The CBP planner

Raquel Fuentetaja
Departamento de Informática, Universidad Carlos III de Madrid
rfuentet@inf.uc3m.es

Abstract

This short paper provides a high-level description of the planner CBP (Cost-Based Planner). CBP performs heuristic search in the state space using several heuristics. On one hand it uses look-ahead states based on relaxed plans to speed-up the search; on the other hand the search is also guided using a numerical heuristic and a selection of actions extracted from a relaxed planning graph. The relaxed planning graph is built taking into account action costs. The search algorithm is a modified Best-First Search (*BFS*) performing Branch and Bound (B&B) to improve the last solution found.

Introduction

Usually, in cost-based planning the quality of plans is inversely proportional to their cost. Many planners that have been developed for being able to deal with cost-based planning use a combination of heuristics together with a search mechanism, as METRIC-FF (Hoffmann 2003), SIMPLANNER (Sapena & Onaindía 2004), SAPA (Do & Kambhampati 2003), SGPlan5 (Chen, Hsu, & Wah 2006), LPG-td(quality) (Gerevini, Saetti, & Serina 2004) or LAMA (Richter, Helmert, & Westphal 2008).

One of the problems of applying heuristic search in cost-based planning is that existing numerical heuristics are in general more imprecise than heuristics for classical planning. The magnitude of the error the heuristic commits can be much larger since costs of actions can be very different. Therefore, making numerical estimations using the cost of actions which are not part of the actual optimal solution can lead to a great difference between the actual optimal value and the estimation. For this reason, some additional technique, apart from a numerical heuristic, is needed to help the search algorithm. In CBP we combine the use of a numerical heuristic with the idea of look-ahead states (Vidal 2004), both extracted from a relaxed planning graph aware of cost information. 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 CBP we test one such combinations. CBP has been described before in (Fuentetaja, Borrajo, & Linares 2009).

The Numerical Heuristic in CBP

To compute the numerical heuristic, we build a 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 (Hoffmann 2003). 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 (Bryce & Kambhampati 2007). 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.

function compute_RPG_hlevel $(s, \mathcal{G}, \mathcal{A}^+)$
let $i = 0$; $P_0 = s$; $OpenApp = \emptyset$; $cost\_limit_0 = 0$;
while $G \not\subseteq P_i$ **do**
$OpenApp = OpenApp \cup \left\{ a \in \mathcal{A}^+ \setminus \bigcup_{j<i} A_j \mid pre(a) \subseteq P_i \right\}$
forall new action in $OpenApp$ **do**
$cum\_cost(a) = cost(a) + cost\_limit_i$
$A_i = \left\{ a \mid a \in \arg\min_{a \in OpenApp} cum\_cost(a) \right\}$
$cost\_limit_{i+1} = \min_{a \in OpenApp} cum\_cost(a)$
$P_{i+1} = P_i \bigcup_{a \in A_i} add(a)$
if $OpenApp = \emptyset$ **then return** fail
$OpenApp = OpenApp \setminus A_i$
$i = i + 1$
return $P_0, A_0, P_1, ..., P_{i-1}, A_{i-1}, P_i$

Figure 1: Algorithm for building the *RPG*.

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 (Hoffmann 2003) to select the most promising successors in the search. Helpful 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.

For the 2011 IPC we present two versions of the planner. The first one computes the *RPG* as have been described before in this section. The second one replaces the *max* cost propagation process with an additive propagation process, which involves to compute the $cum\_cost(a)$ in the line 5 of the algorithm in Figure 1 as:

$$cum\_cost(a) = cost(a) + \sum_{q \in pre(a)} cost\_limit_{level(q)}$$

where $level(q)$ is the index of the first propositional layer containing q. Applying the same tie breaking policy in the extraction procedure this additive version will be equivalent to the cost-propagation process of SAPA with *additive* propagation and ∞-look-ahead, as described in (Bryce & Kambhampati 2007) and also to the heuristic h_a (Keyder & Geffner 2008).

Computing Look-ahead States

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 (Vidal 2004), 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 (Vidal 2004). 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 follow a heuristic procedure to give priority to the actions of the *RP* whose effects are required before (they are subgoals in lower layers).[1] To do this, during the extraction process, each action selected to be in the *RP* is assigned a value we call *level-required*. In the general case, this value is exactly the minimum layer index minus one of all the selected actions for which the subgoal is a precondition.

The *RPGs* built propagating costs differs from the *RPGs* built for classical planning. One of the differences is that in the former we can have a sequence of actions to achieve a fact that in the classical case can be generated using just one action. The reason is that the total cost of applying the sequence of actions is lower than the cost of using only one action. In such a case, all the intermediate facts in the sequence are not really necessary for the task. They only appear for cost-related reasons. The only necessary fact is the last one of the sequence. In this case, the way of computing the *level-required* differs from the general case: we assign the same *level-required* to all actions of the sequence, that is the *level-required* of the action that generate the last (and necessary) fact. The justification of this decision is that once the actions have been selected to be in the *RP* we want to execute as more actions as possible of the *RP*, so we have to attend to causality reasons and not to cost-related reasons.

The *level-required* values provide us a partial order for the actions in the *RP*. We generate the look-ahead state executing first the actions with lower *level-required*. In case of ties we follow the order of the *RP*.

The Figure 2 shows a high-level algorithm describing the whole process for computing the look-ahead state given the source state and the relaxed plan (*RP*). The variables min_order and max_order represent the minimum and maximum *level-required* for the actions in the *RP* respectively. Initially, all the actions in the *RP* are marked as no executed, and the variable *current_order* is set to the minimum level required. The `for` sentence covers the relaxed plan executing the actions applicable in the current state and not yet executed, whose level required equals the current order. The execution of one action implies to update the current state and to initialize the current order to the minimum level required. After covering the relaxed plan, if no new ac-

[1] The heuristic procedure described here slightly differs from the one described in (Fuentetaja, Borrajo, & Linares 2009).

```
function obtain_lookahead_state(state, RP)
let min_order = minimum_level_required(RP)
let max_order = maximum_level_required(RP)
let executed[a] = FALSE, ∀a ∈ RP
let current_order = min_order
let current_state = state
while current_order <= max_orden do
    new_action_applied = FALSE
    forall a ∈ RP do
        if not executed[a] ∧ applicable(a, current_state)∧
          level_required(a) = current_order then
            new_state = apply(a, current_state)
            executed[a] = TRUE
            new_action_applied = TRUE
            current_state = new_state
            current_order = min_ordern
    if not new_action_applied then
        current_order = current_order + 1
if current_state ≠ state
    return current_state
else
    return fail
```

Figure 2: High-level algorithm for computing the look-ahead state.

tion has been executed, the current order is increased by one. This process is repeated until the current order is higher than the maximum level required (`while` sentence). Finally the algorithm returns the last state achieved (i.e. the look-ahead state).

The Search Algorithm

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 two lists: the *open* list, and the secondary list (the *sec-non-ha* list). Nodes are evaluated when included in *open*, and each node saves its relaxed plan. 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*, all look-ahead states that can be generated successively starting from the node are included in *open*. Then, its helpful successors are also included in *open*, and its non-helpful successors in the *sec-non-ha* list. When after doing this, the *open* list is empty, all nodes in *sec-non-ha* are included in *open*. The algorithm finishes when *open* is empty. Figure 3 shows a high-level description of the search algorithm (BB-LBFS, Branch and Bound Look-ahead Best First Search). For the sake of simplicity the algorithm does not include the repeated states prune and the *cost_bound* prune for the Branch and Bound. Repeated states prune is performed pruning states with the same facts and higher *g-values*.

```
function BB-LBFS ()
let cost_bound = ∞; plans = ∅; open = I; 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)
        while lookahead do
            open ← open ∪ {lookahead}
            if goal_state(lookahead) then
                plans ← plans ∪ {lookahead.plan}
                cost_bound = cost(lookahead.plan)
                lookahead = compute_lookahead(lookahead)
        open ← open ∪ helpful_successors(node)
        sec_non_ha ← sec_non_ha ∪ non_helpful_successors(node)
        if open = ∅ do
            open ← open ∪ sec_non_ha
            sec_non_ha = ∅
return plans
```

Figure 3: High-level BB-LBFS algorithm.

Summary

This paper described the main parts of the CBP planner: (1) the computation of the relaxed planning graph to obtain a numerical heuristic and the *helpful actions*; (2) the generation of lookahead states based on the same relaxed planning graphs, and (3) the search algorithm. As aforementioned, the relaxed planning graphs are built making use of action costs. The search algorithm is a best first algorithm modified to give priority to *helpful actions* and to include it look-ahead states. Instead of stopping at first solution it continues the search performing Branch and Bound until a time bound is reached. For the competition we have set the ω value in the evaluation function to three. CBP has been implemented in C, reusing part of the code of METRIC-FF.

References

Bryce, D., and Kambhampati, S. 2007. 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.

Chen, Y.; Hsu, C.; and Wah, B. 2006. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research (JAIR)* 26:323–369.

Do, M. B., and Kambhampati, S. 2003. Sapa: A scalable multi-objective heuristic metric temporal planner. *Journal of Artificial Intelligence Research (JAIR)* 20:155–194.

Fuentetaja, R.; Borrajo, D.; and Linares, C. 2009. A look-ahead B&B search for cost-based planning. In *Proceedings of the 13th Conference of the Spanish Association for Artificial Intelligence (CAEPIA)*, 105–114.

Gerevini, A.; Saetti, A.; and Serina, I. 2004. Planning with numerical expressions in LPG. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, 667–671.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state vari-

ables. *Journal of Artificial Intelligence Research (JAIR)* 20:291–341.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI)*.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the 23rd American Association for the Advancement of Artificial Intelligence Conference (AAAI)*, 975–982.

Sapena, O., and Onaindía, E. 2004. Handling numeric criteria in relaxed planning graphs. In *Advances in Artificial Intelligence. IBERAMIA, LNAI 3315*, 114–123.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS)*, 150–159.