a static way. CSMODELS is based on the system SMODELS. The experimental results show that this new heuristic is promising for answer set programming. A comparison of search times with SMODELS demonstrate CSMODELS' usefulness.

1 Introduction

Answer Set Programming is a new programming paradigm. It is based on logic programming but solutions of a problem are not extracted from a proof session [1]. It is a model-theoretic approach. One writes a logic program for the problem at hand in such a way that the intended models of the program correspond to the solutions of the problem. The semantics used for selecting the intended models in answer set programming is *stable model semantics*. The models found by the semantics are called *stable models* or *answer sets* and each of the models correspond to a solution of the problem.

The main component of answer set programming is a system that finds the answer sets. Such systems can be seen as the implementations of stable model semantics.

The main algorithm for finding an answer set is common to almost all systems. The idea is a simple generate-and-test cycle [2]. In the generation phase,

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 234–246, 2004.

a candidate model is constructed for the input logic program. This model is a partial model. The aim is to transform this candidate model into an answer set. The system periodically checks whether the model at hand is an answer set while augmenting it in the test phase.

At *choice points* (see Sect. 3) the system chooses an uncovered atom and assigns it *true* or *false* as an interpretation to augment the candidate model. Some choices cause the system find an answer set very quickly, but some cause it to enter an incorrect search path and consume lots of time before it backtracks. So, heuristics are usually used for choosing an uncovered atom. They greatly affect the performance of the system.

Working on new heuristics is important for developing better systems for answer set programming [2]. In this work, a new system called CSMODELS[1] (Criticality SMODELS), which is based on SMODELS [3], is developed. CSMODELS uses a new heuristic whose foundation is *criticality*. The notion of criticality has been used in hierarchical planning [4].

Hierarchical planning is a way to deal with search space complexity of planning problems. Hierarchical planners attack the problem at different levels of detail. An *abstraction hierarchy* is used to define these levels.

There are successful abstraction hierarchies that improve the performance of the hierarchical planner, but there are also poor ones which cause considerable backtracking between levels. In a good abstraction hierarchy, the upper levels (the most abstract ones) should deal with the hardest part of the planning problem so the planner tries to solve them first. This property will limit the amount of backtracking between the levels during the plan finding process [5]. The notion of criticality was introduced in [4] for automatically generating abstraction hierarchies based on this property. Criticality of a literal in a planning problem is a numerical value and approximates the difficulty of finding a plan that achieves this literal.

There is also the difficulty of finding a literal in an answer set. Selecting the 'hardest' literals at the first choice points can limit backtracking and can help the system find an answer set quickly as in the case of hierarchical planning. This is the main motivation for our work. In CSMODELS, criticalities of the literals of a logic program are calculated to approximate the difficulty of finding them in an answer set. Then, these values are used as a heuristic at the choice points. It is not trivial to calculate the criticalities this time because the original method was for planning problems, not for logic programs. A method for applying criticality calculation for answer set programming is also developed.

The experimental results obtained with CSMODELS are encouraging. Generally, CSMODELS finds an answer set of a program more efficiently than SMODELS. This increase in the performance of search time is significant for especially large problems.

Background information about criticalities and how to calculate them are given in the next section. Section 3 describes the main algorithm of SMODELS. The essential contribution of this work, which is applying the notion of criti-

[1] http://www.ceng.metu.edu.tr/~orkunt/csmodels/ .

cality to answer set programming, is presented in Section 4. Section 5 describes CSMODELS. Section 6 includes the experimental results. In the final section, conclusions can be found.

2 Criticalities

Criticalities are used for generating abstraction hierarchies for planners. Planners use abstraction to reduce the complexity of the planning problem [6,7]. ABSTRIPS, ABTWEAK, ALPINE, and RESISTOR are some hierarchical planners [6].

Hierarchical planners use an abstraction hierarchy for the planning problem to generate and solve subproblems. An *abstraction hierarchy* divides the whole problem into pieces. A hierarchy level indicates which parts of the problem should be neglected or taken into account to form a subproblem corresponding to that level. After finding an abstract plan, the next step is to go one level below and find a plan for that level. The information that is neglected in the higher level is considered now as part of the problem. This is called *refinement*. This process continues level by level until the ground level (i.e., the lowest level) is reached.

Results of using abstraction in planning are generally encouraging. In some problems it can lead to an exponential reduction in the search space, improving the efficiency of plan finding [8].

There are also discouraging results [9,10]. The main cause of inefficiency is *trashing* [5] between the levels of abstraction hierarchy. During refinement, if no plan can be found in a level, backtracking to a more abstract level (the upper one) to search for another abstract plan is inevitable. There may be many possible abstract plans which cannot be refined. This will cause numerous backtrackings, leading to trashing. A good abstraction hierarchy should avoid trashing. To limit trashing, a system should try to solve a subproblem which constitutes the hardest part of the original problem first [5].

Bundy, Giunchiglia, Sebastini, and Walsh [4] provide a method for generating good hierarchies and provide details of an implementation called RESISTOR. RESISTOR sorts the precondition literals of a planning problem according to their difficulties to achieve (i.e., their costs). So, it partitions the problem into parts in terms of difficulty to generate a good abstraction hierarchy.

The method discussed in [4] uses a numerical simulation of the plan finding process. This method has introduced the notion of *criticality*. Criticality captures the cost of a literal; criticality of a literal is an approximation of the cost of achieving it. Criticalities have numerical values; smaller values correspond to easier-to-achieve literals, larger values correspond to harder-to-achieve literals.

The *criticality function* $C(p, n)$ gives the criticality value of literal p. There is an interpretation of $C(p, n)$ as *the difficulty of finding a plan of length* $\leq n$ *achieving* p. This interpretation gives rise to a group of criticality function definitions.[2] Our work is based on RESISTOR's criticality functions. After defining

[2] Other definitions and detailed information can be found in [4].

the criticality functions, criticality values are calculated in an iterative way. The iteration is on n starting from 0. So, we basically see $C(p, n)$ as a function that gives the criticality value of literal p at the n-th iteration. Note that the variable n refers to plan length in the interpretation just to find criticality function definitions.

RESISTOR's definition of $C(p, n)$ is inspired by electrical resistors. In calculating the difficulty of achieving p, operators whose effect is p can be regarded as electrical resistors connected in parallel. Fig. 1 shows this circuit. The more operators there are, the more paths there will be for achieving p. Consequently, finding p becomes less difficult. As in equation (1), $C(p, n)$ is calculated by the parallel sum (total resistance of parallel connected resistors) of the criticalities of operators whose effect is p and the initial criticality $C(p, 0)$ (the difficulty of finding a plan of length 0).

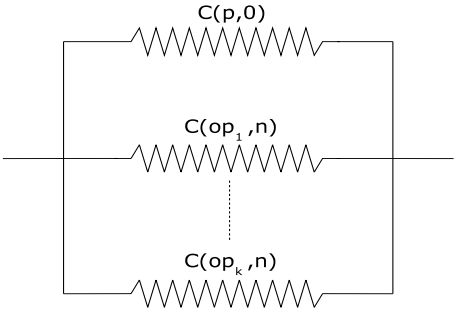

Fig. 1. Criticality circuit for the precondition p

$$\frac{1}{C(p,n)} = \frac{1}{C(p,0)} + \frac{1}{C(op_1,n)} + \ldots + \frac{1}{C(op_k,n)} \tag{1}$$

Equation (1) introduces *the notion of criticality of an operator*. $C(op, n)$ for an operator op is defined to make calculations simple. It is interpreted as the *difficulty of finding a plan of length 1 to n which ends with the occurrence of operator op*. For calculating the criticality of an operator, preconditions of that operator can be regarded as electrical resistors connected in series (Fig. 2).

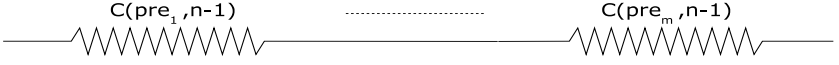

Fig. 2. Criticality circuit for the operator op

$$C(op, n) = C(pre_1, n-1) + \ldots + C(pre_m, n-1) \tag{2}$$

Since the interpretation of $C(op, n)$ expects the occurrence of *op* at the n-th step, all the preconditions must be achieved up to that point. This is why criticalities of preconditions at the $(n-1)$-th iteration are used in equation (2).

Using equations (1) and (2), criticalities of all the precondition literals are calculated iteratively. Initial criticality values for every literal (i.e., $C(p, 0)$ terms) are assigned to 1. Since criticality functions are monotonically decreasing and convergent [4], the limiting values of criticalities will lie in the interval [0,1] after some iterations.[3] RESISTOR continues to iterate until no change occurs in the values (within some predefined *accuracy*). This is a computationally practical way of terminating iterations.

Abstraction hierarchies are generated by sorting the final criticality values. However, what is important for our work is just how they are calculated.

3 SMODELS: An Answer Set Programming System

SMODELS has been developed at Helsinki University of Technology, and is now one of the most popular answer set programming systems. The system implements stable model semantics [11] for logic programs. Reference [3] explains SMODELS' philosophy and implementation in detail.

Input to SMODELS is variable free logic programs. Programs with variables are transformed to ground logic programs by a front-end system called LPARSE[4] [12]. SMODELS uses a candidate model which is an empty set at the beginning. It tries to augment this candidate model by adding literals deterministically according to the program by using the properties of stable model semantics. This process of generating new literals deterministically is equivalent to *expanding* the model at hand.

Expanding a model can cause situations in which both an atom and its negation are in the model at the same time. This contradicting situation is called a *conflict*.

At an intermediate stage of the search process, an atom can take one of three different possible values [2]: *true*, *false*, or *undefined*. If an atom has a value of undefined, it is an *uncovered* atom.

At the end of first expansion the candidate model corresponds to a *well-founded model* [13] of the program. The aim is to expand the candidate model until it becomes a stable model. The decision criterion for a candidate model to be a stable model is that all the atoms of the program should be covered by it and there should be no conflicts. At several stages during the whole search process there are uncovered atoms and SMODELS cannot deterministically assign values to these atoms (i.e., true or false). These stages are called *choice points*. At choice points, SMODELS should just select one of the uncovered atoms and give an interpretation as true or false. Then, the main algorithm will again try to expand the newly generated candidate model with the addition of the chosen

[3] Actually, if the initial criticality values are 1, then all the criticalities will be in this interval.

[4] Available at http://www.tcs.hut.fi/Software/smodels/ .

atom. Remember that there can be conflicts after expansion. An incorrect choice at a previous choice point can be the reason for a conflict. So, in case of a conflict SMODELS backtracks to the previous choice. Fig. 3 shows a simplified version of SMODELS' main algorithm [3].

```
function smodels(CM)   {CM: Candidate Model}
  CM := expand(CM)
  if conflict(CM) then
       return false
  else if no atom is undefined in CM then
       return true   {CM is a stable model}
  else
       c := Choose an uncovered atom
       if smodels(CM ∪ {c}) then
            return true
       else
            return smodels(CM ∪ {not c})
       end if
  end if
```

Fig. 3. Main algorithm of SMODELS

SMODELS' heuristic to choose an atom is based on minimizing the remaining search space after a choice is made [3]. SMODELS selects every uncovered literal temporarily and adds it to the candidate model at a choice point. It does so in order to determine what happens if it chooses that uncovered literal actually. After expanding the candidate model, there will possibly be newly derived literals. The number of these new literals is the heuristic score of the chosen literal. If the chosen literal is positive then the score of the literal is called *positive score* of that atom. Similarly, the score of a negative literal of the same atom is *negative score* of that atom.

Given the minimum of positive and negative scores of each atom, SMODELS' heuristic chooses the maximum one to guarantee that it reduces the search space maximally.

The heuristic of SMODELS is a *dynamic heuristic* [2]: at every choice point it recomputes the scores.

4 Applying the Criticality Notion for Answer Set Programming

Criticalities make it possible to find hard-to-achieve literals and to work on them first during hierarchical plan finding. This will reduce the amount backtracking between the levels of abstraction hierarchy.

The same idea can also be applied to answer set programming. During the answer set finding process, choosing the hard-to-find literals at the early choice

points can reduce the backtracking and permits the system to find an answer set quickly. There are costs of finding literals in an answer set similar to the costs of achieving literals in a planning problem. Knowing the costs of literals of an input logic program can be useful for a system at the choice points. This intuition is the main motivation for the heuristic developed in our work.

Every atom in the stable model has to be *grounded*. If an atom has at least one rule that has generated it, then it is grounded. Rules that have an atom in the head are possible *generators* of it.

The only condition for a rule to generate the literal in its head is that its body must be true. So, if we accept the rule's body as a *precondition* we can transform a rule into a *planning operator (action)*.[5] The *effect* of this planning operator is the head atom. Let all the generator rules for literal a below constitute a portion of a logic program.

$$a \leftarrow b,\ not\ c \quad (rule_1) \tag{3}$$

$$a \leftarrow d \quad (rule_2) \tag{4}$$

The corresponding planning operator for rule (3) is given below.

$$a \leftarrow^{r_1} b,\ not\ c \quad \Longrightarrow \quad \begin{array}{l} \text{Operator : } r_1 \\ \text{Preconditions : } b,\ not\ c \\ \text{Effect : } a \end{array}$$

When we consider the above rules as planning actions, equations (5–7) are used for calculating the criticality of literal a. Note that the initial criticality of a, $C(a, 0)$, is set to 1 like all the other literals' initial criticalities.

$$C(rule_1, n) = C(b, n-1) + C(not\ c, n-1) \tag{5}$$

$$C(rule_2, n) = C(d, n-1) \tag{6}$$

$$\frac{1}{C(a,n)} = \frac{1}{C(a,0)} + \frac{1}{C(rule_1,n)} + \frac{1}{C(rule_2,n)} \tag{7}$$

There is a term representing criticality value of a negative literal in equation (5). How can the criticality of a negative literal be found? There are no rules that can generate a negative literal in the sense of generating positive ones. In fact *negation-as-failure* means that if it is not possible to achieve an atom a, then *not* a is assumed to be true. Considering the properties of stable model semantics, we can define the necessary conditions for *not* a to be in the stable model as no generator rule will have a chance to generate a. The bodies of all generator rules must be interpreted as false, so literal *not* a is interpreted as true.

[5] Lin and Reiter [14] define logic rules as planning actions in the context of defining semantics for logic programs using situation calculus.

Here is the only rule generating *not a* formed by taking all the generator rules (3–4) of a into consideration:[6]

$$not\ a \leftarrow (not\ b \vee c),\ not\ d \quad (rule_{not\ a}) \tag{8}$$

Based on rule (8), the criticality of literal *not a* is calculated using the following equations:

$$C(rule_{not\ a}, n) = C((not\ b \vee c), n-1) + C(not\ d, n-1) \tag{9}$$

$$\frac{1}{C(not\ a, n)} = \frac{1}{C(not\ a, 0)} + \frac{1}{C(rule_{not\ a}, n)} \tag{10}$$

Calculating the criticality value of a negative literal is no different than calculating a positive literal except for compound literals with disjunction. Achieving a compound literal as a whole should be easier than achieving each one of the basic literals individually. So, the criticality value of a compound literal should be less than each of the criticalities of its basic literals. Calculating the criticality of compound literal $(notb \vee c)$ by the equation $C((notb \vee c), n) = C(notb, n) \times C(c, n)$ is a meaningful approximation, since the multiplication of two or more real numbers in the interval [0,1] is always smaller than each number (or equal to the smallest one at least).

A *constraint rule* is a rule that has no atom in the head (i.e., a rule that has an implicit false in the head). They are not used in criticality calculations, since they cannot be generators for any atom.

Using the above equations, we can calculate the criticalities of all the literals of a logic program if all the rules in the program are normal. But SMODELS enlarges the syntax and semantics of logic programs by extended rules [3]. These extended rules are *choice, cardinality,* and *weight* rules. Our system, CSMODELS, supports choice rules, but not cardinality or weight rules.

$$\{a\} \leftarrow b,\ not\ c \tag{11}$$

Rule (11) is a choice rule. A choice rule's head may not be true although its body is satisfied by the model unlike normal rules. One can rewrite the choice rule using only normal rules. But this translation introduces new atoms to the input program [3]. Choice rule (11) can be rewritten as two rules:

$$\begin{array}{l} a \leftarrow not\ c\_a,\ b,\ not\ c \\ c\_a \leftarrow not\ a,\ b,\ not\ c \end{array} \tag{12}$$

The atom *c_a* is a newly introduced atom. By treating choice rules as if two normal rules of the form (12), we can handle choice rules for criticality calculations. In this way we do not enlarge the program. Answer sets in the output do not contain the introduced atoms, since they are only used for criticality calculations.

[6] Body of the rule (8) is the inverse of the completion of literal a [15]. We can see the inverse of the completion of literal x as a generator rule for literal *not x*.

5 CSMODELS (Criticality SMODELS)

CSMODELS uses criticalities as a heuristic and is based on the SMODELS system. It implements the method of applying criticality calculation to logic programs of SMODELS described in Sect. 4. Implementation details of CSMODELS can be found in [16]. The main algorithm is the same as that of SMODELS except the parts related to the choice points.

CSMODELS calculates the criticality values at the first choice point. Unlike SMODELS, CSMODELS' heuristic is not dynamic. So, the same heuristic scores calculated at the first choice point are used at all the other choice points. At the first choice point, the candidate model at hand corresponds to a well-founded model (WFM) of the program. WFM covers some of the atoms of the input program, so there is no need to calculate the criticality values of the already covered atoms. We set the criticality of a literal in WFM to 0, indicating that it is very easy to achieve that literal or the literal is already achieved. Also, we set the criticality of the inverse of that literal to 1 indicating the impossibility or difficulty of achieving the literal.[7]

$$\text{If literal } l \in \text{WFM} \quad \Longrightarrow \quad \begin{array}{l} C(l) = 0 \\ C(not\, l) = 1 \end{array}$$

After calculating criticality values of all uncovered literals, CSMODELS finds the *criticality heuristic scores* to select a literal at the choice points. It temporarily selects every uncovered literal one by one and adds them to the candidate model just like SMODELS computes heuristic scores. Let x be an uncovered literal by the WFM, the expansion of the model formed by adding x to WFM will probably generate new literals that have been uncovered previously. Let these literals be

$$NewLiterals(x) = \{a,\, b, \ldots,\, not\, k,\, not\, l, \ldots\}. \tag{13}$$

CSMODELS finds the criticality heuristic score of literal x by the following equation:

$$\begin{array}{rl} Score(x) = & C(a) + (1 - C(not\, a)) + \\ & C(b) + (1 - C(not\, b)) \; + \ldots + \\ & C(not\, k) + (1 - C(k)) + \\ & C(not\, l) + (1 - C(l)) \; + \ldots \end{array} \tag{14}$$

The purpose of the score of a literal is to approximate the *difficulty of choosing that literal at a choice point*. Knowing that choosing x generates literal a, the system faces the cost of achieving a and avoiding the generation of *not* a when choosing x. That is why the terms $C(a)$ and $1 - C(not\, a)$ are added to the score. This is done for all the literals in $NewLiterals(x)$. Just like SMODELS, there are positive and negative criticality heuristic scores of an atom.

[7] Note that 1 is the maximum criticality that a literal can have.

The set *NewLiterals* for an uncovered literal can be different at different choice points. Thus, the same literal can have different criticality heuristic scores at different choice points. However, if we calculate the criticality values and the criticality heuristic scores at every choice point, the proportion of the time used by these calculations to total search time will be high. Because of these costly calculations, CSMODELS uses a static heuristic.

In order to limit the amount of backtracking, CSMODELS selects hard-to-choose literals first. So, the system sorts the uncovered atoms according to their heuristic scores. The *sorting criterion* affects the performance of the heuristic. The main sorting criterion of CSMODELS, which is called *maxmin* sorting criterion, is the same as that of SMODELS. Atoms are sorted according to minimum of their positive and negative criticality heuristic scores in descending order. Sorting atoms according to the sum of positive and negative heuristic scores in descending order is another criterion named *maxsum*. Experiments showed that using maxsum helped CSMODELS to find an answer set in time less than using maxmin for problems having many answer sets.

Fortunately, using maxsum in problems that do not have many answer sets usually leads to conflict at the first choice point. CSMODELS first uses the maxsum criterion. If it ends up with an immediate conflict, it switches back to maxmin. This causes CSMODELS to behave identically for all problems.

6 Experimental Results

CSMODELS has been tested to find out its performance. Test problems are taken from common application domains of answer set programming. We compare results of CSMODELS with those of SMODELS. Since both systems take the same ground logic program for the experiments, grounding time is the same for both. The grounding times are not included in the search time results of the experiments.

The main measure for the tests is the duration which states how long the search for an answer set took in CPU seconds. Another measure for comparison is the number of choice points. This number shows how many times a system used its heuristic to find an answer. The same ratio between the number of choice points may not be observed for the search times in comparisons of CSMODELS and SMODELS since there is not a direct correlation. However, the reduction in the number of choice points generally leads to performance gains in terms of search times.

Tests for planning problems have been performed on a 200 MHz Pentium computer with 256 MB of main memory running Linux 2.4.5; for colorability and n-queens problems a 733 MHz Pentium III computer with 256 MB of main memory running Linux 2.2.19 has been used.

Several well-known planning problems are tested. These are Towers of Hanoi, simple robot-box domain, and blocks-world planning. In addition to planning problems, several tests of colorability and n-queens problems have been performed. If the original logic programs for these problems have cardinality rules,

they are rewritten using normal rules without affecting the answer sets. Tables 1, 2 and 3 report the results of CSMODELS and SMODELS for all the problem instances. The results for choice points are shown in parentheses in tables.

Towers of Hanoi problem with 4 disks has been tested. It can be solved in 15 steps optimally. The logic program representing the simple robot–box domain is based on the reference [6].[8] The optimal solution has 13 steps. For the blocks-world planning problem, two different domain representations have been tested. One representation is taken from the reference [17] where concurrency is allowed. Three different problem instances with 15, 17, and 19 blocks from the reference [18] are tested (rows BW_2 (15), BW_2 (17) and BW_2 (19)).[9] They have optimal plans with 8, 9 and 10 steps, respectively. Another representation is the one used in reference [19]. A problem instance with 11 blocks from the same reference has been used (row BW_1 (11)).[10] It has an optimal solution with 9 steps. Unlike the first representation, this one does not allow concurrent moves.

Table 1. A comparison of search time (CPU seconds) and number of choice points for planning problems

	SMODELS	CSMODELS
Hanoi	4.630 (12)	5.540 (8)
Robot–Box	39.940 (11)	40.800 (8)
BW_1 (11)	73.400 (7)	57.040 (5)
BW_2 (15)	38.500 (29)	37.390 (7)
BW_2 (17)	77.330 (27)	61.550 (10)
BW_2 (19)	174.660 (4111)	90.620 (6)

The results show that for small size problems like Towers of Hanoi and simple robot-box domains, CSMODELS' performance is almost the same as SMODELS' performance. When the sizes of the problems become larger, criticality heuristic of CSMODELS helps the system solve the problems more efficiently. This claim is supported especially by the problem instances of the blocks-world representation BW_2. By increasing the number of blocks from 15 to 19, the performance difference between CSMODELS and SMODELS becomes obvious. Also the reduction in the number of choice points is consistent with the performance gains.

Other than the planning domain, colorability (4-colorability problem instances are used in experiments) and n-queens problems have been used in testing. The logic programs for these problems are based on I. Niemela's representations.[11]

[8] This domain is about controlling a robot to move boxes between rooms.

[9] They correspond to I. Niemela's bw-large.c, bw-large.d and bw-large.e problems, respectively [18].

[10] This problem instance corresponds to E. Erdem's P4 [19].

[11] The logic program for n-queens problem is actually from [17]. The one for colorability problem is adapted from [18] by rewriting choice rules.

Table 2. A comparison of search time (CPU seconds) and number of choice points for colorability problem

	p100	*p300*	*p600*	*p1000*	*p3000*
SMODELS	0.530 (32)	3.530 (89)	12.380 (177)	35.210 (310)	305.060 (860)
CSMODELS	0.540 (38)	3.050 (79)	10.080 (162)	26.070 (235)	230.370 (787)

Table 3. A comparison of search time (CPU seconds) and number of choice points for n-queens problem

	8×8	*18×18*	*20×20*	*22×22*
SMODELS	0.020 (3)	1.830 (302)	9.360 (1483)	117.730 (16156)
CSMODELS	0.030 (7)	1.530 (152)	5.490 (571)	31.170 (2401)

Several tests have been carried out ranging from small size problem instances to large sized ones for colorability and n-queens. Results show that CSMODELS is more efficient than SMODELS for colorability and n-queens problems. For larger sized problem instances, the performance gains obtained by CSMODELS are more significant.

7 Conclusion

Answer set programming systems utilize some heuristics to compute answer sets. Heuristics affect the search path that the system explores within the entire search space. As some paths lead to efficient solutions and some not, heuristics are one of the key ingredients influencing the performance of a system.

In our work, a new heuristic for answer set programming has been developed. The resulting system called CSMODELS uses this new heuristic. The main insight of the heuristic comes from hierarchical planning. The notion of criticality, which is introduced for generating abstraction hierarchies, is applied for answer set programming in the new heuristic. The experimental results indicate that CSMODELS outperforms SMODELS in terms of efficiency. The performance difference becomes more significant when the problem size gets larger. Generally, the results show that this new heuristic is promising for answer set programming.

The choice rules from the extended rules of SMODELS' new versions are supported by CSMODELS. However, cardinality and weight rules are not supported. Rewriting cardinality and weight rules using normal rules lead to an exponential growth in the number of rules. Handling cardinality and weight rules in CSMODELS is a topic for future work.

Heuristics for answer set programming are similar to the heuristics developed for satisfiability solvers. Although there is a substantial amount of work done for the latter, there is not much for the former [2]. New heuristics will help systems to be more efficient and make answer set programming more applicable.

References

1. V. Lifschitz. Answer set planning. In *International Conference on Logic Programming*, pages 23–37, 1999.
2. W. Faber, N. Leone, and G. Pfeifer. A comparison of heuristics for answer set programming. In *Proc. of the 5th Dutch-German Workshop on Nonmonotonic Reasoning Techniques and their Applications (DGNMR 2001)*, pages 64–75, 2001.
3. P. Simons. Extending and implementing the stable model semantics. Research Report 58, Helsinki University of Technology, Helsinki, Finland, 2000.
4. A. Bundy, F. Giunchiglia, R. Sebastiani, and T. Walsh. Calculating criticalities. *Artificial Intelligence*, 88(1–2):39–67, 1996.
5. F. Giunchiglia. Using ABSTRIPS abstractions – where do we stand? Technical Report 9607–10, IRST, Trento, Italy, 1996.
6. C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.
7. F. Giunchiglia, A. Villafiorita, and T. Walsh. Theories of abstraction. *AI Communications*, 10(3–4):167–176, 1997.
8. C. A. Knoblock. Abstracting the Tower of Hanoi. In *Working Notes of AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions*, pages 13–23, 1990.
9. D. E. Smith and M. A. Peot. A critical look at Knoblock's hierarchy mechanism. In *Proc. of 1st International conference Artificial Intelligence Planning Systems (AIPS-92)*, pages 307–308, 1992.
10. C. Backstrom and P. Jonsson. Planning with abstraction hierarchies can be exponentially less efficient. In *Proc. of the 14th International Joint Conference on Artificial Intelligence*, pages 1599–1604, 1995.
11. M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. The MIT Press, 1988.
12. T. Syrjanen. LPARSE 1.0 user's manual. Available at http://www.tcs.hut.fi/Software/smodels/lparse.ps.
13. A. van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
14. F. Lin and R. Reiter. Rules as actions: A situation calculus semantics for logic programs. *Journal of Logic Programming*, 31(1–3):299–330, 1997.
15. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.
16. O. Sabuncu. Using criticalities as a heuristic for answer set programming. MS Thesis, Middle East Technical University, Department of Computer Engineering, Ankara, Turkey, 2002.
17. I. Niemela and M. Trunszczynski. Answer-set programming: a declarative knowledge representation paradigm. In Lecture notes of ESSLLI 2001 Summer School, 2001.
18. I. Niemela. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.
19. E. Erdem. Applications of logic programming to planning: Computational experiments. Unpublished draft, http://www.cs.utexas.edu/users/esra/papers.html, 1999.