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 & Onaindia 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, $G$, and the relaxed domain actions, $A^+$. 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$+1$ (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$, 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 $P_i$ 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 $\infty$.

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 $\infty$-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.
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 propositional layer 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 EPA with additive propagation and $\infty$-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 build 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, Borrjaio, & Linares 2009).

---

**Figure 1**: Algorithm for building the RPG.

```python
function compute_RPG_level (s, G, A*)
let i = 0; P_0 = s; OpenApp = \emptyset; cost_limit_0 = 0;
while G \notin P_i do
    OpenApp = OpenApp \cup \{ a \in A^* \setminus \bigcup_{j<i} A_j \mid pre(a) \subseteq P_i \}
    forall new action in OpenApp do
        cum_cost(a) = cost(a) + cost_limit_i
        A_i = \{ a \in arg min_{a \in OpenApp} cum_cost(a) \}
        cost_limit_{i+1} = min_{a \in OpenApp} cum_cost(a)
        P_{i+1} = P_i \cup 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
```

```
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_order 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 = current_order + 1
      if not new_action_applied then
        current_order = current_order + 1
      if current_state ≠ state
        return current_state
      else
        return fail
  end
end

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 = ∅
        open ← open ∪ sec_non_ha
      sec_non_ha = ∅
    end
  end
end
return plans

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

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 lookahead 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

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


