LPG: A Planner Based on Local Search for Planning Graphs with Action Costs

Alfonso Gerevini and **Ivan Serina**
Dipartimento di Elettronica per l'Automazione, Università degli Studi di Brescia
Via Branze 38, I-25123 Brescia, Italy
{gerevini,serina}@ing.unibs.it

Abstract

We present LPG, a fast planner using local search for solving planning graphs. LPG can use various heuristics based on a parametrized objective function. These parameters weight different types of inconsistencies in the partial plan represented by the current search state, and are dynamically evaluated during search using Lagrange multipliers. LPG's basic heuristic was inspired by Walksat, which in Kautz and Selman's Blackbox can be used to solve the SAT-encoding of a planning graph. An advantage of LPG is that its heuristics exploit the structure of the planning graph, while Blackbox relies on general heuristics for SAT-problems, and requires the translation of the planning graph into propositional clauses. Another major difference is that LPG can handle action costs to produce good quality plans. This is achieved by an "anytime" process minimizing an objective function based on the number of inconsistencies in the partial plan and on its overall cost. The objective function can also take into account the number of parallel steps and the overall plan duration. Experimental results illustrate the efficiency of our approach showing, in particular, that for a set of well-known benchmark domains LPG is significantly faster than existing Graphplan-style planners.

Introduction

Blum and Furst's "planning graph analysis" (1997) has become a popular approach to planning on which various planners are based (e.g., (Blum and Furst 1997; Fox and Long 1998; Kautz and Selman 1999; Koehler *et al.* 1997; Kambhampati, Minh and Do 2001; Smith and Weld 1999)).

In this paper we present LPG, a planner based on local search and planning graphs, which considerably extends preliminary techniques introduced in (Gerevini and Serina 1999). The search space of LPG is formed by particular subgraphs of a planning graph, called *action graphs*, representing partial plans. The operators for moving from one search state to the next one are some graph modifications corresponding to particular revisions of the represented partial plan.

LPG uses various heuristics based on a parametrized evaluation function, where the parameters weight different types of inconsistencies in an action graph, and are dynamically evaluated using Lagrange multipliers. LPG's basic heuristic, Walkplan, was inspired by Walksat (Selman, Kautz, and Cohen 1994), a stochastic local search procedure which in Kautz and Selman's Blackbox (1999) can be used to solve the SAT-encoding of a planning graph (Kautz, McAllester, and Selman 1996).

An advantage of Walkplan with respect to Blackbox is that it operates directly on the planning graph. The search neighborhood of Walkplan is defined and evaluated by exploiting the structure of the graph, which in the SAT-translation of the graph is lost, or hidden. Another difference regards the number of parallel steps in the plan (length of the plan). LPG can increment the length of the plan during the search process, while in SAT-based planners this is done off-line.

Most existing planners have a limited notion of plan quality that is based on the number of actions, or parallel time steps. LPG handles a more accurate notion taking into account the execution cost of an action. Execution costs are represented as numbers associated with the actions. The search uses this information to produce plans of good quality by minimizing an objective function including a term for the evaluation of the overall execution cost of a plan. This is performed through an "anytime" process generating a succession of plans, each of which improves the previous ones. A similar process can be used to generate plans of good quality in terms of their number of parallel steps or their overall duration.

Experimental results illustrate the efficiency of our approach for plan generation indicating, in particular, that LPG is significantly faster than four other planners based on planning graphs: IPP4.1 (Koehler *et al.* 1997), STAN3s (Fox and Long 1998), GP-CSP (Kambhampati, Minh and Do 2001), and the recent version 3.9 of Blackbox. Other experimental results show the effectiveness of our incremental process for generating good quality plans.

The second section introduces our basic local search techniques for planning graphs; the third section presents some techniques improving the search process; the fourth section concerns plan quality in LPG; the fifth section gives our experimental results; the final section gives conclusions and mentions further work.

Local Search in the Space of Action Graphs

In this section we give the necessary background on planning graphs (Blum and Furst 1997), action graphs, and the basic local search techniques introduced in (Gerevini and Serina 1999).

Planning Graphs and Action 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. An action node at level t represents an action (instantiated operator) that can be planned at time step t. A fact node represents a proposition corresponding to a precondition of one or more 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 represent the positive facts of the initial state of the planning problem (every fact that is not mentioned in the initial state is assumed false).

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* from 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 $[a]$. Each fact node f at a level l is associated with a *no-op* action node at the same level, which represents a dummy action having $[f]$ as its only precondition and effect.

Two action nodes a and b are marked as mutually exclusive in the graph when one of the actions deletes a precondition or add-effect of the other (*interference*), or when a precondition node of a and a precondition node of b are marked as mutually exclusive (*competing needs*). Two proposition nodes p and q in a proposition level are marked as exclusive if all ways of making proposition $[p]$ true are exclusive with all ways of making $[q]$ true (each action node a having an add-edge to p is marked as exclusive of each action node b having an add-edge to q). When two facts or actions are marked as mutually exclusive, we say that there is a *mutex* relation between them.

A fact node q of level i is *supported* in a subgraph $\mathcal{G}'$ of a planning graph $\mathcal{G}$ if either (i) in $\mathcal{G}'$ there is an action node at level $i-1$ representing an action with (positive) effect $[q]$, or (ii) $i = 0$ (i.e., $[q]$ is a proposition of the initial state).

Given a planning problem Π, the corresponding planning graph $\mathcal{G}$ can be incrementally constructed level by level starting from level 0 using a polynomial algorithm. The last level of the graph is a propositional level where the goal nodes are present, and there is no mutex relation between them. Without loss of generality, we will assume that the goal nodes of the last level represent preconditions of a special action $[a_{end}]$, which is the last action in any valid plan.

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

A **solution graph** *for $\mathcal{G}$ is an action graph $\mathcal{A}_s$ of $\mathcal{G}$ such that all precondition nodes of the actions in $\mathcal{A}_s$ are supported, and there is no mutex relation between action nodes of $\mathcal{A}_s$.*

A solution graph for $\mathcal{G}$ represents a valid plan for Π. If the search for a solution graph fails, $\mathcal{G}$ can be iteratively expanded by adding an extra level, and performing a new search using the resultant graph.

Basic Search Neighborhood and Heuristics

Given a planning graph $\mathcal{G}$, the local search process starts from an initial A-graph $\mathcal{A}$ of $\mathcal{G}$ (i.e., a partial plan), and transforms it into a solution graph through the iterative application of some graph modifications improving the current partial plan. The two basic modifications consist of adding a new action node to the current A-graph, or removing an action node from it (together with the relevant edges).

At any step of the search process, which produces a new A-graph, the set of actions that can be added or removed is determined by the constraint violations that are present in the current A-graph. A **constraint violation** (or *inconsistency*) is either a mutex relation involving action nodes in the current A-graph, or an unsupported fact which is precondition of an action in the current partial plan.

The general scheme for searching a solution graph (a final state of the search) consists of two main steps. The first step is an initialization of the search in which we construct an initial A-graph. The second step is a local search process in the space of all A-graphs, starting from the initial A-graph. We can generate an initial A-graph in several ways. Three possibilities that can be performed in polynomial time, and that we have implemented in LPG are: (i) a randomly generated A-graph; (ii) an A-graph where all precondition facts are supported (but in which there may be some violated mutex relations); an A-graph obtained from an existing plan given in input to the process.

Once we have computed an initial A-graph, each basic search step randomly chooses an inconsistency in the current A-graph. If this is an unsupported fact node, then in order to resolve (eliminate) it, 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 chosen inconsistency is a mutex relation, then we can remove one of the action nodes of the mutex relation. When we add (remove) an action node to deal with an inconsistency, we also add (remove) all edges connecting the action node with the corresponding precondition and effect nodes in the planning graph.

Given a planning graph $\mathcal{G}$, an action graph $\mathcal{A}$ of $\mathcal{G}$ and a constraint violation s in $\mathcal{A}$, the **neighborhood** $N(s, \mathcal{A})$ of s in $\mathcal{A}$ is the set of action graphs obtained from $\mathcal{A}$ by applying a graph modification that resolves s. At each step of the local search, the elements in the neighborhood are weighted according to an evaluation function, and an element with the best evaluation is then chosen as the possible next subgraph (search state). Given an action subgraph $\mathcal{A}$ of a planning graph $\mathcal{G}$, the **general evaluation function** F of $\mathcal{A}$ is defined as follows:

$$F(\mathcal{A}) = \sum_{a \in \mathcal{G}} mutex(a, \mathcal{A}) + precond(a, \mathcal{A})$$

where:

$$mutex(a, \mathcal{A}) = \begin{cases} 0 & \text{if } a \notin \mathcal{A} \\ me(a, \mathcal{A}) & \text{if } a \in \mathcal{A} \end{cases}$$

$$precond(a, \mathcal{A}) = \begin{cases} 0 & \text{if } a \notin \mathcal{A} \\ pre(a, \mathcal{A}) & \text{if } a \in \mathcal{A} \end{cases}$$

and $me(a, \mathcal{A})$ is the number of action nodes in $\mathcal{A}$ which are mutually exclusive with a, while $pre(a, \mathcal{A})$ is the number of precondition facts of a which are not supported in $\mathcal{A}$. It is easy to see that, when the search reaches an A-graph for which F is zero, this is a solution graph.

The simple use of the general function to guide the local search has the drawback that it can lead to local minima from which the search can not escape. For this reason, instead of using F, we use a parametrized **action evaluation function** E which allows to specify different types of heuristics. This function specifies the cost of inserting (E^i) and of removing (E^r) an action $[a]$ in the partial plan represented by the current action subgraph $\mathcal{A}$:

$$E([a], \mathcal{A})^i = \alpha^i \cdot pre(a, \mathcal{A}) + \beta^i \cdot me(a, \mathcal{A}) + \gamma^i \cdot unsup(a, \mathcal{A})$$

$$E([a], \mathcal{A})^r = \alpha^r \cdot pre(a, \mathcal{A}) + \beta^r \cdot me(a, \mathcal{A}) + \gamma^r \cdot sup(a, \mathcal{A}),$$

where $me(a, \mathcal{A})$ and $pre(a, \mathcal{A})$ are defined as in F; $unsup(a, \mathcal{A})$ is the number of unsupported precondition facts in $\mathcal{A}$ that become supported by adding a to $\mathcal{A}$; $sup(a, \mathcal{A})$ is the number of supported precondition facts in $\mathcal{A}$ that become unsupported by removing a from $\mathcal{A}$; α, β and γ are parameters specializing the function.

By appropriately setting the values of these coefficients we can implement different heuristic methods aimed at making the search less influenced by local minima. Their values have to satisfy the following constraints: α^i, $\beta^i > 0$ and $\gamma^i \leq 0$ in E^i; α^r, $\beta^r \leq 0$ and $\gamma^r > 0$ in E^r. Note that the positive coefficients of E (α^i, β^i and γ^r) determine an increment of E which is related to an increment of the number of inconsistencies. Analogously, the non-positive coefficients of E determines a decrement of E which is related to a decrement of the number of inconsistencies.

In (Gerevini and Serina 1999) we proposed three basic heuristics to guide the local search: *Walkplan*, *Tabuplan* and *T-Walkplan*. In this paper we focus on Walkplan, which is similar to the heuristic used by Walksat (Selman, Kautz, and Cohen 1994). In Walkplan we have $\gamma^i = 0$, $\alpha^r = 0$, and $\beta^r = 0$, i.e., the action evaluation function is

$$E([a], \mathcal{A})^i = \alpha^i \cdot pre(a, \mathcal{A}) + \beta^i \cdot me(a, \mathcal{A})$$

$$E([a], \mathcal{A})^r = \gamma^r \cdot sup(a, \mathcal{A}).$$

The best element in the neighborhood is the A-graph introducing the fewest new constraint violations (Walkplan does not consider the constraint violations of the current A-graph that are resolved). Like Walksat, Walkplan uses a *noise parameter p*: given an A-graph $\mathcal{A}$ and a (randomly chosen) constraint violation, if there is a modification that does not introduce new constraint violations, then the corresponding A-graph in $N(s, \mathcal{A})$ is chosen as the next A-graph; otherwise, with probability p one of the graphs in $N(s, \mathcal{A})$ is chosen randomly, and with probability $1-p$ the next A-graph is chosen according to the minimum value of the action evaluation function.

Neighborhood Refinements

In this section we present some improvements of the basic local search technique described in the previous section.

Dynamic Heuristics Based on Lagrange Multipliers

As shown in (Gerevini and Serina 2001) the values of the coefficients α, β and γ in E can significantly affect the efficiency of the search. Since their value is static, the optimal performance can be obtained only when their values are appropriately tuned before search. Moreover, the best parameter setting can be different for different planning domains, or even for different problems in the same domain. In order to cope with this drawback, we have revised the evaluation functions by weighting their terms with *dynamic* coefficients similar to the *Lagrange multipliers* for discrete problems that have been used for solving SAT problems (Shang and Wah 1998; Wah and Wu 1999; Wu and Wah 2000).

The use of these multipliers gives two important improvements to our local search. First, the revised cost function is more informative, and can discriminate more accurately the elements in the neighborhood. Secondly, the new cost function does not depend on any static coefficient (the α, β and γ coefficients can be omitted, and the initial default value of all new multipliers is scarcely important).

The general idea is that each constraint violation is associated with a dynamic multiplier that weights it. If the value of the multiplier is high, then the corresponding constraint violation is considered "difficult", while if the value is low it is considered "easy". Using these multipliers, the quality of an A-graph will be estimated not only in terms of the number of constraint violations that are present in it, but also in terms of an estimation of the difficulty to solve the particular constraint violations that are present.

Since the number of constraint violations that can arise during the search can be very high, instead of maintaining a distinct multiplier for each of them, our technique assigns a multiplier to a *set* of constraint violations associated with the same action. Specifically, for each action a, we have a multiplier λ_m^a for all mutex relations involving a, and a multiplier λ_p^a for all preconditions of a. Intuitively, these multipliers estimate the difficulty of satisfying all preconditions of an action, and of avoiding to have that a is mutually exclusive with any other action in the A-graph. The multipliers can be used to refine the cost $E([a], \mathcal{A})^i$ of inserting the action node a and the cost $E([a], \mathcal{A})^r$ of removing a. In particular, the evaluation function for Walkplan becomes

$$E_\lambda([a], \mathcal{A})^i = \lambda_p^a \cdot pre(a, \mathcal{A}) + \lambda_m^a \cdot me(a, \mathcal{A})$$

$$E_\lambda([a], \mathcal{A})^r = \Delta_a^-,$$

where Δ_a^- is the sum of the supported preconditions in $\mathcal{A}$ that become unsupported by removing a, weighted by $\lambda_p^{a_j}$, i.e.,

$$\Delta_a^- = \sum_{a_j \in \mathcal{A}-\{a\}} \lambda_p^{a_j} \cdot (pre(a_j, \mathcal{A} - [a]) - pre(a_j, \mathcal{A})).$$

The multipliers have all the same default initial value, which is dynamically updated during the search. Intuitively, we want to increase the multipliers associated with actions introducing constraint violations that turn out to be difficult to solve, while we want to decrease the multipliers associated with actions whose constraint violations tend to be easier. These will lead to prefer graphs in the neighborhood that appear to be globally easier so solve (i.e., that are closer to a solution graph).

The update of the multipliers is performed whenever the local search reaches a local minimum or a plateau (i.e., when all the elements in the neighborhood have E_λ greater than zero). The values of the multipliers of the actions that are responsible of the constraint violations present in the current action graph are then increased by a certain small quantity δ^+, while the multipliers of the actions that do not determine any constraint violation are decreased by δ^-. For example, if a is in the current A-graph and is involved in a mutex relation with another action in the graph, then λ_m^a is increased by a predefined positive small quantity δ^+ (otherwise it is decreased by δ^-).[1]

No-op Propagation

This technique is based on the observation that, whenever an action is added to the current plan (action graph), its effects can be propagated to the next time steps (levels), unless there is an action interfering with them. Suppose that P is one of the effects of an action a at time step (level) t. P is propagated to the next levels by adding the corresponding no-ops, until we reach a level where the no-op for P is mutually exclusive with some other action in the current plan. Notice that in order to guarantee soundness, whenever we add an action at some time step after t which is mutex with a propagated no-op, it is necessary to "block" this no-op at level t. Similarly, when an action that blocks a no-op is removed, this no-op can be further propagated (unless there is another action blocking it). The no-op propagation can be efficiently maintained by exploiting the data structures of the planning graph.

Clearly, the propagation is useful because it can eliminate further inconsistencies (an action is added to support a specific precondition, but the effects of this action could also support additional preconditions through the propagation of their no-ops). Moreover, the no-op propagation can be exploited to extend the search neighborhood, and possibly reduce the number of search steps. Given an unsupported precondition Q of and action a at level l of an A-graph $\mathcal{A}$, the basic neighborhood of $\mathcal{A}$ for Q consists of the graphs derived from $\mathcal{A}$ by either removing a or adding an action at level $l-1$ which supports Q. In principle, among these actions we could use the no-op for Q (if available), which corresponds to assuming that Q is achieved by an earlier action (possibly the initial state), and persists up to the time step when a is executed. Instead of using this no-op, we use an action at an earlier level which has Q as one of its effects, and such that Q can be propagated up to l. When there is more than one of such actions, the neighborhood includes all corresponding action graphs derived by adding any one of them.

Experimental results showed that the use of the no-op propagation can significantly reduce both the number of search steps and the CPU-time required to find a solution graph.

Action Ordering and Incremental Graph Length

In general, local search methods do not examine the search space in an exhaustive way, and the hence also the search techniques described in the previous section can not determine when it is necessary to increase the length (number of levels) of the current planning graph. Originally, we overcame this limitation by automatically expanding the graph after a certain number of search steps. However, this has the drawback that, when the graph is extended several times, a significant amount of CPU-time is wasted for searching the sequence of expanded graphs that are generated before the last expansion.

Recently, we have developed an alternative method for expanding the graph *during* search that leads to better performance. In particular, we have extended the search neighborhood with A-graphs derived by two new types of graph modifications that are based on ordering mutex actions and possibly expanding the graph. Given two actions that are mutex at time step i, we examine if this inconsistency can be eliminated by postponing to time step $i+1$ one of the actions, or by anticipating it a step $i-1$. We say that an action a at step i can be postponed to step $i+1$ of $\mathcal{A}$ if the action graph obtained by moving a from i to $i+1$ does not contain new inconsistencies with respect to $\mathcal{A}$.[2] The condition for anticipating a to $i-1$ is analogous.

When one of the actions involved in a mutex relation can be postponed or anticipated, the corresponding action graph is always chosen from the neighborhood. If more than one of these modifications is possible, we choose randomly one of the corresponding A-graphs. If none of the two actions can be postponed (or anticipated), we try to postpone (anticipate) one of them in combination with a graph extension.

[1] In order to make the increment of λ_p^a more accurate, in our implementation δ^+ is weighted by taking into account the proportion k of unsupported preconditions of a with respect to the total number of unsupported preconditions in the action graph (i.e, λ_p^a is increased by $k \cdot \delta^+$). Similarly, when we increase λ_m^a, δ^+ is weighted by the proportion of mutex involving a with respect to the total number of mutex in the graph. Moreover, the unbounded increase or decrease of a multiplier is prevented by imposing maximum and minimum values to them. The default values for δ^+ and δ^- used in our tests are 10^{-3} and $5 \cdot 10^{-6}$ respectively.

[2] In order to simplify the notation, in this and the next sections we indicate with a (f) both an action node (fact node) and the corresponding represented action (fact). The actual meaning of the notation will be clear from the context.

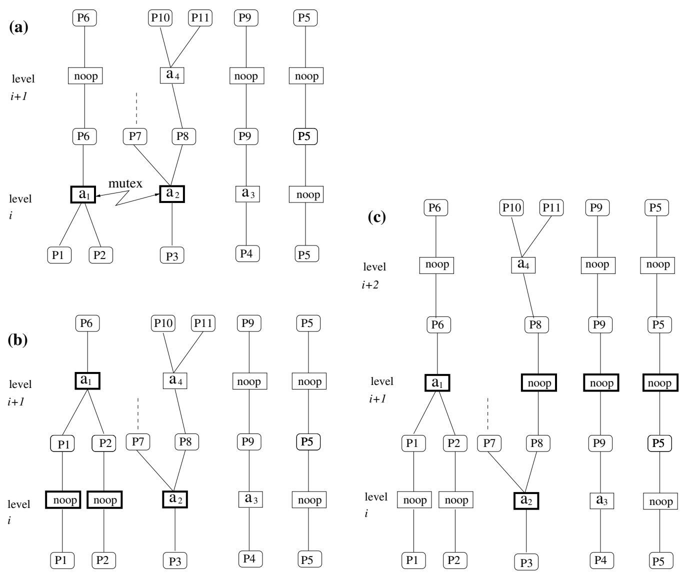

Figure 1: Examples of action ordering and increment of graph length to deal with a mutex relation.

Specifically, first we extend the planning graph of one level; then we shift forward all actions from step $i+1$ ($i-1$); finally we move one of the two actions to step $i+1$ ($i-1$), provided that this move does not introduce new inconsistencies (e.g., the moved action blocks a no-op that is used to support a precondition at a later step). If none of the two actions can be moved even after a graph extension, the action graphs of the neighborhood are those derived by simply removing one of the two actions.

For example, in Figure 1(a) we have that actions a_1 and a_2 are mutex at level i. Assume that a_1 has two supported preconditions P_1 and P_2, and one additive effect P_6 that is needed by other actions in the following levels of the graph. Fact P_8 is supported by a_2 and is a precondition of a_4; fact P_7 is not used to support any precondition. Figure 1(b) illustrates a case in which a_1 can be postponed to level $i+1$. This is possible because there is no mutex relation between (1) a_1 and a_4, (2) a_1 and the no-ops for P_5 and P_9, (3) the no-ops supporting the preconditions of a_1 and any other action at level i.

Figure 1(c) illustrates a case in which a_1 can not be postponed to level $i+1$ (because, for example, a_4 deletes a precondition of a_1), but it can be postponed in combination with a graph expansion. This is possible because there is no mutex relation between (1) a_1 and the no-ops of P_5, P_8, P_9, (2) between the no-ops supporting the preconditions of a_1 and any other action of level i.

Heuristic Precondition Costs

The heuristic functions E and E_λ previously described evaluate the A-graphs in the neighborhood by considering their number of unsupported preconditions and unsatisfied mutex relations. An experimental analysis revealed that, while computing this functions is quite fast, sometimes a more accurate evaluation can be very useful. In fact, it can be the case that, although the insertion of a new action a_1 leads to fewer new unsupported preconditions than those introduced by an alternative action a_2, the unsupported preconditions of a_1 are more difficult to satisfy (support) than those of a_2 (i.e., the inconsistencies that a_1 would introduce require more search steps than the inconsistencies of a_2). For this reason we have refined E_λ by using an heuristic estimation (H) of the difficulty of supporting a precondition which exploits information stored in the structure of the underlying planning graph. More precisely, $E_\lambda(a,\mathcal{A})^i$ becomes

$$E_H(a,\mathcal{A})^i = \lambda_p^a \cdot \underset{f\in pre(a)}{MAX}\ H(f,\mathcal{A}) + \lambda_m^a \cdot me(a,\mathcal{A}),$$

where $H(f,\mathcal{A})$ is the *heuristic cost of supporting* f, and is recursively defined in the following way:

$$H(f,\mathcal{A}) = \begin{cases} 0 & \text{if } f \text{ is supported} \\ H(f',\mathcal{A}) & \text{if } a_f \text{ is no-op with precondition } f' \\ \underset{f'\in pre(a_f)}{MAX} H(f',\mathcal{A}) + me(a_f,\mathcal{A}) + 1 & \text{otherw.} \end{cases}$$

where

$$a_f = \underset{\{a'\in A_f\}}{ARGMIN} \left\{E(a',\mathcal{A})^i\right\}$$

and A_f is the set of action nodes of the planning graph at the level preceding the level of f, that have f as one their effects nodes. Informally, the heuristic cost of an unsupported fact f is determined by considering all actions at the level preceding the level of f whose insertion would support f. Among these actions, we choose the one with the best evaluation (a_f) according to the original action evaluation function E.[3] $H(f,\mathcal{A})$ is recursively computed by summing the number of mutex relations introduced by a_f to the maximum of the heuristic costs of its unsupported preconditions ($H(f',\mathcal{A})$). The last term "+1" takes into account the insertion of a_f to support f.

Similarly, we refine the cost of removing an action a from $\mathcal{A}$ by considering the maximum heuristic cost of supporting a fact that becomes unsupported when a is removed. More precisely,

$$E_H(a,\mathcal{A})^r = \underset{\langle f,a_j\rangle\in K}{MAX}\ \lambda_p^{a_j} \cdot H(f,\mathcal{A}-a)$$

where K is the set of pairs $\langle f, a_j\rangle$ such that f is a precondition of an action node a_j which becomes unsupported by removing a from $\mathcal{A}$.

In Figure 2 we give an example illustrating H. Each action node is associated with a pair of numbers (in brackets) indicating the number of unsupported preconditions and the number of mutex relations involving the node. For example, a_3 at level $l+1$ has two unsupported preconditions and is involved in 5 mutex relations. Each fact node is associated with a number specifying the heuristic cost of supporting it. Suppose we want to determine the heuristic cost of the unsupported precondition P_1 at level $l+2$. The action with

[3] When more than one action with minimum cost exists, if the no-op is one of these actions, then we choose it, otherwise we choose randomly among them.

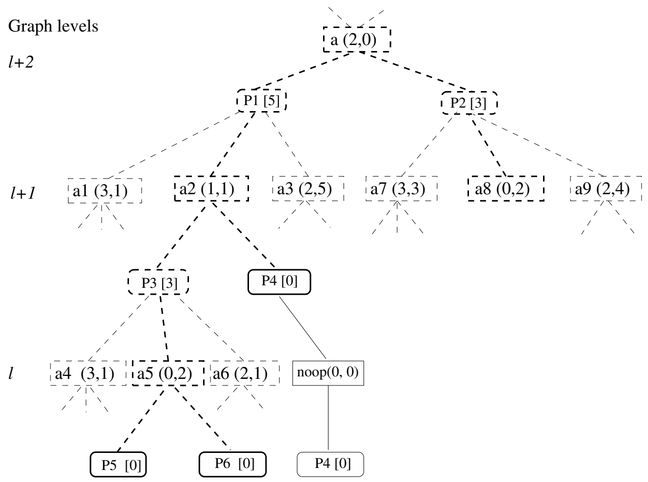

Figure 2: Example illustrating the heuristic cost of unsupported preconditions. Dashed boxes correspond to actions/facts which do not belong to $\mathcal{A}$

minimum cost achieving P_1 is a_2. The action with minimum cost achieving P_3 is a_5, which has all preconditions supported. Thus we have

$$H(P_3, \mathcal{A}) = \underset{f' \in pre(a_5)}{MAX} H(f', \mathcal{A}) + me(a_5, \mathcal{A}) + 1 = 0+2+1.$$

Since $H(P_4, \mathcal{A}) = 0$, we have

$$H(P_1, \mathcal{A}) = \underset{f' \in pre(a_2)}{MAX} H(f', \mathcal{A}) + me(a_2, \mathcal{A}) + 1 = 3+1+1.$$

Modelling Plan Quality

In this section we extend our techniques to manage plans where each action has an arbitrary number associated with it representing, for example, its execution cost. While taking this information into account is practically very important to produce good quality plans, most of the domain-independent planners in the literature neglect it, and use the number of actions (or parallel steps) as a simple measure of plan quality. In our framework execution costs can be easily modelled by extending the objective function of the local search with a term evaluating the overall execution cost of a plan (action graph), which is naturally defined as the sum of the costs of its actions. Similarly, we can easily model the number of parallel steps in a plan, and use this criterion as a measure of plan quality.

The user can weight the "optimization" terms of the revised objective function (execution cost and number of steps) according to the relative importance. More formally, the general evaluation function F can be revised to

$$F^+(\mathcal{A}) = \mu_C \cdot \sum_{a \in \mathcal{A}} Cost(a) + \mu_S \cdot Steps(\mathcal{A}) + \\ + \sum_{a \in \mathcal{A}} (\lambda_m^a \cdot mutex(a, \mathcal{A}) + \lambda_p^a \cdot precond(a, \mathcal{A}))$$

where $Cost(a)$ is the execution cost of a, μ_C and μ_S are non-negative coefficients set by the user to weight the relative importance of execution cost and number of parallel steps ($\mu_C + \mu_S \leq 1$), and $Steps(\mathcal{A})$ is the number of steps in the A-graph $\mathcal{A}$ containing at least one action different from no-ops. The evaluation function E_H defined in the previous section is refined to E_H^+ in the following way (U denotes the set of preconditions achieved by a that become unsupported by removing a):

$$\begin{aligned} E_H^+(a, \mathcal{A})^i = {} & \frac{\mu_C}{max_{CS}} \cdot (Cost(a) + \underset{f \in pre(a)}{MAX} H_C(f, \mathcal{A})) + \\ & + \frac{\mu_S}{max_{CS}} \cdot (Steps(\mathcal{A} + a) - Steps(\mathcal{A}) + \\ & \qquad \underset{f \in pre(a)}{MAX} H_S(f, \mathcal{A})) + \\ & + \frac{1}{max_E} \cdot E_H(a, \mathcal{A})^i \end{aligned}$$

$$\begin{aligned} E_H^+(a, \mathcal{A})^r = {} & \frac{\mu_C}{max_{CS}} \cdot (-Cost(a) + \underset{f \in U}{MAX} H_C(f, \mathcal{A} - a)) + \\ & + \frac{\mu_S}{max_{CS}} \cdot (Steps(\mathcal{A} - a) - Steps(\mathcal{A}) + \\ & \qquad \underset{f \in U}{MAX} H_S(f, \mathcal{A} - a)) + \\ & + \frac{1}{max_E} \cdot E_H(a, \mathcal{A})^r \end{aligned}$$

where $H_C(f, \mathcal{A})$ and $H_S(f, \mathcal{A})$ are an estimation of the increase of the overall execution cost and parallel steps, respectively, that is required to support the fact f in $\mathcal{A}$. H_C and H_S are recursively defined in a way similar to the definition of H given in the previous section: if f is supported in $\mathcal{A}$, then $H_C(f, \mathcal{A}) = 0$ and $H_S(f, \mathcal{A}) = 0$, otherwise

$$H_C(f, \mathcal{A}) = Cost(a_f) + \underset{f' \in pre(a_f)}{MAX} H_C(f', \mathcal{A})$$

$$H_S(f, \mathcal{A}) = Steps(\mathcal{A} + a_f) - Steps(\mathcal{A}) + \underset{f' \in pre(a_f)}{MAX} H_S(f', \mathcal{A})$$

in which a_f is defined as in the definition of H. For example, consider again Figure 2 and suppose that $Cost(a_2) = 10$ and $Cost(a_5) = 30$. We have that $H_C(P_1, \mathcal{A}) = 40$.

The factors $1/max_{CS}$, and $1/max_E$ are used to normalize each term of E_H^+ to a value less than or equal to 1. Without this normalization the values of the terms corresponding to the execution cost (first term) could be much higher than the value corresponding to the constraint violations (third term). This would guide the search towards good quality plans without paying sufficient attention to their validity. On the contrary, especially when the current partial plan contains many constraint violations, we would like that the search give more importance to satisfying these logical constraints. The value of max_{CS} is defined as $\mu_C \cdot max_C + \mu_S \cdot max_S$, where max_C (max_S) is the maximum value of the first (second) term of E_H^+ over all A-graphs (partial plans) in the neighborhood, multiplied by the number κ of constraint violations in the current action graph; max_E is defined as the maximum value of E_H over all possible action insertions/removals that eliminate the inconsistency under consideration.[4]

The revised heuristic function E_H^+ can be used to produce a succession of valid plans where each plan is an improvement of the previous ones in terms of execution cost or parallel steps. The first of these plans is the first plan π without constraint violations that is generated (i.e., the third term of F^+ is zero). π is then used to initialize a second search which is terminated when a new valid plan π' that is better than π is generated. π' is then used to initialize a third search, and so on. This is an *anytime process* that incrementally improves the quality of the plans, and that can be stopped at any time to give the best plan computed so far. Each time we start a new search some inconsistencies are forced in the initial plan by randomly removing some of its actions. Similarly, some random inconsistencies are introduced during search when a valid plan that does not improve the plan of the previous search is reached.

This method of modelling plan quality can be extended to deal with a limited form of temporal planning where each action a is associated with a temporal duration $D(a)$. Under the assumption that the execution of all actions at each step of the plan terminates before starting executing the actions at the next step, we can search for parallel plans with the shortest possible duration by including in F^+ a new term that is proportional to the sum of the longest action at each step of the current partial plan. Let $Delay(a, \mathcal{A})$ be the temporal delay determined by a in the A-graph $\mathcal{A}$, i.e.,

$$Delay(a, \mathcal{A}) = MAX(0, D(a) - \underset{a' \in S}{MAX}\ D(a')),$$

where S is the set of actions at the same level of a in $\mathcal{A}$ (except a). The new terms in E_H^+ to handle plan duration are proportional to

$$+Delay(a, \mathcal{A}) + \underset{f \in pre(a)}{MAX}\ H_D(f, \mathcal{A}), \text{ in } E_H^+(a, \mathcal{A})^i$$

$$-Delay(a, \mathcal{A}) + \underset{f \in U}{MAX}\ H_D(f, \mathcal{A} - a), \text{ in } E_H^+(a, \mathcal{A})^r$$

where $H_D(f, \mathcal{A})$ is a heuristic estimation of the increase of the overall plan duration that is required to support f (the formal definition of $H_D(f, \mathcal{A})$ is similar to the definition of $H_C(f, \mathcal{A})$).

Despite the previous assumption could be relaxed in the computed plan by considering its causal structure (dependencies between action effects and preconditions), in general, it is not guaranteed that minimizing the duration term of F^+ always leads to the shortest possible parallel plans. On the other hand, when the domain admits only linear plans, minimizing the duration term corresponds to searching for optimal plans. Hence, our simple approach for duration optimization should be seen as an approximate technique.

[4]The role of κ is to decrease the importance of the first two optimization terms when the current plan contains many constraint violations, and to increase it when the search approaches a valid plan. For lack of space we omit further details on the formalization of max_C, max_S and max_E.

Experimental Results

Our techniques are implemented in a system called LPG (Local search for Planning Graphs). Using LPG we have conducted three experiments. The first experiment was aimed at comparing the performance of Walkplan in LPG and other planners based on planning graphs for some well-known benchmark problems. We considered Blackbox 3.9 with solvers Walksat and Chaff (Moskewicz *et al.* 2001); GP-CSP (Kambhampati, Minh and Do 2001); IPP4.1 (Koehler *et al.* 1997); and STAN3s (Fox and Long 1998; 1999).[5]

The results of this experiment are in Table 1. The 2nd and the 3rd columns of the table give the total time required by LPG, and the search time of Walksat for the SAT-encoding of the planning graph with the minimum number of levels that is sufficient to solve the problem.[6] Walkplan performed always better than Walksat, and was up to four orders of magnitude faster. The fourth column of Table 1 gives the results for Blackbox using the recent solver Chaff instead of Walksat, which currently is the fastest systematic SAT solver. LPG was significantly faster than Blackbox also in this case. The remaining columns of the table give the CPU-times for GP-CSP, IPP and STAN, which were up to 3 or 4 orders of magnitude slower than LPG. (However, it should be noted that these planners, as well as Blackbox with Chaff, compute plans with minimal number of steps.)

In the second experiment we tested the performance of LPG for some problems of the AIPS-98/00 competition set with respect to the planner that won the last competition, FF (Hoffmann and Nebel 2001). The results are in Table 2.[7] In terms of CPU-time LPG was comparable to FF. In Logistics LPG performed better than FF; in Elevator (`m10`-problems) FF performed better than LPG; while in Mprime we have mixed results.[8] For `mprime-05` FF required more than 600 seconds, while LPG determined that the problem is not solvable without search (this was detected during the construction of the planning graph). In terms of number of actions

[5]All tests were run on a Pentium II 400 MHz with 768 Mbytes. We did not use version 4 of STAN because it uses methods alternative to planning graphs.

[6]Without this information the times required by Blackbox using Walksat were significantly higher than those in the table. The parameters of Walksat (noise and cutoff) were set to the combination of values that gave the best performance over many tries. For Blocks-world and Gripper we used cutoff 5000000, various values of noise and several restarts. For `log-d` and `bw-large-a/b` we used the the option "`compact -b 8`", because it made the problems easier to solve. In this and in the next experiments the noise factor for Walkplan was always 0.1; the cutoff was initially set to 500, and then automatically incremented at each restart by a factor of 1.1.

[7]However, note that comparing these two systems is not completely fair, because FF computes linear plans while LPG computes parallel plans, and hence they solve (computationally) different problems.

[8]We considered a slight modification of Mprime where operator parameters with different names are bound to different constants, which is an implicit assumption in the implementation of both FF and LPG.

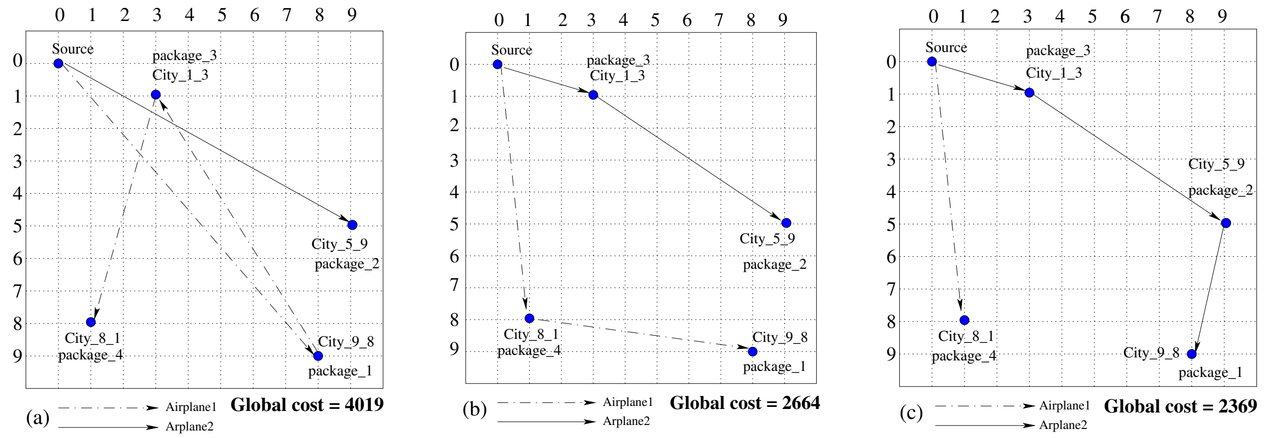

Figure 3: Three plans and the relative execution costs computed by FF (1st picture) and by LPG (2nd and 3rd pictures) for a simple logistics problem with action costs.

Planning problem	LPG Wplan	Blackbox Wsat	Blackbox Chaff	GP-CSP	IPP	STAN
`rocket-a`	0.05	1.25	5.99	1.55	20.2	6.49
`rocket-b`	0.06	1.51	6.16	3.02	38.83	4.24
`log-a`	0.22	3.21	5.93	1.60	777.8	0.24
`log-b`	0.28	5.76	6.74	22.7	341.0	1.11
`log-c`	0.32	14.28	7.19	28.8	—	896.6
`log-d`	0.42	35.10	11.5	98.0	—	—
`bw-large-a`	0.24	2.06	0.69	6.82	0.17	0.21
`bw-large-b`	0.61	131.0	51.6	783	12.39	5.4
`TSP-7`	0.02	0.14	0.11	0.13	0.04	0.01
`TSP-10`	0.03	0.72	6.47	8.48	1.96	0.04
`TSP-15`	0.07	31.23	—	—	419.0	0.26
`TSP-30`	0.39	*out*	—	—	—	11.9
`gripper10`	0.31	—	—	—	40.38	36.3
`gripper12`	0.74	—	—	—	330.1	810.2

Table 1: Average CPU-seconds over 25 runs required by LPG using Walkplan (Wplan) with E_H and other planners based on planning graph (Wsat indicates Walksat). The times for LPG are total search times; the times for Walksat refer only to the planning graph with the minimum number of levels where a solution exists; the times for Chaff and GP-CSP are the total times using the relative default settings, except for Gripper and the hardest TSP problems, where for Chaff we used the settings suggested by Blackbox for hard problems. "—" means that the planner did not find a solution after 1500 seconds (in these cases the average was over 8 runs); "*out*" that the system went out of memory.

Planning problem	LPG Time	LPG Steps	LPG Act.	FF Time	FF Act.
`log-35-0`	5.57 (4.76)	100.5	246.5	19.19	193
`log-36-0`	6.15 (4.79)	100.2	263.8	14.74	209
`log-37-0`	15.46 (7.36)	110.7	308.3	26.48	237
`log-38-0`	10.25 (7.46)	105.0	289.1	25.75	224
`log-39-0`	22.94 (7.49)	117.6	313.0	66.74	239
`m10-s20-0`	1.51 (0.81)	65.74	74.61	0.29	64
`m10-s21-0`	1.81 (0.94)	72.71	80.57	0.40	70
`m10-s22-0`	2.31 (1.10)	75.32	83.96	0.47	73
`m10-s23-0`	2.76 (1.26)	79.81	88.70	0.52	76
`m10-s24-0`	3.24 (1.45)	84.56	92.72	0.55	79
`mprime-01`	0.10 (1.92)	5.9	7.5	0.03	5
`mprime-02`	0.50 (8.72)	7.6	12.3	0.16	10
`mprime-03`	0.03 (3.90)	4.0	4.0	0.05	4
`mprime-04`	0.02 (1.56)	7.2	9.7	0.02	10
`mprime-05`	0.0 (2.43)	—	—	> 600	—

Table 2: CPU-seconds, number of steps and number of actions for LPG (Walkplan) and FF for some problems of the AIPS competitions. The data for LPG are averages over 25 runs. The CPU-time in brackets is the time required for constructing the graph. This is largely dominated by the instantiation of the operators (especially for `Mprime`), which in FF's implementation is very fast and in LPG can be drastically reduced by using a similar implementation.

for the problems of Table 2, FF-plans are better than LPG-plans except in 2 cases. We believe there are several reasons for this: LPG considers only one inconsistency at each search step, and in this sense its search is "more local" than FF search, whose heuristic considers all the problem goals; Walkplan, like Walksat, does not distinguish elements in the search neighborhood that improve the current A-graph; in this experiment the coefficient of the action cost term in E_H was set to a low value, because we were interested in the fast computation of the *first* solution.

However, as we have described in the previous section, LPG can generate good quality plans in a more general sense by minimizing the heuristic evaluation function E_H^+. The third experiment that we have conducted concerns the use of E_H^+ in LPG to produce plans of good quality. In particular, we have tested LPG with a modification of the Logistics domain where each action has an execution cost associated with it. Figure 3 shows three plans computed by FF and LPG for a simple problem in this domain (for the sake of clarity we show only the fly actions). We have two airplanes, four packages, and four cities, each of which has an airport, a

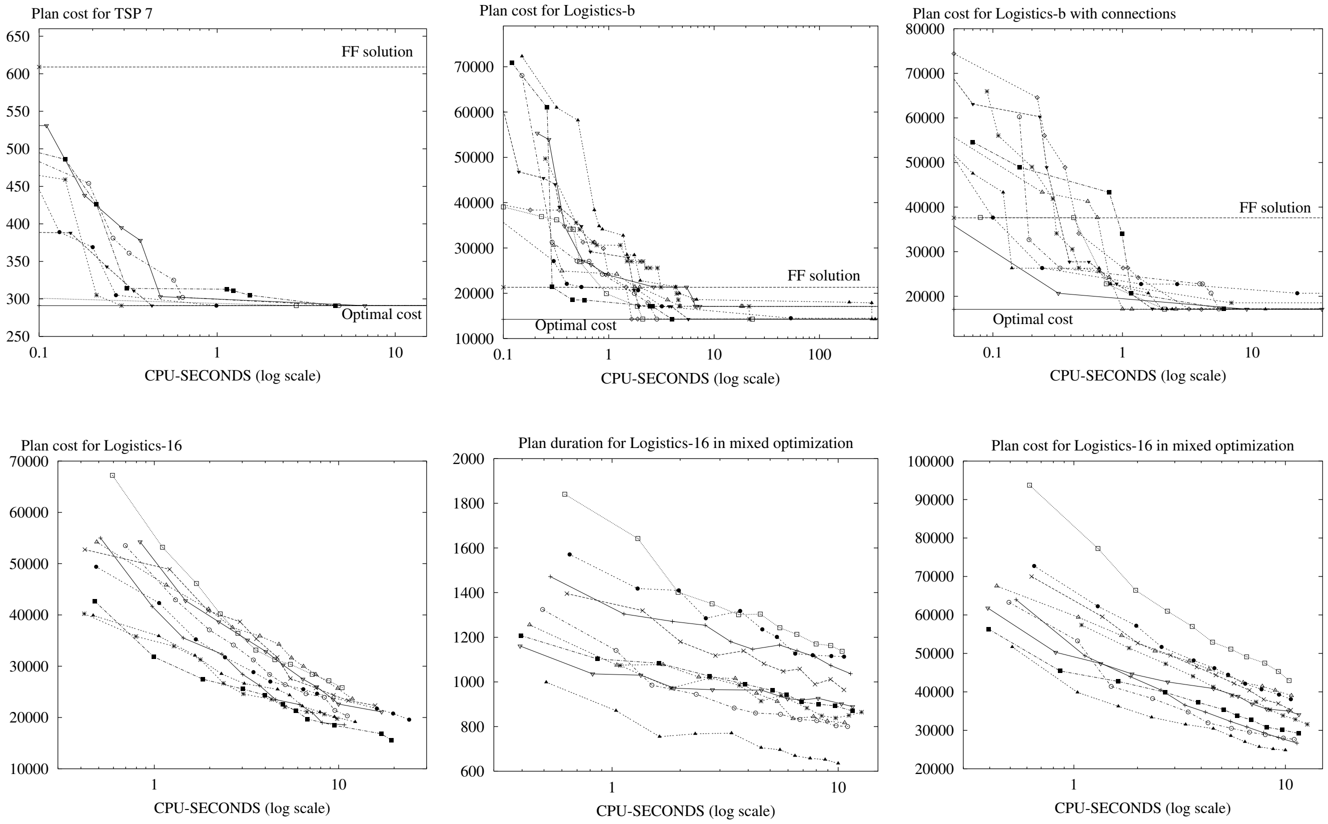

Figure 4: Incremental quality of the plans computed by LPG and relative CPU-time. The first three pictures plots refer to single problem instances (10 runs of LPG for each problem). Each of the other three pictures plots average measures (cost or duration) for 10 variants of `logistics-16`. For each variant we considered the sequence of the first 11 plans generated. The points connected refer to the average cost or duration for the same variant. Average data are over 25 runs of the incremental optimization process, for a total of 2750 plans in each picture.

port and a truck. Initially, all packages and airplanes are at the `source` city; the final locations of the packages is shown in the pictures of the figure. Cities and airports are placed in a grid, and the cost of flying between two airports depends the relative Euclidean distance in the grid (e.g., the distance between `city_9_8` to `city_1_3` in Figure 3(a) is the distance between points (9,8) and (1,3) on the grid). The cost of flying by `Airplane1` (fly1) is the travel distance times 100, while the cost of flying by `Airplane2` (fly2) is the travel distance times 50 (i.e., fly1 is twice as expensive as fly2); loading and unloading packages on/from trucks or airplanes have cost 1, driving a truck between the port and the airport of a city has cost 10.

Figure 3(a) shows the fly actions of the airplanes in the plan computed by FF, which does not take into account action costs, and produces a plan with overall cost 4019. Figures 3(b/c) show the fly actions in the first two plans computed by a run of LPG. The first plan has overall cost 2664, the second 2369. On average the optimal plan for this problem is computed by LPG in 0.85 seconds.

Figure 4 gives results for a TSP problem with 7 cities and further logistics problems. The horizontal lines indicate the optimal execution costs, and the cost of the solution computed by FF. In `TSP-7` and `logistics-b` cities and the relative airports were randomly placed in a grid of 100×100 locations, and the action costs were determined as in Figure 3. The third picture regards a modification of `logistics-b` in which only 60% of all possible pairs of cities are connected by direct flights (in the original version there is a direct flight between each pair of cities). This problem was automatically generated by a program that randomly generates variants of a Logistics problem by augmenting it with flight connections, and ensuring that each pair of cities is connected via some path of possible flights.

The plots of the first three pictures show the increasing plan quality in terms of execution cost for 10 runs of LPG (using E_H^+ with $\mu_C = 1.0$), indicating that LPG quickly found the optimal solution, or a solution close to the optimal one.

The fourth picture of Figure 4 plots the average execution costs for 10 randomly generated variants of a more difficult problem from the AIPS-00 competition (`logistics-16`). The variants differ in the initial position of the airports and cities on a grid of 100×100 locations, which was randomly determined. These results show that the incremental process for generating better quality plans on average significantly improves the first plans produced (the more time is dedicated to LPG, the better the plan is).

The last two pictures of Figure 4 concern the minimization of plan duration and execution cost, respectively, when both these terms are included in the evaluation function E_H^+. Action durations were determined in the following way: load/unload actions have duration 1; the duration of flying `Airplane1` (the expensive one) is the same as the travel distance; the duration of flying `Airplane2` is twice the travel distance.

Improving plan duration and execution cost simultaneously led to sequences of plans where these criteria do not always decrease monotonically, if considered independently. This is because in some cases less expensive plans might require longer execution. In general, when we have multiple optimization criteria, there is tradeoff among the different criteria to consider. In LPG this tradeoff can be taken into account by appropriately weighting the optimization terms of the evaluation function (in the tests of last 2 plots of Figure 4 μ_C was 0.05 and the weight for the temporal term 0.5).

Conclusions

We have presented a collection of techniques for local search on planning graphs that are implemented in LPG. The basic search scheme of LPG is similar to the stochastic search of Walksat, but has a different search space. While Walksat uses simple general heuristics for SAT problems, LPG uses several techniques to define and evaluate the search neighborhood that exploit the structure of the underlying graph. These include Lagrange multipliers, no-op propagation, action ordering and incremental graph length, and heuristic precondition costs.

In our framework each action can be associated with a number representing its cost. The overall plan cost can be easily modeled by using an appropriate evaluation function, which can be incrementally minimized through an anytime process. Similarly, LPG can produce plans of good quality in terms of the number of parallel steps in the plan, and the overall duration of the plan.

Experimental results in several benchmark domains give an empirical demonstration of the efficiency of our approach, showing, in particular, that LPG can be orders of magnitude faster than other existing planners based on planning graphs. On the other hand, the first plan computed by LPG can be of poor quality. When this is the case, LPG produces additional plans that incrementally improve plan quality in terms of one or more criteria specified in the evaluation function.

Current work include further experiments to test other local search schemes implemented in LPG (Tabuplan and T-Walkplan), and the development of new types of graph modifications and heuristics to guide the search. Finally, very recently an extended version of LPG handling complex PDDL2.1 domains involving numerical quantities and action durations (http://prometeo.ing.unibs.it/lpg) participated in the 3rd International Planning Competition, and it was awarded for "distinguished performance of the first order".[9]

[9] http://www.dur.ac.uk/d.p.long/competition.html

References

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Fox, M., and Long, D. 1998. Efficient implementation of the Plan Graph in STAN. *Journal of Artificial Intelligence research (JAIR)*, vol. 10, 87–115.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI-99).*

Gerevini, A., and Serina, I. 1999. Fast planning through greedy action graphs. In *Proc. of the 16th Nat. Conf. of the Am. Assoc. for Artificial Intelligence (AAAI-99).*

Gerevini, A., and Serina, I. 2001. Lagrange multipliers for local search on planning graphs. In *Proc. of the ECAI-2000 Workshop on Local Search for Planning and Scheduling*. Springer-Verlag.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence research (JAIR)* 14:253–302.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and graph-based planning. In *Proc. of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99).*

Kautz, H.; McAllester, D.; and Selman, B. 1996. Encoding plans in propositional logic. In *Proc. of the 5th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'96).*

Koehler, J.; Nebel, B.; Hoffmann, J.; and Dimopoulos, Y. 1997. Extending planning graphs to an ADL subset. In *Proc. of Fourth European Conference on Planning (ECP'97).* Springer Verlag.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proc. of 39th Design Automation Conference*.

Kambhampati, R.; Minh, S.; and Do, B. 2001. Planning as Constraint Satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence* 132(2):151–182.

Selman, B.; Kautz, H.; and Cohen, B. 1994. Noise strategies for improving local search. In *Proc. of the Twelfth Nat. Conference of the American Association for Artificial Intelligence (AAAI-94)* , 337–343.

Shang, Y., and Wah, B. 1998. A discrete Lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization* 12,1:61–99.

Smith, D., and Weld, D. 1999. Temporal planning with mutual exclusion reasoning. In *Proc. of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99).*

Wah, B. W., and Wu, Z. Oct. 1999. The Theory of Discrete Lagrange Multipliers for Nonlinear Discrete Optimization. In *Proc. of Principles and Practice of Constraint Programming*. Springer-Verlag.

Wu, Z., and Wah, B. 2000. An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In *Proc. of the 17th Nat. Conf. of the American Assoc. for Artificial Intelligence (AAAI-00).*