The YAHSP planning system: Forward heuristic search with lookahead plans analysis

Vincent Vidal
CRIL - Université d'Artois
rue de l'Université - SP 16
62307 Lens, France
vidal@cril.univ-artois.fr

Introduction

Planning as heuristic search has proven to be a successful framework for STRIPS non-optimal planning, since the advent of planners capable to outperform in most of the classical benchmarks the previous state-of-the-art planners Graphplan (Blum & Furst 1997), Blackbox (Kautz & Selman 1999), IPP (Koehler *et al.* 1997), STAN (Long & Fox 1999), LCGP (Cayrol, Régnier, & Vidal 2001), ... Although these planners (except LCGP) compute optimal parallel plans, which is not exactly the same purpose as non-optimal planning, they also offer no optimality guarantee concerning plan length in number of actions.

The planning as heuristic search framework indeed lead to some of the most efficient planners, as demonstrated in the two previous editions of the International Planning Competition with planners such as HSP2 (Bonet & Geffner 2001), FF (Hoffmann & Nebel 2001) and AltAlt (Nguyen, Kambhampati, & Nigenda 2002). FF was in particular awarded for outstanding performance at the 2^{nd} International Planning Competition and was generally the top performer planner in the STRIPS track of the 3^{rd} International Planning Competition.

The YAHSP planning system ("Yet Another Heuristic Search Planner", more details in (Vidal 2004)) extends a technique introduced in the FF planning system (Hoffmann & Nebel 2001) for calculating the heuristic, based on the extraction of a solution from a planning graph computed for the relaxed problem obtained by ignoring deletes of actions. It can be performed in polynomial time and space, and the length in number of actions of the relaxed plan extracted from the planning graph represents the heuristic value of the evaluated state. This heuristic is used in a forward-chaining search algorithm to evaluate each encountered state.

We introduce a novel way for extracting information from the computation of the heuristic, by considering the high quality of the relaxed plans extracted by the heuristic function in numerous domains. Indeed, the beginning of these plans can often be extended to solution plans of the initial problem, and there are often a lot of other actions from these plans that can effectively be used in a solution plan. YAHSP uses an algorithm for combining some actions from each relaxed plan, in order to find the beginning of a valid plan that can lead to a reachable state. Thanks to the quality of the extracted relaxed plans, these states will frequently bring us closer to a solution state. The lookahead states thus calculated are then added to the list of nodes that can be chosen to be expanded by increasing order of the numerical value of the heuristic. The best strategy we (empirically) found is to use as much actions as possible from each relaxed plan and to perform the computation of lookahead states as often as possible.

This lookahead strategy can be used in different search algorithms. We propose a modification of a classical best-first search algorithm in a way that preserves completeness. Indeed, it simply consists in augmenting the list of nodes to be expanded (the open list) with some new nodes computed by the lookahead algorithm. The branching factor is slightly increased, but the performances are generally better and completeness is not affected.

Our experimental evaluation of the use of this lookahead strategy in a complete best-first search algorithm demonstrates that in numerous planning benchmark domains, the improvement of the performance in terms of running time and size of problems that can be handled have been drastically improved (cf. (Vidal 2004)).

Computing and using lookahead states and plans

A *state* is a finite set of ground atomic formulas (i.e. without any variable symbol) also called *fluents*. *Actions* are classical STRIPS actions. Let a be an action; $Prec(a)$, $Add(a)$ and $Del(a)$ are fluent sets and respectively denote the preconditions, add effects, and del effects of a. A *planning problem* is a triple $\langle O, I, G\rangle$ where O is a set of actions, I is a set of fluents denoting the initial state and G is a set of fluents denoting the goal. A *plan* is a sequence of actions. The *application* of an action a on a state S (noted $S \uparrow a$) is possible if $Prec(a) \subseteq S$ and the resulting state is defined by $S \uparrow a = (S \setminus Del(a)) \cup Add(a)$. Let $P = \langle a_1, a_2, \ldots, a_n\rangle$ be a plan. P is *valid* for a state S if a_1 is applicable on S and leads to a state S_1, a_2 is applicable on S_1 and leads to S_2, ..., a_n is applicable on S_{n-1} and leads to S_n. In that case, S_n is said to be *reachable* from S for P and P is a *solution plan* if $G \subseteq S_n$. $First(P)$ and $Rest(P)$ respec-

tively denote the first action of P (a_1 here) and P without the first action ($\langle a_2, \ldots, a_n \rangle$ here). Let $P' = \langle b_1, \ldots, b_m \rangle$ be another plan. The *concatenation* of P and P' (denoted by $P \oplus P'$) is defined by $P \oplus P' = \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$.

Principle and use of lookahead plans

In classical forward state-space search algorithms, a node in the search graph represents a planning state and an arc starting from that node represents the application of one action to this state, that leads to a new state. In order to ensure completeness, all actions that can be applied to one state must be considered. The order in which these states will then be considered for development depends on the overall search strategy: depth-first, breadth-first, best-first...

Let us now imagine that for each evaluated state S, we knew a valid plan P that could be applied to S and would lead to a state closer to the goal than the direct descendants of S (or estimated as such, thanks to some heuristic evaluation). It could then be interesting to apply P to S, and use the resulting state S' as a new node in the search. This state could be simply considered as a new descendant of S.

We have then two kinds of arcs in the search graph: the ones that come from the direct application of an action to a state, and the ones that come from the application of a valid plan to a state S and lead to a state S' reachable from S. We will call such states *lookahead states*, as they are computed by the application of a plan to a node S but are considered in the search tree as direct descendants of S. Nodes created for lookahead states will be called *lookahead nodes*. Plans labeling arcs that lead to lookahead nodes will be called *lookahead plans*. Once a goal state is found, the solution plan is then the concatenation of single actions for arcs leading to classical nodes and lookahead plans for the arcs leading to lookahead nodes.

Completeness and correctness of search algorithms are preserved by this process, because no information is lost: all actions that can be applied to a state are still considered, and because the nodes that are added by lookahead plans are reachable from the states they are connected to. The only modification is the addition of new nodes, corresponding to states that can be reached from the initial state.

Computing relaxed plans

The determination of an heuristic value for each state as performed in the FF planner offers a way to compute such lookahead plans. FF creates a planning graph for each encountered state S, using the relaxed problem obtained by ignoring deletes of actions and using S as initial state. A relaxed plan is then extracted in polynomial time and space from this planning graph. The length in number of actions of the relaxed plan corresponds to the heuristic evaluation of the state for which it is calculated. Generally, the relaxed plan for a state S is not valid for S, as deletes of actions are ignored during its computation: negative interactions between actions are not considered, so an action can delete a goal or a fluent needed as a precondition by some actions that follow it in the relaxed plan. But actions of the relaxed plans are used because they produce fluents that can be interesting to obtain the goals, so some actions of these plans can possibly be interesting to compute the solution plan of the problem. In numerous benchmark domains, we can observe that relaxed plans have a very good quality because they contain a lot of actions that belong to solution plans.

The computation of relaxed plans in YAHSP works closely as in FF, with one notable difference which holds in the way actions are added to the relaxed plan. In FF, actions are arranged in the order they get selected. We found useful to use the following algorithm. Let a be an action, and $\langle a_1, a_2, \ldots, a_n \rangle$ be a relaxed plan. All actions in the relaxed plan are chosen in order to produce a subgoal in the relaxed planning graph at a given level, which is either a problem goal or a precondition of an action of the relaxed plan. a is ordered after a_1 iff:

- the level of the subgoal a was selected to satisfy is strictly greater than the level of the subgoal a_1 was selected to satisfy, or
- these levels are equal, and either a deletes a precondition of a_1 or a_1 does not delete a precondition of a.

In that case, the same process continues between a and a_2, and so on with all actions in the plan. Otherwise, a is placed before a_1.

Computing lookahead plans

The algorithm for computing lookahead plans (cf. Figure 1) takes as input the current planning state S, and the relaxed plan RP that has been computed by the heuristic function. Several strategies can be imagined: searching plans with a limited number of actions, returning several possible plans, etc. From our experiments, the best strategy we found is to search one plan, containing as most actions as possible from the relaxed plan. One improvement we made to that process is the following. When no action of RP can be applied, we replace one of its action a by an action a' taken from the global set of actions O, such that a':

- does not belong to RP,
- is applicable in the current lookahead state S',
- produces at least one add effect f of a such that f is a precondition of another action in RP and f does not belong to S'.

At first, we enter in a loop that stops if no action can be found or all actions of RP have been used. Inside this loop, there are two parts: one for selecting actions from RP, and another one for replacing an action of RP by another action in case of failure in the first part.

In the first part, actions of RP are observed in turn, in the order they are present in the sequence. Each time an action a is applicable in S, we add a to the end of the lookahead plan and update S by applying a to it (removing deletes of a and adding its add effects). Actions that cannot be applied are kept in a new relaxed plan called $failed$ in the order they get selected. If at least one action has been found to be applicable, when all actions of RP have been tried, the second part is not used (this is controlled by the boolean $continue$). The relaxed plan RP is overwritten with $failed$ and the process is repeated until RP is empty or no action can be found.

```
function lookahead (S, RP)    /* S: state, RP: relaxed plan */
  let plan = ⟨⟩ ;
  let failed = ⟨⟩ ;
  let continue = true ;
  while continue ∧ RP ≠ ⟨⟩ do
    continue ← false ;
    forall i ∈ [1, n] do    /* with RP = ⟨a_1, ..., a_n⟩ */
      if Prec(a_i) ⊆ S then
        continue ← true ;
        S ← S ↑ a_i ;
        plan ← plan ⊕ ⟨a_i⟩
      else
        failed ← failed ⊕ ⟨a_i⟩
      endif
    endfor ;
    if continue then
      RP ← failed ;
      failed ← ⟨⟩
    else
      RP ← ⟨⟩ ;
      while ¬continue ∧ failed ≠ ⟨⟩ do
        forall f ∈ Add(First(failed)) do
          if f ∉ S ∧ ∃a ∈ (RP ⊕ failed) | f ∈ Prec(a) then
            let actions =
                {a ∈ O | f ∈ Add(a) ∧ Prec(a) ⊆ S} ;
            if actions ≠ ∅ then
              let a = choose_best(actions) ;
              continue ← true ;
              S ← S ↑ a ;
              plan ← plan ⊕ ⟨a⟩ ;
              RP ← RP ⊕ Rest(failed) ;
              failed ← ⟨⟩
            endif
          endif
        endfor ;
        if ¬continue then
          RP ← RP ⊕ ⟨First(failed)⟩ ;
          failed ← Rest(failed)
        endif
      endwhile
    endif
  endwhile
  return(S, plan)
end
```

Figure 1: Lookahead algorithm

The second part is entered when no action has been applied in the most recent iteration of the first part. The goal is to try to repair the current (not applicable) relaxed plan, by replacing one action by another which is applicable in the current state S. Actions of *failed* are observed in turn, and we look for an action (in the global set of actions O) applicable in S, which achieves an add effect of the action of *failed* we observe, this add effect being a precondition not satisfied in S of another action in the current relaxed plan. If several achievers are possible for the add effect of the action of *failed* we observe, we select the one that has the minimum cost in the relaxed planning graph used for extracting the initial relaxed plan (the cost of an action is the sum of the initial levels of its preconditions). When such an action is found, it is added to the lookahead plan and the global loop is repeated. The action of *failed* observed when a repairing action was found is not kept in the current relaxed plan.

Conclusion

We presented a new method for deriving information from relaxed plans, by the computation of lookahead plans. They are used in a complete best-first search algorithm for computing new nodes that can bring closer to a solution state. Although lookahead states are generally not goal states and the branching factor is increased with each created lookahead state, the experiments we conducted prove that in numerous domains from previous competitions (Rovers, Logistics, DriverLog, ZenoTravel, Satellite), our planner can solve problems that are up to ten times bigger (in number of actions of the initial state) than those solved by FF or by a classical best-first search without lookahead.YAHSP seems also to present good performances in domains from the 4^{th} IPC, such as Pipesworld, Satellite and Promela/Philosophers where it solves all the problems, or Psr and Promela/Optical-Telegraph were a very few problems are not solved. The domain which seems to be the more difficult for YAHSP is Airport, where 12 problems are not solved yet.The counterpart for such improvements in performances and size of the problems that can be handled resides in the quality of solution plans that can be in some cases degraded (generally in domains where there are a lot of subgoal interactions). However, there are few of such plans and quality remains generally very good compared to FF.

References

Blum, A., and Furst, M. 1997. Fast planning through planning-graphs analysis. *Artificial Intelligence* 90(1-2):281–300.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Cayrol, M.; Régnier, P.; and Vidal, V. 2001. Least commitment in Graphplan. *Artificial Intelligence* 130(1):85–118.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and Graph-based planning. In *Proc. IJCAI-99*, 318–325.

Koehler, J.; Nebel, B.; Hoffmann, J.; and Dimopoulos, Y. 1997. Extending planning-graphs to an ADL subset. In *Proc. ECP-97*, 273–285.

Long, D., and Fox, M. 1999. The efficient implementation of the plan-graph in STAN. *JAIR* 10:87–115.

Nguyen, X.; Kambhampati, S.; and Nigenda, R. 2002. Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *Artificial Intelligence* 135(1-2):73–123.

Vidal, V. 2004. A Lookahead Strategy for Heuristic Search Planning. In *Proc. ICAPS-2004*.