From: AAAI-99 Proceedings. Copyright © 1999, AAAI (www.aaai.org). All rights reserved.

Fast Planning through Greedy Action Graphs *

Alfonso Gerevini and Ivan Serina
Dipartimento di Elettronica per l'Automazione
Universitá di Brescia, via Branze 38, 25123 Brescia, Italy
{gerevini,serina}@ing.unibs.it

Abstract

Domain-independent planning is a notoriously hard search problem. Several systematic search techniques have been proposed in the context of various formalisms. However, despite their theoretical completeness, in practice these algorithms are incomplete because for many problems the search space is too large to be (even partially) explored.
In this paper we propose a new search method in the context of Blum and Furst's planning graph approach, which is based on local search. Local search techniques are incomplete, but in practice they can efficiently solve problems that are unsolvable for current systematic search methods. We introduce three heuristics to guide the local search (Walkplan, Tabuplan and T-Walkplan), and we propose two methods for combining local and systematic search.
Our techniques are implemented in a system called GPG, which can be used for both plan-generation and plan-adaptation tasks. Experimental results show that GPG can efficiently solve problems that are very hard for current planners based on planning graphs.

Introduction

Domain-independent planning is a notoriously very hard search problem. The large majority of the search control techniques that have been proposed in the recent literature rely on a systematic method that in principle can examine the complete search space. However, despite their theoretical completeness, in practice these search algorithms are incomplete because for many planning problems the search space is too large to be (even partially) explored, and a plan cannot be found in reasonable time (if one exists).

Here we are concerned with an alternative search method which is based on a local search scheme. This method is formally incomplete, but in practice it can efficiently solve problems that are very hard to solve by more traditional systematic methods.[1]

Though local search techniques have been applied with success to many combinatorial problems, they have only recently been applied to planning (Ambite & Knoblock 1997; Kautz & Selman 1998; 1996; Serina & Gerevini 1998). In particular, Kautz and Selman experimented the use of a stochastic local search algorithm (Walksat) in the context of their "planning as satisfiability" framework, showing that Walksat outperforms more traditional systematic methods on several problems (Kautz & Selman 1996).

In the first part of this paper we propose a new method for local search in the context of the "planning through planning graph analysis" approach (Blum & Furst 1995). We formulate the problem of generating a plan as a search problem, where the elements of the search space are particular subgraphs of the planning graph representing partial plans. The operators for moving from one search state to the next one are particular graph modification operations, corresponding to adding (deleting) some actions to (from) the current partial plan. The general search scheme is based on an iterative improvement process, which, starting from an initial subgraph of the planning graph, greedily improves the "quality" of the current plan according to some evaluation functions. Such functions measure the cost of the graph modifications that are possible at any step of the search. A final state of the search process is any subgraph representing a valid complete plan.

We introduce three heuristics: *Walkplan, Tabuplan,* and *T-Walkplan.* The first is inspired by the stochastic local search techniques used by Walksat (Selman, Kautz & Cohen 1994), while the second and the third are based on different ways of using a tabu list storing the most recent graph modifications performed.

In the second part of the paper we propose two methods for combining local and systematic search for planning graphs. The first is similar to the method used in Kautz and Selman's Blackbox for combining different search algorithms, except that in our case we use the same representation of the problem, while Blackbox uses completely different representations. The second method is based on the following idea. We use local search for efficiently producing a plan which is *almost* a solution, i.e., that possibly contains only a few flaws

Copyright ©1999, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

[1] Local search methods are incomplete in the sense that they cannot detect that a search problem has *no* solution.

(unsatisfied preconditions or exclusion relations involving actions in the plan), and then we use a particular systematic search to "repair" such a plan and produce a valid solution.

Our techniques are implemented in a system called GPG (Greedy Planning Graph). Experimental results show that GPG can efficiently solve problems that are hard for IPP (Koehler *et al.* 1997), Graphplan (Blum & Furst 1995) and Blackbox (Kautz & Selman 1999).

Although this paper focuses on plan-generation, we also give some preliminary experimental results showing that our approach can be very efficient for solving plan-adaptation tasks as well. Fast plan-adaptation is important, for example, during plan execution, when the failure of some planned action, or the acquisition of new information affecting the world description or the goals of the plan, can make the current plan invalid.

In the rest of the paper, we first briefly introduce the planning graph approach; then we present our local search techniques and the methods for combining systematic and local search; finally, we present experimental results and give our conclusions.

Planning Graphs

A planning graph is a directed acyclic levelled graph with two kinds of nodes and three kinds of edges. The levels alternate between a fact level, containing fact nodes, and an action level containing action nodes. A fact node represents a proposition corresponding to a precondition of one or more operators instantiated at time step t (actions at time step t), or to an effect of one or more actions at time step $t-1$. The fact nodes of level 0 represents the positive facts of the initial state of the planning problem.[2] The last level is a proposition level containing the fact nodes corresponding to the goals of the planning problem.

In the following we indicate with $[u]$ the proposition (action) represented by the fact node (action node) u. The edges in a planning graph connect action nodes and fact nodes. In particular, an action node a of level i is connected by:

- *precondition edges* to the fact nodes of level i representing the preconditions of $[a]$,
- *add-edges* to the fact nodes of level $i+1$ representing the positive effects of $[a]$,
- *delete-edges* to the fact nodes of level $i+1$ representing the negative effects of $[a]$.

Two action nodes of a certain level are *mutually exclusive* if no valid plan can contain both the corresponding actions. Similarly, two fact nodes are mutually exclusive if no valid plan can make both the corresponding propositions true.

Two proposition nodes p and q in a proposition level are marked as exclusive if each action node a having

[2]Planning graphs adopt the closed world assumption.

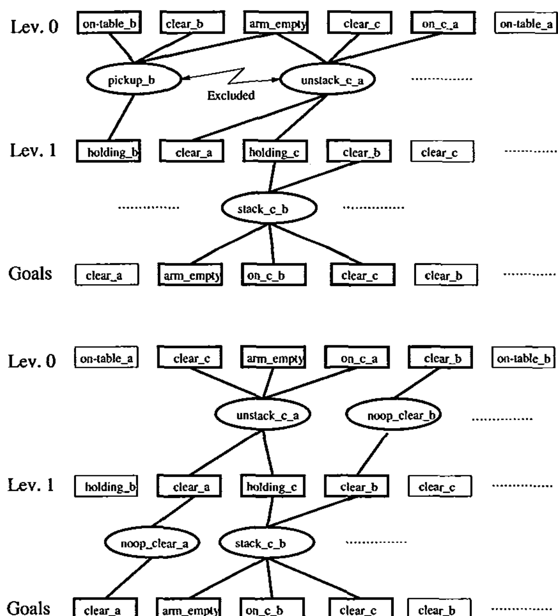

Figure 1: An action subgraph (bold nodes and edges) and a solution subgraph of a planning graph. The problem goals are `clear_a`, `arm_empty`, `on_c_b` and `clear_c`.

an add-edge to p is marked as exclusive of each action node b having an add-edge to q. In the last level of a planning graph there is no pair of mutually exclusive nodes representing goals.

An action node a of level i can be in a "valid subgraph" of the planning graph (a subgraph representing a valid plan) only if all its precondition nodes are *supported*, and a is not involved in any mutual exclusion relation with other action nodes of the subgraph. We say that a fact node q of level i representing a proposition $[q]$ is supported in a subgraph $\mathcal{G}'$ of a planning graph $\mathcal{G}$ if either (a) in $\mathcal{G}'$ there is an action node at level $i-1$ representing an action with (positive) effect $[q]$, or (b) $i = 0$ (i.e., $[q]$ is in the initial state).

Given a planning problem $\mathcal{P}$ and a planning graph $\mathcal{G}$, a *solution* (plan) for $\mathcal{P}$ is a subgraph $\mathcal{G}'$ of $\mathcal{G}$ such that (1) all the precondition nodes of actions in $\mathcal{G}'$ are supported, (2) every goal node is supported, and (3) there are no mutual exclusion relations between action nodes of $\mathcal{G}'$.

Local Search for Planning Graphs

Our local search method for a planning graph $\mathcal{G}$ of a given problem $\mathcal{P}$ is a process that, starting from an initial subgraph $\mathcal{G}'$ of $\mathcal{G}$ (a partial plan for $\mathcal{P}$), transforms $\mathcal{G}'$ into a solution of $\mathcal{P}$ through the iterative application of some graph modifications that greedily improve the "quality" of the current partial plan. Each modification is either an extension of the subgraph to include

a new action node of $\mathcal{G}$, or a reduction of the subgraph to remove an action node (and the relevant edges).

Adding an action node to the subgraph corresponds to adding an action to the partial plan represented by the subgraph (analogously for removing an action node). At any step of the search process the set of actions that can be added or removed is determined by the *constraint violations* that are present in the current subgraph of $\mathcal{G}$. Such violations correspond to

- mutual exclusion relations involving action nodes in the current subgraph;
- unsupported facts, which are either preconditions of actions in the current partial plan, or goal nodes in the last level of the graph.

More precisely, the search space is formed by the *action subgraphs* of the planning graph $\mathcal{G}$, where an action subgraph of $\mathcal{G}$ is defined in the following way:

Definition 1 *An* **action subgraph** $\mathcal{A}$ *of a planning graph* $\mathcal{G}$ *is a subgraph of* $\mathcal{G}$ *such that if* a *is an action node of* $\mathcal{G}$ *in* $\mathcal{A}$*, then the fact nodes of* $\mathcal{G}$ *corresponding to the preconditions and positive effects of* [a] *are also in* $\mathcal{A}$*, together with the edges of* $\mathcal{G}$ *connecting them to* a.

A *solution subgraph* (a final state of the search space) is defined in the following way:

Definition 2 *A* **solution subgraph** *of a planning graph* $\mathcal{G}$ *is an action subgraph* $\mathcal{A}_s$ *containing the goal nodes of* $\mathcal{G}$ *and such that*

- *all the goal nodes and fact nodes corresponding to preconditions of actions in* $\mathcal{A}_s$ *are supported;*
- *there is no mutual exclusion relation between action nodes.*

The first part of Figure 1 shows a simple example of an action subgraph $\mathcal{A}$. The actions `pickup_b` and `unstack_c_a` of level 0 are mutually exclusive, and therefore they can not be both present in any solution subgraph. Note that the goal node `clear_a` is not supported in $\mathcal{A}$ and does not belong to $\mathcal{A}$, even though it must belong to all solution subgraphs. The second part of Figure 1 gives a solution subgraph.

Our general scheme for searching a solution graph consists of two main steps. The first step is an initialization of the search in which we construct an initial action subgraph. The second step is a local search process in the space of all the action subgraphs, starting from the initial action subgraph. In the context of local search for CSP, the initialization phase is an important step which can significantly affect the performance of the search phase (Minton *et al.* 1992). In our context we can generate an initial action subgraph in several ways. Two possibilities that we have considered in our experiments are: (1) a randomly generated action-subgraph; (2) an action-subgraph where all precondition facts and goal facts are supported (but in which there may be some violated mutual exclusion relations). These kinds of initialization can be performed in linear time in the size of the graph $\mathcal{G}$.

The search phase is performed in the following way. A constraint violation in the current action subgraph is randomly chosen. If it is an unsupported fact node, then in order to eliminate this constraint violation, we can either add an action node that supports it, or we can remove an action node which is connected to that fact node by a precondition edge. If the constraint chosen is an exclusion relation, then we can remove one of the action nodes of the exclusion relation. Note that the elimination of an action node can remove several constraint violations (i.e., all those corresponding to the set of exclusion relations involving the action node eliminated). On the other hand, the addition of an action node can introduce several new constraint violations. Also, when we add (remove) an action node to satisfy a constraint, we also add (remove) all the edges connecting the action node with the corresponding precondition and effect nodes in the planning graph – this ensures that each change to the current action subgraph is another action subgraph.

The decision of how to deal with a constraint violation can be guided by a *general objective function*, which is defined in the following way:

Definition 3 *Given the partial plan* π *represented by an action subgraph* $\mathcal{A}$*, the* **general objective function** $f(\pi)$ *of* π *is defined as:*

$$f(\pi) = g(\mathcal{A}) + \sum_{a \in \mathcal{A}} me(a, \mathcal{A}) + p(a, \mathcal{A})$$

where a *is an action in* $\mathcal{A}$*,* $me(a, \mathcal{A})$ *is the number of action nodes in* $\mathcal{A}$ *which are mutually exclusive with* a*,* $p(a, \mathcal{A})$ *is the number of precondition facts of* a *which are not supported, and* $g(\mathcal{A})$ *is the number of goal nodes in* $\mathcal{A}$ *which are not supported.*

It is easy to see that the value of this objective function is zero for any valid plan of a given planning problem. This function can be used in the search process at each search step to discriminate between different possible graph modifications, and to choose one which minimizes the objective function.

Local search heuristics

The use of the general objective function to guide the local search might be effective for some planning problems, but it has the drawback that it can lead to local minima from which the search can not escape. For this reason, instead of using the general objective function we use an *action cost function* F. This function defines the cost of inserting (F^i) and of removing (F^r) an action $[a]$ in the partial plan π represented by the current action subgraph $\mathcal{A}$. $F([a], \pi)$ is defined in the following way:

$$\begin{cases} F([a],\pi)^i = \alpha^i \cdot p(a,\mathcal{A}) + \beta^i \cdot me(a,\mathcal{A}) + \gamma^i \cdot unsup(a,\mathcal{A}) \\ F([a],\pi)^r = \alpha^r \cdot p(a,\mathcal{A}) + \beta^r \cdot me(a,\mathcal{A}) + \gamma^r \cdot sup(a,\mathcal{A}), \end{cases}$$

where $me(a,\mathcal{A})$ and $p(a,\mathcal{A})$ are defined as in Definition 3, $unsup(a,\mathcal{A})$ is the number of unsupported precondition facts in $\mathcal{A}$ that become supported by adding a to $\mathcal{A}$, and $sup(a,\mathcal{A})$ is the number of supported precondition facts in $\mathcal{A}$ that become unsupported by removing a from $\mathcal{A}$.

By appropriately setting the values of the coefficients α, β and γ we can implement various heuristic methods aimed at making the search less susceptible to local minima by being "less committed" to following the gradient of the general objective function. Their values have to satisfy the following constraints:

$$\alpha^i > 0,\ \beta^i > 0,\ \gamma^i \leq 0,\quad \alpha^r \leq 0,\ \beta^r \leq 0,\ \gamma^r > 0.$$

Note that the positive coefficients of F (α^i, β^i and γ^r) determine an increment in F which is related to an increment of the number of constraint violations. Analogously, the non-positive coefficients of F (α^r, β^r and γ^i) determine a decrement in F which is related to a decrement of the number of constraint violations.

In the following we describe three simple search heuristics, and in the last part of the paper we will present some preliminary experimental results obtained by using these heuristics. The general search procedure at each step randomly picks a constraint violation s and considers the costs of the action deletions/insertions which resolve s. The action subgraphs that can be obtained by performing the modifications corresponding to such action deletions/insertions constitute the neighborhood $N(s,\mathcal{A})$ of s, where $\mathcal{A}$ is the current action subgraph. The following heuristics can be used to choose the next action subgraph among those in $N(s,\mathcal{A})$.

Walkplan

Walkplan uses a random walk strategy similar to the strategy used in Walksat (Selman, Kautz & Cohen 1994). Given a constraint violation s, in order to decide which of the action subgraphs in $N(s,\mathcal{A})$ to choose, Walkplan uses a greedy bias that tends to minimize the number of new constraint violations that are introduced by the graph modification. Since this bias can easily lead the algorithm to local minima from which the search cannot escape, it is not always applied.

In particular, if there is a modification that does not introduce new constraint violations, then the corresponding action subgraph in $N(s,\mathcal{A})$ is chosen as the next action subgraph. Otherwise, with probability p one of the subgraphs in $N(s,\mathcal{A})$ is chosen randomly, and with probability $1-p$ the next action subgraph is chosen according to minimum value of the action cost function.

Walkplan can be implemented by setting the α, β and γ coefficients to values satisfying the following constraints: $\alpha^i, \beta^i > 0, \gamma^i = 0$ and $\alpha^r, \beta^r = 0, \gamma^r > 0$.

Tabuplan

Tabuplan uses a tabu list (Glover & Laguna 1993; Glover, Taillard, & de Werra 1993) which is a special short term memory of actions inserted or removed. A simple strategy of using the tabu list that we have tested in our experiments consists of preventing the deletion (insertion) of an action just introduced (removed) for the next k search steps, where k is the length of the tabu list.

At each step of the search, from the current action subgraph Tabuplan chooses as next subgraph the "best" subgraph in $N(s,\mathcal{A})$ which can be generated by adding or removing an action that is *not* in the tabu list. The length k of the tabu list is a parameter that is set at the beginning of the search, but that could also be dynamically modified during the search (Glover & Laguna 1993).

T-Walkplan

This heuristic uses a tabu list simply for *increasing the cost* of certain graph modifications, instead of preventing them as in Tabuplan. More precisely, when we evaluate the cost of adding or removing an action $[a]$ which is in the tabu list, the action cost $F([a],\pi)$ is incremented by $\delta \cdot (k-j)$, where δ is a small quantity (e.g. 0.1), k is the length of the tabu list, and j is the position of $[a]$ in the tabu list.[3]

Combining local and systematic search

As the experimental results presented in the next section show, local search techniques can efficiently solve several problems that are very hard to solve for IPP or Graphplan. On the other hand, as general planning algorithms they have the drawback that they cannot detect when a valid plan does *not* exist in a planning graph with a predefined number of levels. (Hence they cannot determine when the planning graph should be extended.) Furthermore, we observed that some problems that are very easy to solve for the systematic search as implemented in IPP, are harder for our local search techniques (though they are still solvable in a few seconds, and our main interest concerns problems that are very hard for current planners based on planning graphs).

Motivated by these considerations, we have developed a simple method for automatically increasing the size of a planning graph, as well as two methods for

[3] We assume that the tabu list is managed according to a first-in-first-out discipline.

[4] In our current implementation the search from `goal-level` to `init-level` at step 5 is initially restricted by limiting the possible number of levels in the (re)planning graph to 3. This number is automatically increased by 2 each time the replanning window is increased by 1 level.

ADJUST-PLAN

Input: A plan $\mathcal{P}$ containing some flaws and a CPU-time limit `max-adjust-time`.

Output: Either a correct plan or `fail`.

1. Identify the set F of levels in $\mathcal{P}$ containing a flaw; If F is empty, then return $\mathcal{P}$;
2. Let `i` be a level in F and remove `i` from F;
3. If `i` is the last level of $\mathcal{P}$, then set `init-level` to `i` − 1 and `goal-level` to `i`, otherwise set `init-level` to `i` and `goal-level` to `i + 1`;
4. While CPU-time ≤ `max-adjust-time`
   5. Systematically replan using as initial facts F(`init-level`) and as goals G(`goal-level`), where F(`init-level`) is the set of facts that are true at level `init-level`, and G(`goal-level`) is the set of preconditions of the actions in $\mathcal{P}$ at level `goal-level` (including the no-ops);
   6. If there is no plan from F(`init-level`) to G(`goal-level`), or a search limit is exceeded, then decrease `init-level` or increase `goal-level` (i.e., we enlarge the replanning window), otherwise insert the (sub)plan found into $\mathcal{P}$ and goto 1.[4]
7. Return fail.

Figure 2: Description of the algorithm used by GPG for adjusting a plan generated by the local search.

combining our local search techniques with IPP's systematic search. These methods are implemented in a planner called GPG (Greedy Planning Graph).[5]

Like IPP, GPG starts searching when the construction of the planning graph has reached a level in which all the fact goals of the problem are present and are not mutually exclusive. When used in a purely local-search mode, if after a certain number of search steps a solution has not been found, GPG extends the planning graph by adding a level to it, and a new search on the extended graph is performed.[6]

The first method for combining local and systematic search borrows from Kautz and Selman's Blackbox the idea of combining different search algorithms in a serial way. In particular, we alternate systematic and local search in the following way. First we search the planning graph using a systematic method as in Graphplan or IPP until either (a) a solution is found, (b) the problem is proved to be unsolvable, or (c) a predefined CPU time limit is exceeded (this limit can be modified by the user). If the CPU-time has been exceeded, then we activate a local search on the planning graph using our local search techniques, which has the same termination conditions as the systematic search, except that the problem cannot be proved to be unsolvable. If the local search also does not find a solution within a certain CPU-time limit, then we extend the planning graph by adding a new level to it, and we repeat the process.

The second method exploits the fact that often when the local search does not find a solution in a reasonable amount of time, it comes "very close" to it, producing action subgraphs representing plans that are *quasi*-solutions, and that can be adjusted to become a solution by making a limited number of changes to them. A *quasi*-solution is an *almost correct* plan, i.e., a plan $\mathcal{P}$ which contains a few unsatisfied preconditions or exclusion relations involving actions in the plan.

This method consists of two phases. In the first phase we search for a *quasi*-solution of the planning problem using local search (the number of levels in the graph is automatically increased after a certain number of search steps). The second phase identifies the flaws that are present in $\mathcal{P}$, and tries to "repair" them by running ADJUST-PLAN, an algorithm performing systematic search (see Figure 2).

ADJUST-PLAN first identifies the levels of $\mathcal{P}$ which contain a pair of mutually exclusive actions or an action with some unachieved precondition(s). Then it processes these levels in the following way. If level `i` contains a flaw, then it tries to repair it by replanning from time level `i` to level `i + 1` using systematic search. If there exists no plan or a certain search limit is exceeded, then the replanning window is enlarged (e.g., we replan from `i - 1` to `i + 1`).[4] The process is iterated until a (sub)plan is found, or the search has reached a predefined CPU-time limit (`max-adjust-time`). The idea is that if $\mathcal{P}$ contains some flaw that cannot be repaired by limited (systematic) replanning, then ADJUST-PLAN returns fail, and the local search is executed again to provide another plan that may be easier to repair.[7] When a subplan is found, it is appropriately inserted into $\mathcal{P}$.

At step 6 of ADJUST-PLAN the replanning window can be increased going either backward in time (i.e., `init-level` is decreased), forward in time (i.e., `goal-level` is increased), or both.[8] The introduction of any subplan found at step 5 does not invalidate the

[5]GPG is written in C and it uses IPP's data structures. IPP is available at http://www.informatik.uni-freiburg/~koehler/ipp.html.

[6]GPG has a default value for this search limit, which is increased each time the graph is extended. The user can change its value, as well as its increment when the graph is extended.

[7]GPG has a default `max-adjust-time` that can be modified by the user. In principle, if `max-adjust-time` were set to sufficiently high values, then ADJUST-PLAN could increase the replanning window to reach the original initial and goal levels of the planning graph. This would determine a complete systematic search, that however we would like to avoid. Also, note that in our current implementation of ADJUST-PLAN, during replanning the actions of $\mathcal{P}$ that are present in the replanning window are ignored (a new local planning graph is constructed).

[8]Note that when the replanning window is increased by moving the goal state forward, keeping the same initial state, we can use the memoization of unachieved subgoals to prune the search as indicated in (Blum & Furst 1995; Koehler *et al.* 1997).

rest of the plan. On the contrary, such a subplan may be useful for achieving unachieved preconditions that are present at levels later than the level that started the process at step 2. Note also that since we are currently considering STRIPS-like domains, step 1 can be accomplished in polynomial time by doing a simulation of $\mathcal{P}$. Similarly, the facts that are (necessarily) true at any level can be determined in polynomial time.

As the experimental results presented in the next section show, this method often leads to a solution very efficiently. However, such a solution is not guaranteed to be optimal with respect to the number of time steps that are present in the solution. For this reason, when ADJUST-PLAN finds a solution, we attempt to optimize the adjusted plan, trying to produce a more compact plan (however, for lack of space we omit the description of this process).

On the other hand, it should be noticed that the plans obtained by IPP or Graphplan from a planning graph with the minimum number of levels are *not* guaranteed to be optimal in terms of the number of *actions* involved, which is an important feature of the quality of a plan. The experimental results in the next section show that GPG can generate plans with a lower number of actions than the number of actions in plans generated by IPP. Thus, in this sense, in general the quality of the plans obtained with our method is no worse than the quality of the plans obtained with IPP or Graphplan.

Finally, note that the two methods for combining local and systematic search can be merged into a single method, which iteratively performs systematic search, local search and plan-adjustment.

Experimental results

In order to test the effectiveness of local search techniques, we have been conducting three kind of experiments. The first is aimed at testing the efficiency of local search in a planning graph with the (predetermined) minimum number of levels that are sufficient to solve the problem. The second is aimed at testing the combination of local search and plan-adjustment described in the previous section, in which the number of levels in the planning graph is not predetermined. The third concerns the use of our techniques for the task of repairing a precomputed plan in order to cope with some changes in the initial state or in the goal state. Preliminary results from these experiments concern the following domains: Rocket, Logistics, Gripper, Blocks-world, Tyre-world, TSP, Fridge-world and Monkey-world.[9]

Table I gives the results of the first kind of experiments, which was conducted on a Sun Ultra 1 with 128 Mbytes. The CPU-times for the local search techniques are averages over 20 runs. The table includes the CPU-time required by IPP and by Blackbox (using only Walksat for the search) for solving each of the problems considered. Each run of our methods consists of 10 tries, and each try consists of (a) an initialization phase, in which an initial action subgraph is generated; (b) a search phase with a default search limit of 500,000 steps (graph modifications).[10]

The first part of the table concerns some problems that are hard to solve for IPP, but that can be efficiently solved using local search methods. In particular, our techniques were very efficient for Rocket and Logistics, where the local search methods were up to more than four orders of magnitude faster than the systematic search of IPP.

The second half of Table I gives the results for some problems that are easy to solve for the systematic search of IPP. Here our local search techniques also performed relatively well. Some of the problems were solved more efficiently with a local search and some others with a systematic search, but in all cases the search required at most a few seconds.

Compared to Blackbox, in general, in this experiment the performance improvement of our search methods was less significant, except for bw_large_a where they were significantly faster than Walksat.

Table II gives results concerning the second kind of experiment, where GPG uses local search for a fast computation of a *quasi*-solution, which is then processed by ADJUST-PLAN. This experiment was conducted on a Sun Ultra 60 with 512 Mbytes. As local search heuristic in every run we used Walkplan with noise equal to either 0.3, 0.4 or 0.5 (lower noise for harder problems, higher noise for easier problems).[11] In all the tests `max-adjust-time` was set to 180 seconds and the number of flaws admitted in a quasi-solution was limited to either 2, 4 or 6.

Compared to IPP, GPG found a solution very efficiently. In particular, on average GPG required less than a minute (and 6.3 seconds in the fastest run) to solve Logistics-d, a test problem introduced in (Kautz & Selman 1996) containing 10^{16} possible states. More-

[9]The formalization of these domains and of the relative problems is available as part of the IPP package. For the blocks world we used the formalization with four operator: `pick_up`, `put_down`, `stack` and `unstack`.

[10]It should be noted that the results of this experiment were obtained by setting the parameters of our heuristics and of Walksat to particular values, that were empirically chosen as the best over many values tested.

[11]However, we observed that the combination of local search and plan-adjustment in GPG does not seem to be significantly sensitive to the values of the parameters of the heuristics, and we expect that the use of default values for all the runs would give similar results. Probably the major reason of this is that here local search is not used to find a solution, but to find a *quasi*-solution. Further experiments for confirming this observation are in progress.

Problem	graph levels	graph creation	Walkplan	Tabuplan	T-Walkplan	IPP	Blackbox (Walksat)
Rocket_a	7	0.46	48.57 (2)	1.16	0.5	126.67	5.77
Rocket_b	7	0.49	6.5	1.4	2.78	334.51	8.23
Logistics_a	11	1.58	2.5	1.77	1.04	2329.06	4.0
Logistics_b	13	1.15	19.9	85.43 (3)	5.25	1033.75	12.83
Logistics_c	13	2.22	19.4	37.63	7.05	> 24 hours	20.91
Bw_large_a	12	0.3	4.04	123 (8)	13.1	0.38	705 (4)
Blocks-suss	6	0.04	0.01	0.01	0.02	0.01	0.139
Tyre-fixit	12	0.08	0.09	0.2	0.077	0.05	0.273
Rocket-10	3	0.1	0.006	0.003	0.005	0.005	0.083
Monkey-2	8	0.2	5.49	6.9	0.68	0.04	3.882
TSP-complete	9	0.28	1.26	0.99	0.59	1.65	0.833
Fridge-2	6	0.88	0.056	0.063	0.09	0.06	0.243
Logistics-4	9	0.37	0.01	0.007	0.01	0.01	0.041
Logistics-4pp	9	0.34	0.007	0.005	0.008	0.005	0.079
Logistics-6h	11	0.48	0.33	2.01	0.15	2.82	0.156
Logistics-8	9	0.65	0.17	0.18	0.05	0.01	0.147

Table I: CPU seconds required by our local search heuristics, by IPP (v. 3.3) and by Blackbox (v. 3.4) for solving some hard problems (first part of the table) and some easy problems (second part). The numbers into brackets indicate the number of runs in which a solution was not found (if this is not indicated, then a solution was found in all the runs.)

Problem	GPG					IPP			Blackbox	
	total time mean(min)	local time	repair time	levels mean (min)	actions mean (min)	total time	num. levels	num. actions	Graphplan + Satz	Graphplan + Walksat
Bw_large_a	1.6(0.8)	0.35	1.25	13.6 (12)	13.6 (12)	0.31	12	12	1.24	1.24
Bw_large_b	27.3(2.3)	3.5	23.8	20.6 (18)	20.6 (18)	7.75	18	18	109.88	3291
Rocket_a	1.1(0.5)	0.8	0.28	9.2 (7)	33 (30)	53.5	7	34	59.7	60.86
Rocket_b	1.3(0.4)	1.0	0.3	9.4 (7)	32.4 (30)	146.7	7	30	65.64	60.58
Logistics_a	3.9(0.8)	0.36	3.5	13.6 (11)	63.3 (56)	956	11	64	70.34	76.34
Logistics_b	3.9(1.4)	0.9	3.0	15.6 (13)	58 (52)	423	13	45	112.05	302
Logistics_c	2.42(2.15)	1.66	0.76	15.6 (14)	70.8 (69)	> 24 h.	–	–	162.4	539
Logistics_d	57(6.3)	5.6	50.7	20.5 (17)	94.5 (84)	> 24 h.	–	–	132.2	912
Gripper-10	51.2(15.4)	46.8	4.3	26.7 (19)	35.16 (29)	126.47	19	29	2607.8	2467
Gripper-12	320(63.8)	71.4	248.7	29 (29)	38.4 (35)	1368.5	23	35	> 9070	> 8432

Table II: Performance of GPG using Walkplan and ADJUST-PLAN compared to IPP 3.3 and Blackbox 3.4. The 2nd column gives the average (minimum) CPU-seconds required by GPG to find a solution over 5 runs. The 3rd gives the average time spent by the local search, and the 4th the time for adjusting the plan. The 5th column gives the average (minimum) number of levels that were required to find a solution by GPG. The 6th column gives the average (minimum) number of actions in a solution found by GPG. The 7–9th columns give CPU-seconds, number of levels and actions required by IPP. The 10–11th columns give the average CPU-seconds required by Blackbox over 5 runs.

over, in terms of the number of actions, the plans generated by GPG on average are not significantly worse than the plans generated by IPP. On the contrary, in some cases GPG found plans involving a number of actions lower than in the plans found by IPP. For example, for Rocket_a GPG found plans involving a minimum of 30 actions and on average of 33 actions, while the plan generated by IPP contains 34 actions.

Concerning the problems in the blocks world (bw_large-a e bw_large-b), on average GPG did not perform as well as IPP (though these problems are still solvable in seconds). In these problems the local search can take a relatively high amount of time for generating quasi-solutions, which can be expensive to repair. The reasons for this are not completely clear yet, but we believe that they partly depend on the fact that in planning graphs with a limited number of levels these problems have only a few solutions and quasi-solutions (Clark *et al.* 1997).

In general, GPG was significantly faster than Blackbox (last columns of Table II) regardless the SAT-solver that we used (either Satz or Walksat).[12]

Table III shows preliminary results of testing the use of local search for the task of plan adaptation, where in the input we have a valid plan (solution) for a problem P, and a set of changes to the initial state or the goal state of P. The plan for P is used as initial subgraph of the planning graph for the revised problem P', that is greedily modified to derive a new plan for P'.

In particular, the table gives the plan-adaptation and plan-generation times for some variants of Logistics_a, where in general the local search performed much more efficiently than a complete replanning using IPP (up to five orders of magnitude faster).[13]

I_1-5 correspond to five modifications of the problem obtained by changing a fact in the initial state that affects the applicability of some planned action. G_1-11 are modifications corresponding to some (significant) changes to the goals in the last level of the planning graph. Every change considered in this ex-

[12] Graphplan plus Satz was run using the default settings of Blackbox for all the problems, except for Gripper-10/12 in which we used the settings suggested in (Kautz & Selman 1999) for hard logistics problems. Graphplan plus Walksat iteratively performed a graph search for 30 seconds, and then ran Walksat using the same settings used for Table I, except for Gripper-10/12 and bw_large_b where we used cutoff 3000000, 10 restarts and noise 0.2 (for Gripper-12 we tested cutoff 30000000 as well, but this did not help Blackbox, which did not find a solution.)

[13] These tests were performed on a Sun Ultra 10, 64Mb.

Log_a	W	T	T-W	ADJUST	IPP
I_1	0.008	0.009	0.019	0.1	1742
I_2	0.32	30.3	0.264	0.73	236
I_3	0.117	29.7	0.214	0.93	1198
I_4	0.318	141.9	0.231	1.9	1186
I_5	0.008	0.005	0.006	0.31	3047
G_1	0.502	162.1	0.31	0.88	69.6
G_2	0.008	0.007	0.008	0.32	864
G_3	0.066	142	0.03	0.22	3456
G_4	0.111	15.19	0.194	0.77	2846.6
G_5	0.009	0.389	0.005	0.37	266.3
G_6	0.006	0.196	0.005	0.09	459.5
G_7	0.005	0.034	0.006	0.23	491
G_8	0.006	0.006	0.005	0.04	520
G_9	0.102	0.464	0.068	0.77	663
G_10	0.162	22.145	0.034	0.78	445
G_11	0.121	160.8	0.284	0.77	1023

Table III: Plan-adaptation CPU-seconds required on average by the local search methods (20 runs), by ADJUST-PLAN and by IPP for some modifications of Logistics_a. W indicates Walkplan, T Tabuplan and T-W T-walkplan.

periment admits a plan with the same number of time steps as in the input plan. Some additional results for this experiment are given in (Gerevini & Serina 1999).

Further experimental results concern the use of ADJUST-PLAN for solving plan-modification problems. In particular, the fifth column of the Table III gives the CPU-time for 16 modifications of Logistics_a. These results together with others concerning 44 modifications of Logistics_b (Gerevini & Serina 1999), Rocket_a and Rocket_b indicate that adjusting a plan using ADJUST-PLAN is much more efficient than a complete replanning with IPP (up to three orders of magnitude faster).

Finally, we are currently testing a method for solving plan-adaptation problems based on a combination of local search and ADJUST-PLAN. The general idea is that we first try to adapt the plan using local search and without increasing the number of time steps in the plan. Then, if the local search was not able to efficiently adapt the plan and this contains only a few flaws, we try to repair it using ADJUST-PLAN, otherwise we use GPG with the combination of local and systematic search described in the previous section. Preliminary results in the Logistics and Rocket domains indicate that the approach is very efficient.

Conclusions

We have presented a new framework for planning through local search in the context of planning graph, as well as methods for combining local and systematic search techniques, that can be used for both plan-generation and plan-adaptation tasks.

Experimental results show that our methods can be much more efficient than the search methods currently used by planners based on the planning graph approach. Current work includes further experimental analysis and the study of further heuristics for the local search and the plan-adjustment phases of GPG.

Our search techniques for plan-generation have some similarities with Blackbox. A major difference is that, while Blackbox (Walksat) performs the local search on a CNF-translation of the graph, GPG performs the search directly on the graph structure. This gives the possibility of specifying further heuristics and types of search steps exploiting the semantics of the graph, which in Blackbox's translation is lost. For example, if an action node that was inserted to support a fact f violates some exclusion constraint c, then we could *replace* it with another action node, which still supports f and does not violate c. This kind of replacement operators are less natural to specify and more difficult to implement using a SAT-encoding of the planning problem.

Another significant difference is the use of local search for computing a quasi-solution, instead of a complete solution, which is then repaired by a plan-adjustment algorithm.

Acknowledgments

This research was supported in part by CNR project SCI*SIA. We thank Yannis Dimopoulos, Len Schubert and the anonymous referees for their helpful comments.

References

Ambite, J. L., and Knoblock, C. A. 1997. Planning by rewriting: Efficiently generating high-quality plans. In *Proc. of AAAI-97*, 706–713. AAAI/MIT Press.

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proc. of IJCAI-95*, 1636–1642.

Clark, D. A.; Frank, J.; Gent, I. P.; MacIntyre, E.; Tomov, N.; and Walsh, T. 1997. Local search and the number of solutions. In *Proc. of CP-97*, 119–133. Springer Verlag.

Gerevini, A., and Serina, I. 1999. Fast Planning through Greedy Action Graphs. Tech. Rep. 710, Computer Science Dept., Univ. of Rochester, Rochester (NY), USA.

Glover, F., and Laguna, M. 1993. Tabu search. In Reeves, C. R., ed., *Modern heuristics for combinatorial problems.* Oxford, GB: Blackwell Scientific.

Glover, F.; Taillard, E.; and de Werra, D. 1993. A user's guide to tabu search. *Annals of Oper. Research.* 41:3–28.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of AAAI-96*, 1194–1201.

Kautz, H., and Selman, B. 1998. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proc. of AIPS-98.*

Kautz, H., and Selman, B. 1999. Blackbox (version 3.4). `http://www.research.att.com/~kautz/blackbox`.

Koehler, J.; Nebel, B.; Hoffman, J.; and Dimopoulos, Y. 1997. Extending planning graphs to an ADL subset. In *Proc. of ECP'97.* Springer Verlag.

Minton, S.; Johonson, M.; Philips, A.; and Laird, P. 1992. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* 58:161–205.

Selman, B., and Kautz, H. 1994. Noise Strategies for Improving Local Search. In *Proc. of AAAI-94*, 337–343.

Serina, I., and Gerevini, A. 1998. Local search techniques for planning graph. In *Proc. of the 17th UK Planning and Scheduling SIG Workshop.*