LPSP: A Linear Plan-level Stochastic Planner

Ronen I. Brafman
Department of Math and CS
Ben-Gurion University
Beer Sheva, Israel 84105
brafman@cs.bgu.ac.il

Holger H. Hoos
Department of Computer Science
Darmstadt University of Technology
D-62483 Darmstadt, Germany
hoos@informatik.tu-darmstadt.de

Craig Boutilier
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4, Canada
cebly@cs.ubc.ca

Abstract

We describe LPSP, a domain-independent planning algorithm that searches the space of linear plans using stochastic local search techniques. Because linear plans, rather than propositional assignments, comprise the states of LPSP's search space, we can incorporate into its search various operators that are suitable for manipulating plans, such as plan-step reordering based on action dependencies, and limited forward/backward search. This, in turn, leads to a flexible planning algorithm that outperforms the SATPLAN planner on difficult blocks world problems.

1 Introduction

The last several years have witnessed the emergence of a number of novel classical planning algorithms, including Ginsberg's approximate planning [Gin96], Blum and Furst's GRAPHPLAN [BF95], Kautz and Selman's SATPLAN [KS96] constraint-based planning [JP96] and refinement planning [KKY95]. While considerable research has been directed in the last decade toward the development of least-commitment planners that search in plan space—with a notable lineage defined by TWEAK [Cha87], SNLP [MR91] and UCPOP [PW92]—these new algorithms employ ideas that differ, sometimes considerably, from those underlying more classic work in generative planning.

SATPLAN, in particular, is a very different form of planner, seemingly relying on almost none of the intuitions used to develop state-space or plan-space planners in the past. While similar in some respects to constraint-based planning, it embodies none of the concepts usually used to direct the search for plans, such as projection, regression, means-ends analysis, producers, consumers, causal links, threats, clobberers, and so on (at least not explicitly). Instead, using a propositional encoding of a planning problem, it searches stochastically for a satisfying assignment, from which a plan can be extracted.[1] On many problems, it outperforms other planning approaches by orders of magnitude.

While SATPLAN's success is certainly good news for those concerned with the prospects for generative planning, one cannot but admit a certain disappointment that many of our intuitions about planning, embodied in such elegant algorithms such as SNLP and UCPOP, may have outlived their usefulness. Although many new and interesting questions arise in the context of satisfiability-based planning, one might hope that notions such as means-ends analysis, threats, causal links, etc. might still have a role to play in computationally effective planning.[2]

In this paper, we propose a novel planning algorithm that, like SATPLAN, is based on stochastic local search; but unlike SATPLAN, our method searches over the space of linear plans and uses explicit plan construction steps—involving means-end analysis, projection and threat removal—to determine successors in state space. The *LPSP* algorithm (linear, plan-space, stochastic planner) thus requires search operators that are substantially more expensive computationally than those of SATPLAN, yet it is able to finds solutions to very large problems in a very small number of steps. For instance, on the large blocks world problems discussed in [KS96],[3] LPSP is able to find the optimal solution between 2 to 10 times faster than SATPLAN using roughly 100 steps.

This research is part of a more extensive research program aimed at understanding stochastic planning and stochastic local search. We were in part motivated by the question of whether the use of a detailed, low-level propositional representation of planning problems is an essential aspect of stochastic-search based planning, and specifically the success of SATPLAN. LPSP shows that by returning to more traditional search spaces, we can benefit from many of the intuitions developed in past planning research. During this ongoing quest, we have come to appreciate the error in the illusory view that SATPLAN is devoid of such insights. For example, if one examines the detailed interactions of linear encodings of a planning problem [KMS96] with the Walksat search mechanism [SKC94], we can see that SATPLAN has a bias toward (implicitly) extending valid (or near valid) actions sequences. In other words, the encoding provides certain information that can be exploited in a way that *appears*

[1] Depending on the choice of encoding, these propositional models may encode plan steps, intermediate states, or both.

[2] Of course, it has not been claimed that stochastic planning outperforms more traditional approaches on all domains, and an understanding of the nature of domains on which it is better/worse remains an issue.

[3] That is, bw-large.d: a 19 block problem (18 step optimal plan); and bw-large.c: a 15 block problem (14 step optimal plan).

to embody certain planning intuitions; but this interaction is often accidental, and can be hard to verify. Furthermore, it can be almost impossible to apply new planning intuitions to search guidance: one must devise a new encoding whose interaction with stochastic search embodies these intuitions.

The approach taken in LPSP is to provide a representation that allows these explicit planning intuitions and operators to impact search. In fact, as we describe in the concluding section, LPSP is a specific instance of a more general approach that admits different plan representations and search procedures. Our LPSP algorithm is quite simple; it operates roughly as follows. We generate an initial linear plan (sequence of actions of fixed size) from which search proceeds. We also assume the existence of a scoring function that measures the degree of *conflict* exhibited by a plan: plans with a higher score are "less valid" than plans with a lower score. We then compute plan successors by considering the replacement of each action with a different action. We then apply the best replacement (i.e., choose the successor with the minimal score), unless no replacement improves the score of the current plan. In this case, we do one of two things: if the current plan score is sufficiently high (i.e., it has a large number of conflicts), we perform a stochastic shuffling of the plans actions; if the score is low (i.e., it is promising) we perform an optimization step involving the (stochastic) application of various plan construction techniques with limited forward search. This optimization step differentiates LPSP from SATPLAN to great degree, and relies on the choice of plan space as our search space. It is also crucial to the success of LPSP—without it the planner's performance is abysmal.

The rest of this paper is organized as follows. Following a brief review of the classical planning problem, we present a more detailed description of the LPSP. This is followed by a description of our experimental results comparing LPSP to SATPLAN, and a short discussion.

2 The Planning Problem

We focus on the well-known classical AI planning problem. We assume we are given an initial state I, a set of goal states G, and a set of actions A (i.e., partial functions mapping states to states). Our task is to find a sequence of actions $\langle a_1, \cdots, a_n \rangle$ such that the sequential application of these action in this order starting at I will yield a state in G (i.e., $a_n(a_{n-1}(\cdots(a_1(I)\cdots))) \in G$). The complexity of this planning problem depends on the language used for describing I, G and A [ENS95].

We adopt the popular STRIPS language for representing actions [FN71]. We consider problems formulated using a propositional STRIPS representation, where the states correspond to propositional assignments, goal states are described via conjunctions of propositions, and actions are represented using two lists: the *precondition* list, containing a conjunction of propositions, and the effect list, containing a conjunction of literals.[4] An instance of the *move* action from the blocks world domain is shown in Figure 1. The action can be

[4] We assume a basic familiarity with these ideas. For a well written introduction, see [Wel94].

```
MOVE(A,B,C) --
 preconditions: ON(A,B) & CLEAR(A) & CLEAR(C)
 effects:       ON(A,C) & CLEAR(B) & -CLEAR(C)
                & -ON(A,B)
```

Figure 1: An instance of the MOVE action

applied in all states in which the list of preconditions is satisfied, and its result is obtained by adding to the current state description all positive literals in the *effect* list and negating all those propositions that appear as negative literals.

The basic structure used in this paper is a *linear plan*. A linear plan is simply a sequence of actions $\langle a_1, \cdots, a_n \rangle$. Such a plan is called *valid* with respect to initial state I, if for each $i \leq n$, the action a_i can be applied at state $a_{i-1}(a_{i-2}(\cdots a_1(I)\cdots))$; that is, a_i's preconditions hold in this state. A valid plan is a *solution* (w.r.t. I, G) if the application of each action results in the some state in G. Following [MR91], we introduce two artificial actions: an action a_0 that must be executed first in any valid plan, whose effect is to produce the initial state I; and action a_∞ that must occur last in any valid plan, and whose preconditions are the goal conditions. This ensures that finding a valid plan automatically produces a successful plan (by removing a_0 and a_∞).

3 The LPSP Algorithm

The LPSP algorithm searches through the space of linear plans for a solution. We describe the algorithm assuming that plans of a fixed length n are being searched. We deal with arbitrary plans using techniques similar to those described in [BF95, KS96]. Its basic structure is as follows.

```
Repeat until a solution has been found or a maximum
  number of iterations have been tried
  1. Initialize current plan $P$
  2. Repeat until a solution has been found or a maximum
     number of search steps have been tried
     (a) Let $S = s(P)$ be the current plan score
     (b) Call ChooseAction$(P, S)$
     (c) Let $a_k$ be action chosen for replacement and $S_{new} =$
         $s(P')$ where $P'$ is obtained by this replacement in $P$
     (d) If $S_{new} < S$, let current plan be $P'$;
     (e) Else if $S < \delta$, call Optimize$(P, S)$ and let current plan
         be resulting plan;
     (f) Else call Shuffle$(P, S)$ and let current plan be resulting
         plan.
```

The algorithm assumes that a method for generating initial plans has been given and that a scoring function s measuring the "degree of conflict" in a plan has been provided. It uses a procedure *ChooseAction*(P, S) that greedily selects an action a_k in plan P to be replaced by some new action a'_k. This selection is based on the improvements in score offered by candidate replacements. If the selected replacement actually results in an improvement, we update the current plan and proceed with the search. Otherwise, replacement offers no improvement and we consider two alternatives. If the current plan score is below some *optimization threshold* δ, we apply an optimization procedure to P; intuitively, if P is reasonably good, we will perform some plan-directed search. If the threshold is exceeded, we apply a random *Shuffle* procedure. We describe each of these components in turn.

3.1 Initialization and Scoring

The search procedure is restarted with a new plan after a maximum number of search steps and search is stopped after a maximum number of tries. This general scheme can be found in many stochastic local search algorithms, such as GSAT [SLM92] or Walksat [SKC94] A plan is initialized at the beginning of a search try using bi-directional search. If the plan length is n, we choose the last $n/2$ actions by performing regression from the goal state. If multiple actions can be applied, one is chosen randomly. We choose the first $n/2$ actions using an analogous forward search through state space (again randomly choosing from among multiple applicable actions). Thus the initial plan consists of two "valid" fragments that are (highly) unlikely to match where they meet.

The scoring function s is defined as follows. For each action in the plan, its *required atoms* are those ground atoms that appear in some literal in its precondition list (i.e., these are preconditions without polarity). For any action a_k in P and required atom q, the *most recent action* for $\langle a_k, q\rangle$ is the latest occurring action a_j in P, $j < k$, that has an effect on q.[5] If the effect of a_j agrees with this precondition of a_k (i.e., if a_j produces q and a_k needs q, or if a_j produces $\neg q$ and a_k needs $\neg q$), we let $\sigma(a_k, q) = 0$. Intuitively, this means there is no conflict in the plan with regard to a_k's precondition involving q. Otherwise, there is a conflict in the plan in this regard, and we let $\sigma(a_k, q) = (k-j)^2 + (n-k)^2$. The score $s(P)$ is given by the sum of the scores $\sigma(a_k, q)$ for each a_k in P and each of a_k's required atoms q.

Although we arrived at this scoring function empirically, we believe that its main effect is to favor resolution of a conflict between action a_j's effects and a_k's preconditions by *insertion* of an action between a_j and a_k, if possible at position $j+(k-j)/2$): this leads to the greatest reduction in the score of the plan (in this dimension).[6]

3.2 Selecting Actions for Replacement

Having described the scoring function and the plan initialization step, we now describe the main subroutines of LPSP. As mentioned above, given a current plan with score S, we first attempt to replace one of its actions to improve its score. This is the function of the *ChooseAction* procedure which is detailed in Figure 3.2. Intuitively, we calculate for each plan step a_k an action a'_k such that substituting a'_k for a_k in P yields the plan P' with the lowest score among all possible replacements of a_k (let this score be s_k). If there are multiple candidates for a'_k (i.e., multiple actions with the same lowest score), one is chosen at random. Next, all minimal scores s_i, $0 < i \leq n$, are compared. Let $i_{\min}$ be the index of the action that has the smallest value $s_{i_{\min}}$. One possible strategy

```
Procedure choose_action()
  -Let s = score of current plan
  -Choose a random permutation tau over 1,...,n
   (where n is plan size)
  -enough = FALSE
  -Repeat for all plan steps in the order
   determined by tau or until enough = TRUE
    -Let t be the current step; for every
     possible action calculate the score
     obtained by replacing the current action
     in the step t by it
    -Let a' be the action that minimizes this
     score, and let s' be its score
    -If s' < s then
       -with probability 0.8, enough = TRUE
       -s_new = s'
       -act = a'
       -k = t
  -return s_new, act, k;
```

Figure 2: Choosing Actions for Replacement

for selecting the next plan is to simply replace action $a_{i_{\min}}$ by $a'_{i_{\min}}$. Such a greedy replacement step is reminiscent of the means-ends analysis underlying GPS[NS63] and the STRIPS planning algorithm [FN71]. In GPS, for instance, steps are added to a plan in order to reduce the difference between current plan steps. In LPSP, the score can be viewed as quantifying the degree of conflict between plan steps, and step replacement is used to reduce this conflict level.[7]

As described above, the search has only a small stochastic component. As shown in Figure 3.2, we actually use a slightly more stochastic hill-climbing approach: rather than pursuing the steepest descent, a less greedy stochastic choice is made. In particular, a permutation τ of $[1, \cdots, n]$ is chosen, and we calculate $s_{\tau(1)}, s_{\tau(2)}, \ldots$ until we find i such that $s_{\tau(i)}$ is smaller than the current score s. At this point, we stop examining the remaining steps with probability 0.8. This reduces the computational costs of certain steps and allows for additional stochasticity in the replacement choice.

3.3 Optimization and Shuffling

Actions that reduce the conflict level are not always available. When $s_{i_{\min}} \geq s$, rather than replace an action we attempt to revise the current plan P in a way that reorders the actions. If s is below some threshold $\delta(n)$ (i.e., if P is relatively conflict free), we perform an *optimization* procedure by applying various plan construction operations to the actions in P. Otherwise, we perform a randomization step—in particular, we randomly choose a (random) number of action pairs and exchange their place.[8]

The shuffling stage is important one for LPSP, adding an important stochastic element to the planner that helps it escape from local minima. The optimization step, however, is the most crucial step in LPSP. A variant of LPSP without this

[5] If no "true" action has an effect on q, then action a_0 will.

[6] The second term $(n-k)^2$ has only minor effect on the planner's performance.

[7] A natural extension would be to permit the addition, as well as replacement, of plan steps that reduce conflict level.

[8] Procedure *Shuffle* simply chooses a random number $1 \leq k \leq n$, chooses k random pairs of integers $\langle i, j\rangle$ and exchanges action a_i and a_j in P for each such pair.

```
Procedure optimize()
 -For i=1 to n-1
   -For j= i+1 to n
     -if i depends on j then
        exchange steps i and j
 -For i=1 to n-1
   -For j= i+1 to n
     -if i threatens j then
        exchange steps i and j
 -If new_score not better than old_score then
   -let S be the initial state
   -make all actions in plan unchosen
   -while possible
     -choose an unchosen action whose
     -preconditions are satisfied at S and
      mark it chosen
     -Reassign to S the state obtained by
      applying the chosen action to S
```

Figure 3: The Optimization Procedure

optimization step performs very poorly. Without optimization, the LPSP variant is often able to generate plans containing many or all of the steps that appear in some valid solution. Unfortunately, their order is usually incorrect. Since the random shuffling of actions is highly unlikely to stumble upon the correct ordering, and because the cost of each search step (especially action replacement) is considerably higher than the cost of one assignment step of typical stochastic SAT engines, we cannot afford the luxury of waiting for random shuffling of actions to bring about the correct ordering.

The optimization procedure is detailed in Figure 3. It is a based on the heuristic application of some simple intuitions regarding ordering constraints. Intuitively, we attempt to identify incorrectly ordered, but dependent, actions in the current plan and fix the ordering. We proceed in two stages.

We say that action a_k *depends on* action a_l if $k < l$, a_l has an effect that is a precondition of a_k, and no action prior to a_k has this effect. We say that a_k *threatens* a_l if $k < l$ and a_k has an effect that negates some precondition of a_l, and no action a_j ($k < j < l$) has this precondition as an effect. Intuitively, if a_k depends on a_l, swapping their position in the plan has the potential to satisfy the unmet precondition of a_k; and if a_k threatens a_l, swapping has the potential to remove this threat and satisfy this precondition of a_l. Notice that these steps do not completely propagate ordering constraints as might be found in a partial order planner. The reasoning used is “incomplete” but very efficient.[9]

The first stage of optimization examines each action in P in turn, determining whether it depends on some following action; if so, the actions are exchanged in P. Next, we again examine each action to see whether it is threatened by some previous action, and if so, we reverse their ordering. Although this reordering is incomplete (i.e., it does not always generate a correct ordering of the existing actions), it is successful with surprising frequency.

[9] We have considered the use of more complete reasoning about plan constraints in LPSP. We plan to implement such a mechanism in the near future, but suspect that the large overhead with these more complicated search steps may prove detrimental.

The second stage of optimization takes place if the first fails to yield a solution. This reordering phase is based on forward (state-space) search. From the initial state I, we choose an action from P that can be applied at I and add it to a new plan P'. We then repeatedly choose actions from the (remainder of) current plan that can be added to the end of the new plan P' (i.e., applied validly). This continues until no applicable actions can be selected from those left in P. The remaining actions are appended in random order to P' to obtain the new plan. We note that when multiple actions are applicable at a particular stage of P', the action used is selected randomly.[10]

4 Experimental Results

The current version of LPSP is implemented in C++. Due to its early stage of development, a general interface able to read any domain description is not yet available. Rather, the action choice procedures have been hand-coded for each domain. However, this has been done without adding domain dependent information. The main implication of this is that the overhead of compiling a domain description into the form used by the planner is saved. However, we anticipate this stage to be less costly than the plan encoding and the unit resolution stage of SATPLAN, and to take time insignificant compared to planning time.

Experiments were performed on a Sun Ultra 2 workstation with a 200MHz processor and 256 MB RAM. We compared LPSP and SATPLAN on the large blocks world problem instances that are described in [KS96], where SATPLAN using stochastic local search (Walksat) and a linear problem encoding was shown to outperform both GRAPHPLAN and alternate versions of SATPLAN itself. The problems are: `bw_large.a`, which involves 9 blocks and can be optimally solved using 6 steps; `bw_large.b` (11 blocks, 9 step plan); `bw_large.c` (15 blocks, 14 step plan); and `bw_large.d` (19 blocks, 18 step plan).

To ensure a fair comparison, we repeated the SATPLAN experiments on our machine, using the parameter settings found in the public SATPLAN distribution. Because local search steps in LPSP and SATPLAN/Walksat are difficult to compare, we compared the planners using CPU times.[11] For both LPSP and Walksat we ran 100 tries on each problem instance. As in [KS96], the run-times for SATPLAN do not include the time required for transforming the planning problem into a propositional theory and for decoding the solution from SAT into the planning domain.

The results appear in Table 1. The parameters used to obtain these results are shown in Table 2. As can be seen, LPSP is substantially better on the large blocks world problems, but marginally slower on the smaller problems. This is due to the

[10] This stage is implemented very simply by choosing a random permutation of the actions in P, picking applicable actions in the order they occur in this permutation, and then swapping them into the correct position in the plan being generated.

[11] Even for the largest instance, LPSP always finds a solution in less than 1000 steps, while SATPLAN requires approximately ten million steps. But SATPLAN/Walksat performs about 30,000 steps/sec, while LPSP steps might take more than a second each.

	SATPLAN/Wsat		LPSP		
Problem	mean	stddev	mean	median	stddev
bw_large.a	0.45	0.43	2.01	1.45	1.74
bw_large.b	19.14	21.76	26.21	22.74	20.73
bw_large.c	513.65	503.45	72.99	43.73	69.36
bw_large.d	684.59	588.58	322.13	199.73	353.10

Table 1: Experimental results: Comparing SATPLAN and LPSP on hard blocks world planning instances. All data are CPU times in seconds.

	SATPLAN/Wsat		LPSP	
Problem	cutoff	noise	MaxSteps	Opt.Thr. δ
bw_large.a	100k	0.5	1000	-400
bw_large.b	100k	0.35	1000	-1100
bw_large.c	3000k	0.2	1000	-1100
bw_large.d	6000k	0.2	1000	-2100

Table 2: Parameter settings for SATPLAN and LPSP

much greater cost of each plan transformation step. Hence, despite the fact that only a few steps are required for finding a solution, the overall time is greater than that spent by SATPLAN. However, on the larger problems, the reduction in the number of steps required is well worth the extra cost.

We also point out that SATPLAN uses a highly optimized implementation of the underlying local search algorithm Walksat, while our LPSP implementation is comparatively crude. In addition, we have yet to expend significant effort to optimize the parameters used by LPSP. For SATPLAN/Walksat, it is known that its performance critically depends on the settings for the cutoff and noise parameters. Thus, we expect that LPSP can be improved considerably.

The large standard deviations which can be observed in the running times of both algorithms on specific problems indicate a very large variability in the run-time behavior of these stochastic local search algorithms. To study this in more detail, we plotted the cumulative run-time distributions (rtds) for LPSP on each of the blocks world instances in Figure 4. As can be clearly seen from the plots, the shapes of the rtds

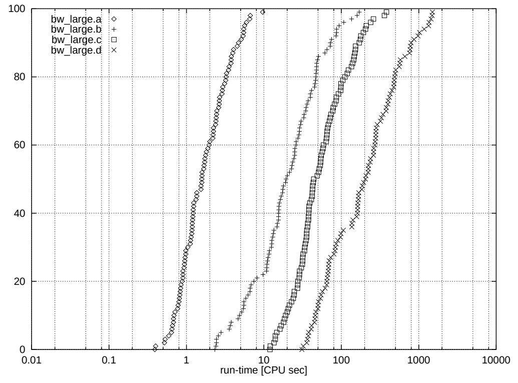

Figure 4: Cumulative Run-time Distributions

are similar for all four instances. Intuitively, starting from an "instance specific" minimal required time to solve a given problem, the probability of finding a plan scales roughly linearly with exponentially increasing run-time; that is, to increase the probability of solving the problem from p to $p+k$, the run-time has to be multiplied by a constant factor. This holds up to a certain maximal run-time after which the problem is almost certainly solved. Thus, despite the inherent incompleteness of LPSP, it solves the blocks world instances with a probability of almost one as the run-time approaches a certain maximal value. Preliminary experiments indicate that similar observations hold for SATPLAN/Walksat.

As can be easily verified, the shape of LPSP's rtds also allows the efficient use of a very simple parallelization strategy: using independent tries on several processors, a linear speedup can be achieved. Note that this form of parallelization is very easy to implement because there is almost no interdependence between the parallel processes.

We also performed some experiments on the logistics domain detailed in [KS96]. Here, SATPLAN substantially outperforms LPSP. In fact, LPSP is currently unable to solve problems that SATPLAN disposes of in roughly two seconds. The difference can be explained by considering the representation used by SATPLAN for these problems. The SATPLAN results for the logistics domain were obtained using a state-based encoding that allows one to consider the concurrent execution of non-interacting actions. In contrast, the blocks world results were obtained using a linear encoding (see [KMS96] on this distinction). As such, the SATPLAN results should be viewed as those of two different planners. The length of the optimal (linear) plans in the logistics domain are at least 47 steps, putting them beyond the reach of LPSP. Using a state-based encoding, SATPLAN can solve such problems because the optimal plan length when concurrent actions are allowed is only 13 steps. As we discuss below, we do not consider these results discouraging.

5 Discussion

We have presented LPSP, the first implemented planning algorithm based on stochastic local search in the space of plans. Our initial experimental results indicate that on certain types of problems, LPSP scales up much better than SATPLAN and therefore, other previous planners. On other domains, it is hindered by its use of linear plan structures.

The main lesson we draw from our initial experience with LPSP is that there is great potential for planners that use stochastic local search techniques in the space of *plans*. There are two main reasons for this conclusion:

1. LPSP scales up better than SATPLAN as a function of the plan length on blocks world problems. While SATPLAN exploits well-optimized SAT engines, we haven't yet had the opportunity to optimizing LPSP's parameters. Moreover, SATPLAN results are generally obtained using different random-walk probabilities for different problem instances.[12] In contrast, the only problem specific param-

[12]We base this observation on the material distributed with the SATPLAN planner.

eter used in LPSP is the score threshold $\delta(n)$ used to direct plan optimization; this is due to the fact that average score is highly dependent on plan length.

2. The use of an intuitive plan representation immediately suggests the possibility of using of many novel concepts (such as various plan representations, measures of plan quality and plan transformations), developed in the classical planning community, by stochastic search algorithms.

This last point is especially important. The dismal performance of LPSP on the logistics domain may suggest dim prospects for LPSP. But we believe that the use of more sophisticated plan representations and search spaces, especially those based on non-linear plans, constraint-based planning representations, and those that allow concurrent action such as GRAPHPLAN [BF95], offer great promise. Indeed, the success of SATPLAN using a state-based encoding bodes well for the extension of LPSP in that fashion.

Stochastic local search techniques for solving satisfiability problems have started to gain wide attention in the AI community and, as a result, considerable advances in the performance of these methods have been achieved. The LPSP algorithm is still in its earliest stages of development. It is our hope that similar improvements will be made in plan-level stochastic local search techniques.

There are a number of optimizations that we hope to examine in the near future, both with respect to the implementation and the underlying algorithm. For instance, we hope to soon investigate the use of non-linear and least commitment plan representations, and more sophisticated ordering techniques, as discussed above. Another idea worth pursuing is direct search in the space of variable-sized plans. This could fit well with LPSP's optimization steps, where actions that do not exist in the current plan could be added if needed, or where existing actions could be deleted if not useful. In addition, we envision many possible avenues of development. For example, one could combine ideas from SATPLAN and LPSP by, say, integrating LPSP's optimization methods with SATPLAN's ability to reason with constraints; or by using SATPLAN for the initial search phase of LPSP. It is our hope that additional ideas from more traditional planning algorithms will be combined with stochastic local search techniques to yield improved planners.

References

[BF95] A. Blum and M. L. Furst. Fast planning through planning graph analysis. In *Proc. Fourteenth International Joint Conference on AI*, 1995.

[Cha87] D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[ENS95] K. Erol, D. Nau, and V. Subrahmanian. Complexity, decidability, and undecidability results for domain independent planning. *Artificial Intelligence*, 76(1-2):76–88, 1995.

[FN71] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[Gin96] M. Ginsberg. A new algorithm for generative planning. In *Proc. of the 5th Intl. Conf. on Principles of Knowledge Representation*. 1996.

[JP96] D. Joslin and M. E. Pollack. Is "'early commitment" in plan generation ever a good idea? In *Proc. of the 13th National Conf. on AI (AAAI '96)*, pages 1188–1193, 1996.

[KKY95] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, 88(1–2):253–315, 1995.

[KMS96] H. Kautz, D. McAllester and B. Selman. Encoding plans in propositional logic. In *KR'96*, 374–384, 1996.

[KS96] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of the 13th National Conference on AI (AAAI '96)*, pages 1194–1201, 1996.

[MR91] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. of the 9th National Conf. on AI (AAAI '91)*, pages 634–639, 1991.

[NS63] A. Newell and H. A. Simon. GPS, a program that simulates human thought. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill, 1963.

[PW92] J. S. Penberthy and D. S. Weld. Ucpop: A sound, complete, partial order planner for adl. In *Principles of Knowledge Representation and Reasoning: Proc. Third Intl. Conf. (KR '92)*, 1992.

[SKC94] B. Selman, H. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *AAAI'94*, 337–343, MIT press, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *AAAI'92*, 440–446, MIT press, 1992.

[Wel94] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, Winter 1994:27–61, 1994.