

A Look-ahead B&B Search for Cost-Based Planning

Raquel Fuentetaja, Daniel Borrajo, and Carlos Linares López

Departamento de Informática, Universidad Carlos III de Madrid
rfuentet@inf.uc3m.es, dborrajo@ia.uc3m.es, clinares@inf.uc3m.es

Abstract This paper focuses on heuristic cost-based planning. We propose a combination of a heuristic designed to deal with this planning model together with the usage of look-ahead states based on relaxed plans to speed-up the search. The search algorithm is a modified Best-First Search (*BFS*) performing Branch and Bound (B&B) to improve the last solution found. The objective of the work is to obtain a good balance between the quality of the plans and the search time. The experiments show that the combination is effective in most of the evaluated domains.

1 Introduction

Planning has achieved significant progress in the last years. This progress is mainly due to powerful heuristic search planners. However, the main efforts have focused on obtaining the solutions faster. In comparison to their predecessors, current planners can solve much more complex problems in considerably less time. These advances have motivated the planning research community to move towards obtaining quality plans maintaining a good balance between quality and speed. In fact, in the last International Planning Competition (IPC-2008) there was a track for STRIPS planning with action costs, in which the evaluation criterion was the quality obtained in a fixed time bound.

Usually, in cost-based planning the quality of plans is inversely proportional to their cost. Most of planners able to deal with cost-based planning have been developed from 2003 until now, as METRIC-FF [6], SIMPLANNER [10], SAPA [4], SGPlan5 [3], and LPG-td(quality) [5]; or the new planners participating in the last competition where LAMA [9] was the winner. Most of these planners use a combination of heuristics together with a search mechanism. One of the problems of applying heuristic search in cost-based planning is that existing heuristics are in general more imprecise than heuristics for classical planning. It is more difficult to estimate costs than estimating number of actions, because the magnitude of the errors the heuristic commits can be much larger, given that costs of actions can be very different. For this reason, the simple combination of a cost-based heuristic with a search algorithm is not enough. Depending on the search algorithm either we can solve few problems or obtain bad quality plans. Some additional technique is needed to help the search algorithm. For example, in the LAMA planner, this technique involves the use of *landmarks* and a combination of heuristics. In this paper, we apply the idea of look-ahead states [11]. The use of look-ahead states reduces significantly the search time in classical planning but has been very little explored in cost-based planning [1]. However, there are multiple ways to implement the idea of look-ahead states: different methods can be defined to compute relaxed plans

and to compute look-ahead states from relaxed plans. Also different search algorithms can be used. In this paper we test one such combinations which provides good results.

This paper is organized as follows: first we define formally the cost-based planning model. Then, we explain the method to boost the search using look-ahead states. This involves to define how look-ahead states are obtained and how they are used in a search algorithm that takes into account cost information. Finally, we include our experimental results and some conclusions.

2 The Cost-Based Planning Model

We consider a cost-based planning problem as a STRIPS planning problem with action costs. A cost-based planning problem is a tuple $(\mathcal{P}, \mathcal{A}, \mathcal{I}, \mathcal{G})$, where \mathcal{P} is a set of propositions, \mathcal{A} is a set of grounded actions (throughout the paper, we will refer to grounded actions simply as actions), and $\mathcal{I} \subseteq \mathcal{P}$ and $\mathcal{G} \subseteq \mathcal{P}$ are the initial state and the set of goals respectively. As in STRIPS planning, each action $a \in \mathcal{A}$ is represented as three sets: $pre(a) \subseteq \mathcal{P}$, $add(a) \subseteq \mathcal{P}$ and $del(a) \subseteq \mathcal{P}$ (preconditions, adds and deletes). Each action also has a fixed cost, $cost(a)$.¹ A plan $\alpha = \{a_1, a_2, \dots, a_n\}$ is an ordered set of grounded actions $a_i \in \mathcal{A}$, that achieves a goal state from the initial state \mathcal{I} ($\mathcal{G} \subseteq apply(a_n, apply(a_{n-1}, \dots apply(a_1, \mathcal{I}) \dots))$)² and whose cost is $cost(\alpha) = \sum_{a_i \in \alpha} cost(a_i)$.

An optimal plan for solving a cost-based planning problem is a plan α^* with the minimum cost. Obtaining optimal plans, as in STRIPS planning, is usually far from being computationally feasible in most medium-size problems. Though there are also optimal planners, usually, cost-based planners combine a non-admissible heuristic with a heuristic search algorithm and obtain suboptimal solutions. In this paper we also propose a suboptimal planner.

3 Using Look-ahead States in Cost-Based Planning

The work reported in [11], applied in the YAHSP planner, shows that the use of look-ahead states based on Relaxed Plans (*RPs*) in classical planning reduces significantly the number of evaluated states (and therefore the search time) of a forward heuristic planner. However, this reduction usually implies an increase in plan length, even using a complete *BFS* algorithm. The success of the approach for reducing the search time, strongly depends on the quality of the relaxed plans (i.e the similarity between the relaxed plan and the real plan). At the same time, the quality of the *RPs* strongly depends on the planning domain. As it was shown by the experiments in [11], *RPs* have less quality in domains where there are many subgoals interactions or with limited resources, though usually these domains are also difficult for the rest of planners. [11] proposes some heuristic considerations in order to improve the quality of the *RPs* and

¹ The conceptual planning model defined does not need numerical state variables, though in the standard language for planning (PDDL), action costs are usually expressed using them.

² The function $apply(a_i, S_j)$ returns the state after applying action a_i in state S_j .

to compute the look-ahead state, considering *RPs* similar to the ones computed by the FF planner [7].

In this paper we wanted to test whether the idea of look-ahead states works in cost-based planning, where the objective is to find *good* quality solutions. An important issue is to obtain relaxed plans sensitive to cost information. Given that relaxed plans are the basis to generate look-ahead states, and we pretend to obtain good quality plans, the relaxed plan should be a good guidance not only to obtain a solution but to obtain a *good* solution. We initially discarded the method of METRIC-FF because the relaxed plan is the same built for the problem without costs. Instead, we have chosen the heuristic explained below, though any other heuristic whose computation produces a relaxed plan sensitive to costs can be applied.

3.1 A Cost-based Heuristic

To compute the heuristic, we build the relaxed planning graph in increasing levels of cost. The algorithm (detailed in Figure 1) receives the state to be evaluated, s , the goal set, \mathcal{G} , and the relaxed domain actions, \mathcal{A}^+ . Then, it follows the philosophy of the Dijkstra algorithm: from all actions whose preconditions have become true in any Relaxed Planning Graph (*RPG*) level before, generate the next action level by applying those actions whose cumulative cost is minimum. The minimum cumulative cost of actions in an action layer i , is associated with the next propositional layer (that contains the effects of those actions). This cost is denoted as $cost_limit_{i+1}$ (initially $cost_limit_0 = 0$). The algorithm maintains an open list of applicable actions (*OpenApp*) not previously applied. At each level i , all new applicable actions are included in *OpenApp*. Each new action in this list includes its cumulative cost, defined as the cost of executing the action, $cost(a)$, plus the cost of supporting the action, i.e. the cost to achieve the preconditions. Here the support cost of an action is defined as the cost limit of the previous propositional layer, $cost_limit_i$, that represents the cost of the most costly precondition (i.e. it is a *max* cost propagation process). Actions with minimum cumulative cost at each iteration are extracted from *OpenApp* and included in the corresponding action level, A_i , i.e. only actions with minimum cumulative cost are executed. The next proposition layer P_{i+1} is defined as usual, including the add effects of actions in A_i . The process finishes with success, returning a Relaxed Planning Graph (*RPG*), when all goals are true in a proposition layer. Otherwise, when *OpenApp* is the empty set, it finishes with failure. In such a case, the heuristic value of the evaluated state is ∞ .

Once we have a *RPG* we extract a *RP* using an algorithm similar to the one applied in METRIC-FF [6]. Applying the same tie breaking policy in the extraction procedure, and unifying some details, the *RPs* we obtain could be obtained with the basic cost-propagation process of SAPA with *max* propagation and ∞ -look-ahead, as described in [2]. However, the algorithm to build the *RPG* in SAPA follows the idea of a breadth-first search with cost propagation instead of Dijkstra. The main difference is that our algorithm guarantees the minimum cost for each proposition at the first propositional level containing the proposition.

Finally, the heuristic value of the evaluated state is computed as the sum of action costs for the actions in the *RP*. Since we generate a relaxed plan, we can use the *helpful actions* applied in [6] to select the most promising successors in the search. Helpful

```

function compute_RPG_hlevel ( $s, \mathcal{G}, \mathcal{A}^+$ )
let  $i = 0; P_0 = s; Open.App = \emptyset; cost.limit_0 = 0;$ 
while  $G \not\subseteq P_i$  do
     $Open.App = Open.App \cup \left\{ a \in \mathcal{A}^+ \setminus \bigcup_{j < i} A_j \mid pre(a) \subseteq P_i \right\}$ 
    forall new action in  $Open.App$  do  $cum.cost(a) = cost(a) + cost.limit_i$ 
     $A_i = \left\{ a \mid a \in \arg \min_{a \in Open.App} cum.cost(a) \right\}$ 
     $cost.limit_{i+1} = \min_{a \in Open.App} cum.cost(a)$ 
     $P_{i+1} = P_i \cup_{a \in A_i} add(a)$ 
    if  $Open.App = \emptyset$  then return fail
     $Open.App = Open.App \setminus A_i$ 
     $i = i + 1$ 
return  $P_0, A_0, P_1, \dots, P_{i-1}, A_{i-1}, P_i$ 

```

Figure 1: Algorithm for building the *RPG*.

actions are the applicable actions in the evaluated state that add at least one proposition required by an action of the relaxed plan and generated by an applicable action in it.

3.2 Considerations for Computing the Look-ahead State

Look-ahead states are obtained by successively applying the actions in the *RP*. There are several heuristic criteria we apply to obtain a good look-ahead state. Some of these considerations have been adopted from Vidal's work in the classical planning case [11], as: (1) we first build the *RP* ignoring all actions deleting top level goals. Relaxed actions have no deletes. So, when an action in the *RP* deletes a top-level goal, probably there will not be another posterior action in the *RP* for generating it. In this case, the heuristic value will be probably a bad estimate; and (2) we generate the look-ahead state using as many actions as possible from the *RP*. This allows to boost the search as much as possible. We do not use other techniques applied in classical planning as the method to replace one *RP* action with another domain action, when no more *RP* actions can be applied [11]. Initially, we wanted to test whether the search algorithm itself is able to repair *RPs* through search.

Determining the *best* order to execute the actions in the *RP* is not an easy task. One can apply the actions in the same order they appear in the *RP*. However, the *RP* comes from a graph built considering that actions are applicable in parallel and they have no deletes. So, following the order in the *RP* can generate plans with bad quality. The reason is that they may have many *useless* actions: actions applied to achieve facts other actions delete. We have the intuition that delaying the application of actions as much as possible (until the moment their effects are required) can alleviate this problem. So, we perform a process of value propagation for actions in the *RP*, their preconditions and effects. The value associated to a proposition represents the level in the *RPG* the proposition is required (we have higher values for propositions required later). The value v_g for each top level goal g is the index of the first level in the *RPG* containing the goal. Then, values are propagated backwards from effects to actions and from actions to preconditions as Figure 2 shows.

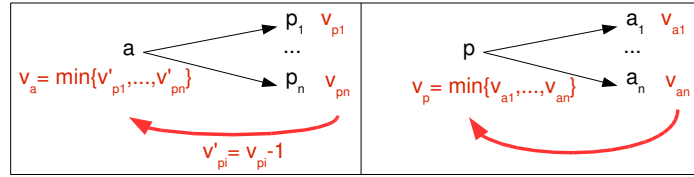


Figure 2: Value propagation process.

Values of actions (v_a) represent the level where their preconditions are required, that is the level where their effects are required minus one. Thus, every effect propagates to the action its level minus one, and then the value for the action is computed as the minimum. Values of propositions (v_p) are the minimum value of the actions where they are preconditions. We use the values v_a of each action in the *RP* in the generation of the look-ahead state for giving priority to actions with lower v_a values, as explained below.

```

function compute_lookahead(node)
let parent = node; new_state = true
if (length(node.RP) < 2 OR length(node.RP) = ∞) return fail
while new_state do
  new_state = false
  current_order = min_v_a(node.RP);
  while current_order ≤ max_v_a(node.RP) do
    forall action = next_not_applied(node.RP, current_order)
      if applicable(parent, action) then
        if not(expensive_repeated_state(parent, action)) then
          parent = apply(parent, action)
          new_state = true
        else
          nrep_states = nrep_states + 1
          if nrep_states < 10 then
            deactivate(action)
            compute_RP_hlevel(parent)
            return compute_lookahead(parent)
          else return fail
      if not new_state then current_order++
if parent ≠ node return parent
else return fail

```

Figure 3: High-level algorithm for computing the look-ahead state.

The Figure 3 shows a high-level algorithm describing the whole process for computing the look-ahead state. In the outer *while* the *current_order* is initialized to the minimum v_a value for the *RP* actions. Then, the algorithm execute an applicable action in the current state (*parent*) with $v_a = \text{current_order}$, updating the current state. This is done for each action with $v_a = \text{current_order}$. If no action with this v_a value can be applied (i.e. no *new_state* is generated), *current_order* is incremented by one. Otherwise, the algorithm continues in the outer *while* and resets newly *current_order* to the minimum v_a value.

In some cases the execution of the actions in the *RP* can lead to a repeated state, *s*. We prune *s* if it has a higher *g-value* than the existing one. Then, instead of taking as current state the parent of the pruned one, we prefer to generate a new relaxed plan from this previous state, but ignoring the last action in the planning graph (i.e. the action that generates the repeated state). However, we do not want to spend much time doing this, so we limit to 10 the number of times a new relaxed plan is built to avoid a repeated state (variable *nrep_states* is initialized to 0 each time the algorithm is called from the search algorithm). This number has been chosen empirically.

3.3 The Search Algorithm

Usually Best-First Search (BFS) algorithms are more adequate for finding near-optimal solutions than local search algorithms as for example Enforced Hill Climbing [7]. Also, as they have memory, they can be easily extended to obtain better solutions systematically, just by continue searching. For this reasons the search algorithm we employ is a *weighted-BFS* with evaluation function $f(n) = g(n) + \omega \cdot h(n)$, modified in the following aspects: first, the algorithm performs a Branch and Bound search. Instead of stopping when the first solution is found, it attempts to improve this solution: the cost of the last solution plan found is used as a bound such that all states whose *g-value* is higher than this cost bound are pruned. As the heuristic is non-admissible we prune by *g-values* to preserve completeness; second, the algorithm uses three lists: the *open* list, and two secondary lists (the *sec-ha* list, and the *sec-non-ha* list). Nodes are evaluated when included in *open*, and each node saves its relaxed plan. As aforementioned, relaxed plans are first built ignoring all actions deleting top-level goals (when this produces an ∞ heuristic value, the node is re-evaluated considering all actions). When a node is extracted from *open*, its look-ahead state is included in the *open* list, its helpful successors are included in the *sec-ha* list, and its non-helpful successors in the *sec-non-ha* list. Each time a new (non repeated) look-ahead state is found the algorithm continues extracting the next node from *open*. Otherwise, if there are nodes in *sec-ha*, they are inserted into *open*. However, if *sec-ha* is empty, all nodes in *sec-non-ha* are included in *open*. This algorithm delays the inclusion in *open* of other successors of the same node distinct of the look-ahead state. Usually, the look-ahead state has a *g-value* higher than the other direct successors, causing a worse *f-value*. However we want to give the opportunity to the search algorithm to expand first the look-ahead state as it can facilitate to find a fast solution, saving heuristic evaluations. Finally, repeated states with higher *g-values* are pruned. The algorithm finishes when *open* is empty. Figure 4 shows a high-level description of the search algorithm (for the sake of simplicity the algorithm does not include the repeated states prune and the *cost_bound* prune for the Branch and Bound).

4 Experimental results

We have implemented the ideas in this paper in a planner based on the code of METRIC-FF, which we call CBP (*Cost-Based Planner*). This planner allows cost-based planning using different heuristics and search algorithms. We have used some numeric domains

```

function BB-LBFS ()
let cost_bound = ∞; plans = ∅; open =  $\mathcal{I}$ ; sec_ha = ∅; sec_non_ha = ∅
while open ≠ ∅ do
  node ← pop_best_node(open)
  if goal_state(node) then /* solution found */
    plans ← plans ∪ {node.plan}
    cost_bound = cost(node.plan)
  else
    lookahead = compute_lookahead(node)
    sec_ha ← sec_ha ∪ helpful_successors(node)
    sec_non_ha ← sec_non_ha ∪ non_helpful_successors(node)
    if lookahead then
      open ← open ∪ {lookahead}
    else if sec_ha then
      open ← open ∪ sec_ha
      sec_ha = ∅
    else
      open ← open ∪ sec_non_ha
      sec_non_ha = ∅

```

Figure 4: High-level BB-LBFS algorithm.

from IPC3 and IPC5, excluding all numerical burden not related to the action costs. For uniformity, we modified the problems of the same domain so that all have the same metric expression. The considered domains are: *Zenotravel*, *Satellite*, *Driverlog*, *Depots* and *Pipesworld*. We also use three of the domains developed by E. Keyder and H. Geffner for planning with action costs [8]: the *Travelling*, *Assignment* and *Minimum Spanning Tree* domains. We contrast the quality of the obtained plans in several time stamps: 5, 25, 125, 625 and 1800 seconds. We use the following configurations in order to determine the adequacy of the application of look-ahead states in cost-based planning: (1) *bbl-bfs* is the configuration explained in the paper with B&B search and look-ahead states, with $\omega = 3$; ³ and (2) *bb-bfs* is the same configuration but without using look-ahead states. Besides, in order to know how competitive the approach is, we compare also with other cost-based planners. We use the planners LPG-td(quality), that is known to perform quite well; and METRIC-FF since it is the base of our code. In four domains we also compare with LAMA. In the other domains, LAMA is not included because the computation of action costs uses arithmetic expressions in the increase effects that LAMA does not support. Since LPG-td(quality) is stochastic we ran the planner five times, plotting the median, as its authors usually do.

Results in the first and last time stamps are shown in Figures 5 and 6. Due to lack of space, other time stamps cannot be shown. The graphics show the percentage of solved problems with plan cost lower than the one indicated by the x -axis.⁴ Only configurations achieving the 100% solve all the problems. From the results, the following general conclusions can be extracted: in very small time bounds, the use of look-ahead states (*bbl-bfs*) allows to solve many more problems than the same search without us-

³ This is the ω used in Vidal's work [11].

⁴ For the sake of clarity some results are presented in logarithmic scale.

ing look-aheads (*bb-bfs*), maintaining the quality or even improving it. The exception is the *Depots* domain, in which *bbl-bfs* performs worse than *bb-bfs* in quality in all time stamps, while the number of solved problems is very similar. As the time progresses, both algorithms solve more problems. In easy domains as *Zenotravel* and *Satellite* good solutions are found quickly, so the available time does not improve them very much. At the last time stamp, in some domains, as in *Zenotravel*, *Satellite* and *Driverlog*, qualities tend to be similar. However *bbl-bfs* solves more problems (in *Driverlog* a 25% more). In the other domains, *bbl-bfs* clearly outperforms *bb-bfs*. The only exception is *Depots*. This domain is characterised by a high level of interacting subgoals. In such types of domains *RPs* have poor quality, and the use a look-ahead strategy based on *RPs* is less adequate. Regarding the performance of the other planners, *bbl-bfs* outperforms LPG in all domains but in *Depots*. In small time bounds it outperforms also LAMA in *Driverlog* and *Assingment*, but at the last time stamps results in *Driverlog* and *Assingment* are very similar, while in *Depots* LAMA behaves better. In *Mst*, LAMA is the best planner in all time stamps. On the other hand, METRIC-FF is clearly outperformed in all time stamps.

5 Conclusions

This paper introduces an approach for dealing with cost-based planning that involves the use of look-ahead states to boost the search, based on a relaxed plan graph heuristic that reasons about costs; and a search algorithm that tries to improve each solution found. The experiments show that the approach is effective for cost-based planning, offering a good trade-off between quality and time in most of the evaluated domains. In domains where there are many interacting subgoals, *RPs* have not good enough quality and the approach is less adequate. The paper also introduces several considerations in order to obtain a good look-ahead state from a given relaxed plan that extends previous work on the application of look-ahead states in classical planning. Also, the method to generate the relaxed plan and the search algorithm we apply differ from that work.

In this paper we have shown the approach is competitive. Future work includes comparing the technique to order the relaxed plan with a vanilla method to test its adequacy, and comparing with other planning algorithms as Vidal's or LAMA.

Acknowledgements

The authors thank E. Keyder and H. Geffner for some of the domains used in this paper. This work has been partially supported by the Spanish MICINN project TIN2008-06701-C03-03 and the UC3M-CAM project CCG08-UC3M/TIC-4141.

References

1. J. Benton, M. van den Briel, and S. Kambhampati. A hybrid linear programming and relaxed plan heuristic for partial satisfaction planning problems. In *Proc. of the 17th ICAPS*, pages 34–41, 2007.
2. D. Bryce and S. Kambhampati. How to skin a planning graph for fun and profit: A tutorial on planning graph based reachability heuristics. *AI Magazine*, 28 No. 1:47–83, 2007.
3. Y. Chen, C. Hsu, and B. Wah. Temporal planning using subgoal partitioning and resolution in SGPlan. *JAIR*, 26:323–369, 2006.

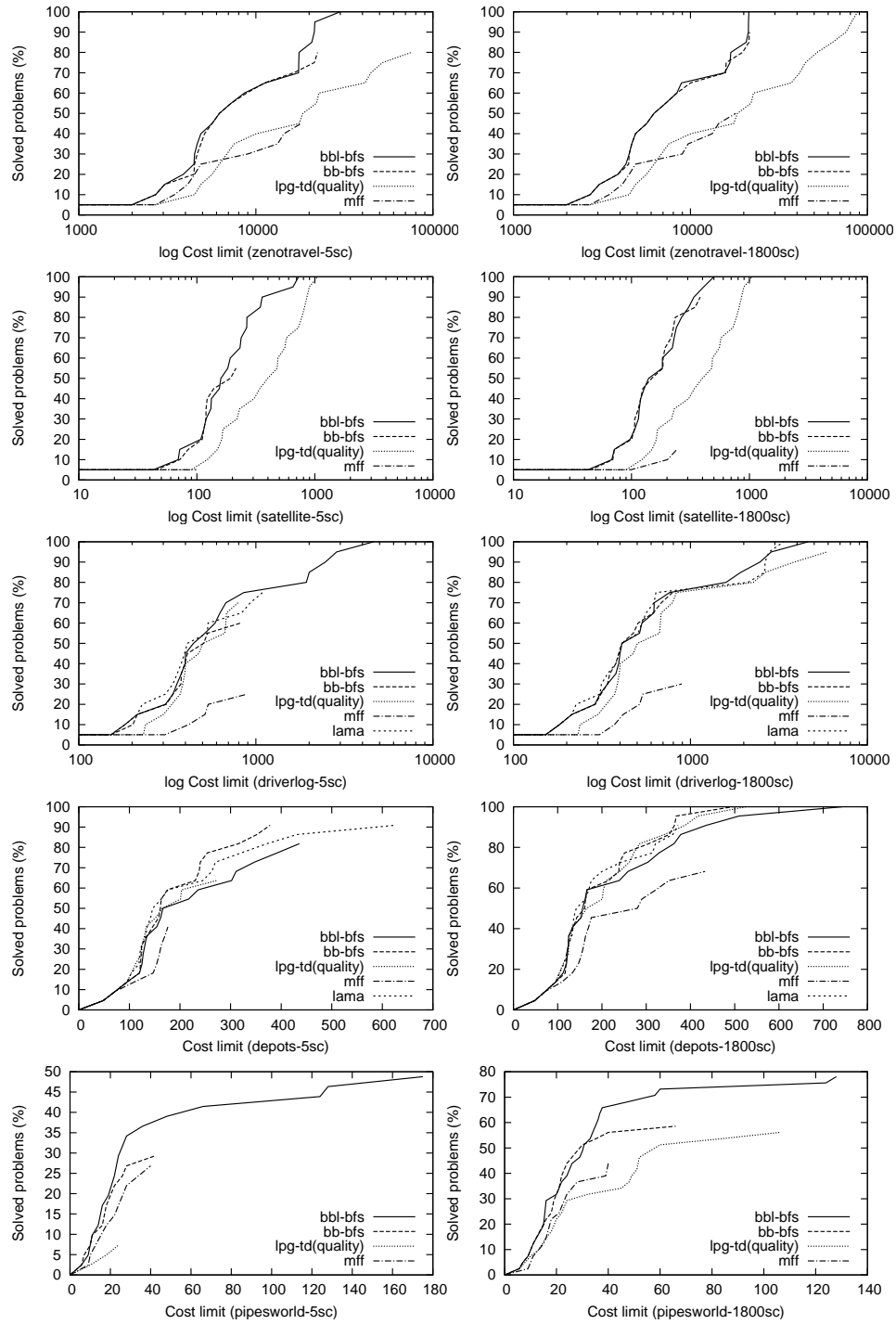


Figure 5: Some of the experimental results (I).

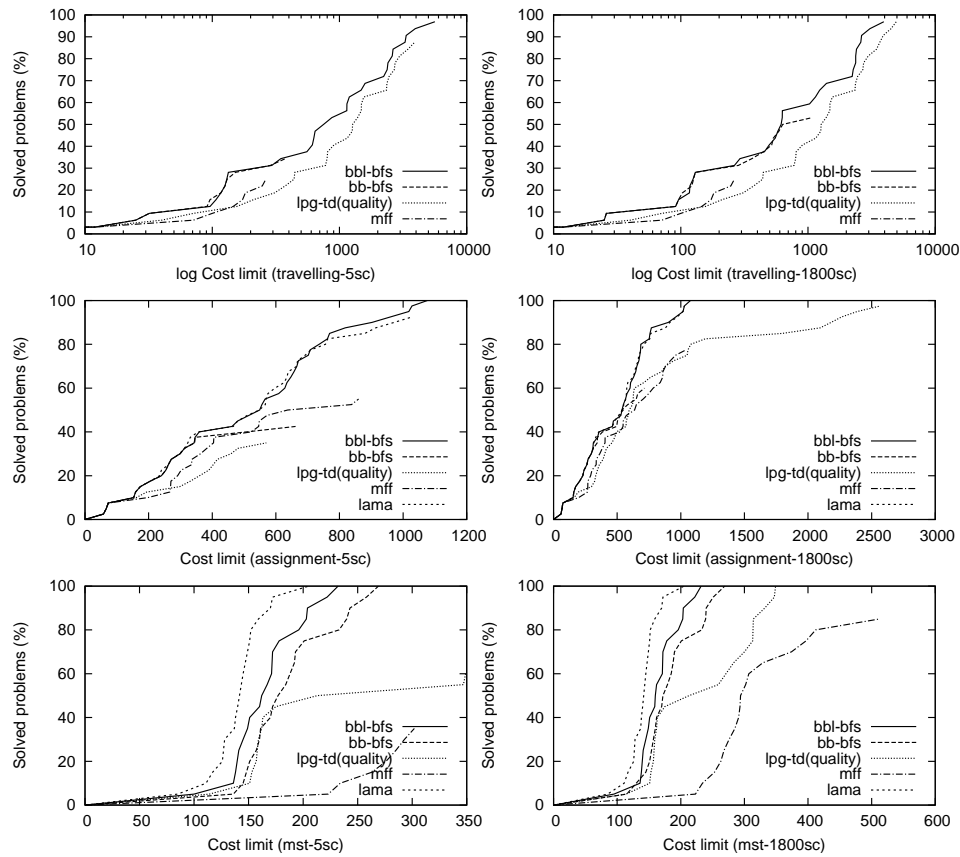


Figure 6: Some of the experimental results (II).

4. M. B. Do and S. Kambhampati. Sapa: A scalable multi-objective heuristic metric temporal planner. *JAIR*, 20:155–194, 2003.
5. A. Gerevini, A. Saetti, and I. Serina. Planning with numerical expressions in LPG. In *Proc. of the 16th ECAI*, pages 667–671, 2004.
6. J. Hoffmann. The Metric-FF planning system: Translating “ignoring delete lists” to numeric state variables. *JAIR*, 20:291–341, 2003.
7. J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.
8. E. Keyder and H. Geffner. Heuristics for planning with action costs revisited. In *Proc. of the 18th ECAI*, 2008.
9. S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In *Proc. of the 23rd AAAI*, pages 975–982, 2008.
10. O. Sapena and E. Onaindía. Handling numeric criteria in relaxed planning graphs. In *Advances in Artificial Intelligence. IBERAMIA, LNAI 3315*, pages 114–123, 2004.
11. V. Vidal. A lookahead strategy for heuristic search planning. In *Proc. of the 14th ICAPS*, pages 150–159, 2004.