A Lookahead Strategy for Heuristic Search Planning

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

Abstract

Relaxed plans are used in the heuristic search planner FF for computing a numerical heuristic and extracting helpful actions. We present a novel way for extracting information from the relaxed plan and for dealing with helpful actions, by considering the high quality of the relaxed plans in numerous domains. For each evaluated state, we employ actions from these plans in order to find the beginning of a valid plan that can lead to a reachable state. We use this lookahead strategy in a complete best-first search algorithm, modified in order to take into account helpful actions. In numerous planning domains, the performance of heuristic search planning and the size of the problems that can be handled have been drastically improved.

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. This is one reason for which the interest of the planning community turned towards the planning as heuristic search framework and other techniques, promising in terms of performance for non-optimal planning plus some other advantages such as easier extensions to resource planning and planning under uncertainty.

The planning as heuristic search framework, initiated by the planners ASP (Bonet, Loerincs, & Geffner 1997), HSP and HSPr (Bonet & Geffner 1999; 2000), 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[1] and was generally the top performer planner in the STRIPS track of the 3[rd] International Planning Competition[2].

We focus in this paper[3] on 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. As a side effect of the computation of this heuristic, another information is derived in FF from the planning graph and its solution, namely the *helpful actions*. They are the actions of the relaxed plan executable in the state for which the heuristic is computed, augmented in FF by all the actions which are executable in that state and produce fluents that where found to be subgoals at the first level of the planning graph. These actions permit FF to concentrate its efforts on more promising ways than considering all actions, forgetting actions that are not helpful in a variation of the hill-climbing local search algorithm. When this last fails to find a solution, FF switches to a classical complete best-first search algorithm. The search is then started again from scratch, without the benefit obtained by using helpful actions and local search.

We introduce a novel way for extracting information from the computation of the heuristic and for dealing with helpful actions, 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. We present in this paper an algorithm for combining some actions from each relaxed plan,

*This work has been supported in part by the IUT de Lens, the CNRS and the Region Nord/Pas-de-Calais under the TACT Programme.

[1] 2[nd] IPC: http://www.cs.toronto.edu/aips2000
[2] 3[rd] IPC: http://www.cis.strath.ac.uk/ derek/competition.html
[3] An extended abstract appeared in (Vidal 2003).

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. In addition to this lookahead strategy, we propose a new way for using helpful actions that also preserves completeness. In FF, actions that are not considered as helpful are lost: this makes the algorithm incomplete. For avoiding that, we modify several aspects of the search algorithm by introducing the notion of *rescue actions* (actions applicable in the state for which we compute the relaxed plan and are not helpful). States will be first expanded with helpful actions as in FF; but in case of failure (i.e. no solution is found by using only helpful actions), we choose a node that can be expanded with rescue actions. As no action is lost and no node is pruned from the search space as in FF, completeness is preserved.

Our experimental evaluation of the use of this lookahead strategy in a complete best-first search algorithm that takes benefit of helpful actions 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. Taking into account helpful actions makes a best-first search algorithm always more efficient, while the lookahead strategy makes it able to solve very large problems in several domains.

This paper is organized as follows. After giving classical definitions, we present the computation and use of lookahead states and plans. We then explain how to use helpful actions in a complete best-first search algorithm. We finally detail the algorithms implemented in our system, and provide an experimental evaluation before some conclusions.

Definitions

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 $G \subseteq S_n$. $First(P)$, $Rest(P)$ and $Length(P)$ respectively denote the first action of P (a_1 here), P without the first action ($\langle a_2, \ldots, a_n\rangle$ here), and the number of actions in P (n 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 $P \oplus P' = \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$.

Computing and using lookahead states and 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.

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. We will describe an algorithm for computing lookahead plans from these relaxed plans.

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.

Using helpful actions: the "optimistic" strategy for search algorithms

In classical search algorithms, all actions that can be applied to a node are considered the same way: the states that they lead to are evaluated by an heuristic function and are then ordered, but there is no notion of preference over the actions themselves. Such a notion of preference during search has been introduced in the FF planner, with the concept of helpful actions. Once the planning graph for a state S is created and a relaxed plan is computed, more information is extracted from these two structures: the actions of the relaxed plan that are executable in S are considered as *helpful*, while the other actions are forgotten by the local search algorithm of FF. But this strategy appeared to be too restrictive, so the set of helpful actions is augmented in FF by all actions executable in S that produce fluents that were considered as subgoals at the first level of the planning graph, during the extraction of the relaxed plan[4]. The main drawback of this strategy, as used in FF, is that it does not preserve completeness: the actions executable in a state S that are not considered as helpful are simply lost. In FF, the search algorithm is not complete for other reasons (it is a variation of an hill-climbing algorithm), and the search must switch to a complete best-first algorithm when no solution is found by the local search algorithm.

In this paper, we present a way to use such a notion of helpful actions in complete search algorithms, that we call *optimistic search algorithms* because they give a maximum trust to the information returned by the computation of the heuristic. The principles are the following:

- Several classes of actions are created. In our implementation, we only use two of them: *helpful actions* (the restricted ones: the actions of the relaxed plan that are executable in the state for which we compute a relaxed plan), and *rescue actions* that are all the actions that are not helpful.
- When a newly created state S is evaluated, the heuristic function returns the numerical estimation of the state and also the actions executable in S partitioned into their different classes (for us, helpful or rescue[5]). For each class, one node is created for the state S, that contains the actions of that class returned by the heuristic function.
- In order to select a node to be expanded, the class of the actions attached to it is taken into account: in our "optimistic" algorithm, it is the first criterion: nodes containing helpful actions are always preferred over nodes containing rescue actions, whatever their numerical heuristic values are.

No information is lost by this process. The way nodes are expanded is simply modified: a state S is expanded first with helpful actions, some other nodes are expanded, and then S can potentially be expanded with rescue actions. As the union of helpful actions and rescue actions is equal to the set of all the actions that can be applied to S, completeness and correctness are preserved.

In addition to that use of helpful actions, we use in our planning system another "optimistic" strategy. Let us first define *goal-preferred actions* as the actions that do not delete a fluent which belongs to the goal and do not belong to the initial state. If giving a maximum trust to the heuristic (and being in that sense "optimistic"), we can suppose that a subgoal which has been previously achieved does not need to be undone for solving the whole problem (with the except of goals satisfied in the initial state). So, for each evaluated state, we build a first relaxed planning graph with goal-preferred actions only, and in case of failure (the goals are not present in the relaxed planning graph), we build another planning graph with all the actions of the problem.

Description of the algorithms

In this section, we describe the algorithms of our planning system and discuss the main differences with the FF planner, on which is based our work. The main functions of the algorithm are the following:

LOBFS(): this is the main function (see Figure 1). At first, the function **compute_node** is called over the initial state of the problem and the empty plan: the initial state is evaluated by the heuristic function, and a node is created and pushed into the open list. Nodes in the open list have the following structure: $\langle S, P, A, h, f\rangle$, where:

- S is a state,
- P is the plan that lead to S from the initial state,
- A is a set of actions applicable in S,
- h is the heuristic value of S (the length of the relaxed plan computed for S),
- f is a flag indicating if the actions of A are helpful actions (value $helpful$) or rescue actions (value $rescue$).

We then enter in a loop that selects the best node (function **pop_best_node**) in the open list and expands it until a plan is found or the open list is empty. In contrast to

[4]Fluents considered as subgoals during the extraction of a relaxed plan are either problem goals, or preconditions of actions not yet marked true (i.e. no action has been selected yet in the relaxed plan to produce them). Subgoals required to be true at the first level of the planning graph (at time point 1) can thus be produced by some actions executable in the initial state. Choosing only one action for each of these subgoals could then be enough; but in FF, all actions that produce them are considered as helpful

[5]It must be noted that an action can be helpful for a given state, but rescue for another state: it depends on the role it plays in the relaxed plan and in the planning graph.

standard search algorithms, the actions chosen to be applied to the state are already known, as they are part of the information attached to the node. These actions come from the computation of the heuristic in the function **compute_heuristic** called by **compute_node**, which returns a set of helpful actions and a set of rescue actions.

pop_best_node(): returns the best node of the open list. Nodes are compared following three criteria of decreasing importance. Let $N = \langle S, P, A, h, f\rangle$ be a node in the open list. The first criterion is the value of the flag f: $helpful$ is preferred over $rescue$. When two flags are equal, the second criterion minimizes the heuristic value $f(S) = W \times h + Length(P)$, as in WA^* algorithm (Pearl 1983). In our current implementation, we use $W = 3$. Increasing the value of the W parameter can lead faster to a solution, but with lower plan quality (Korf 1993). As reported in (Bonet & Geffner 2001) for the HSP2 planner, a value between 2 and 10 does not make a significant difference (they use $W = 5$); but from our experiments, we found that using a lower value improves a bit plan quality while having no impact on efficiency. When two heuristic values are equal, the third criterion minimizes the length of the plan P that lead to S. This generally gives better plan quality than minimizing h, which could be another possibility.

compute_node(S, P): it is called by **LOBFS** over the initial state and the empty plan, or by **LOBFS** or itself over a newly created state S and the plan P that lead to that state (see Figure 1). Calls from **LOBFS** come from the initial state or the selection of a node in the open list and the application of one action to a given state. Calls from itself come from the computation of a valid plan by the lookahead algorithm and its application to a given state.

If the state S belongs to the close list or is a goal state, the function terminates. Otherwise, the state is evaluated by the heuristic function (**compute_heuristic**, which returns a relaxed plan, a set of helpful actions and a set of rescue actions). This operation is performed the first time with the goal-preferred actions (actions that do not delete a fluent that belongs to the goal and do not belong to the initial state).

If a relaxed plan can be found, two nodes are added to the initial state: one node for the helpful actions (the flag is set to $helpful$) and one node for the rescue actions (the flag is set to $rescue$). A valid plan is then searched by the lookahead algorithm, and **compute_node** is called again over the state that results from the application of this plan to S (if this valid plan is at least two actions long).

If no relaxed plan can be found with the goal-preferred actions, the heuristic is evaluated again with all actions. In that case, we consider that S is of lower interest: if a relaxed plan is found, only one node is added to the open list (helpful actions and rescue actions are merged), the flag is set to $rescue$, and no lookahead is performed.

compute_heuristic(S, A): this function computes the heuristic value of the state S in a way similar to FF. At first, a relaxed planning graph is created, using only actions from the set A. This parameter allows us to try to restrict actions to be used to goal-preferred actions: this heuristic proved to be useful in some benchmark domains. Once created, the relaxed planning graph is then searched backward for a solution.

```
let Π = ⟨O, I, G⟩ ;    /* planning problem */
let GA = {a ∈ O | ∀f ∈ Del(a), f ∉ (G \ I)} ;
  /* goal-preferred actions */
let open = ∅ ;    /* open list: nodes to be expanded */
let close = ∅ ;    /* close list: already expanded nodes */

function LOBFS ()
  compute_node(I, ⟨⟩) ;
  while open ≠ ∅ do
    let ⟨S, P, actions, h, flag⟩ = pop_best_node()
    forall a ∈ actions do
      compute_node(S ↑ a, P ⊕ ⟨a⟩)
    endfor
  endwhile
end

function compute_node (S, P)    /* S: state, P: plan, */
  if S ∉ close then
    if G ⊆ S then output_and_exit(P) endif ;
    close ← close ∪ {S} ;
    let ⟨RP, H, R⟩ = compute_heuristic(S, GA) ;
    if RP ≠ fail then
      open ← open ∪ {⟨S, P, H, Length(RP), helpful⟩,
                     ⟨S, P, R, Length(RP), rescue⟩} ;
      let ⟨S′, P′⟩ = lookahead(S, RP) ;
      if Length(P′) ≥ 2 then
        compute_node(S′, P ⊕ P′)
      endif
    else
      let ⟨RP, H, R⟩ = compute_heuristic(S, O) ;
      if RP ≠ fail then
        open ← open ∪ {⟨S, P, H ∪ R, Length(RP), rescue⟩}
      endif
    endif
  endif
end
```

Figure 1: Lookahead Optimistic Best-First Search algorithm

In our current implementation, there are two main differences compared to FF. The first difference holds in the way actions are used for a relaxed plan. In FF, when an action is selected at a level i, its add effects are marked $true$ at level i (as in classical Graphplan), but also at level $i-1$. As a consequence, a precondition established at a given level and required by another action at the same level will not be considered as a new goal. In our implementation, add effects of an action are only marked true at level i, but its preconditions are required at level i and not at the first level they appear, as in FF. A precondition of an action can then be achieved by an action at the same level, and the range of actions that can be selected to achieve it is wider.

The second difference 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.

The differences between FF and our implementation we described here are all heuristic and motivated by our experiments, since making optimal decisions for these problems are not polynomial, as stated in (Hoffmann & Nebel 2001).

The function **compute_heuristic** returns a structure $\langle RP, H, R\rangle$ where: RP is a relaxed plan, H is a set of helpful actions, and R is a set of rescue actions. As completeness is preserved by the algorithm, we restrict the set of helpful actions compared to FF: they are only actions of the relaxed plan applicable in the state S for which we compute the heuristic. In FF, all the actions that are applicable in S and produce a goal at level 1 are considered as helpful. Rescue actions are all actions applicable in S and not present in RP, and are used only when no node with helpful actions is present in the open list. So, $H \cup R$ contains all the actions applicable in S.

lookahead(S, RP): this function searches a valid plan for a state S using the actions of a relaxed plan RP calculated by **compute_heuristic** (cf. Figure 2). 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

```
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 2: Lookahead algorithm

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.

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 (function **choose_best**; 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. This repairing technique is also completely heuristic, but gave good results in our experiments.

Experimental evaluation

Planners, benchmarks and objectives

We compare four planners: FF v2.3[6], and three different settings of our planning system called YAHSP (which stands for Yet Another Heuristic Search Planner[7]) implemented in Objective Caml[8] compiled for speed:

- BFS (Best First Search): classical *WA** search, with *W* = 3. The heuristic is based on the computation of a relaxed plan as in FF. BFS does not compute helpful actions: all possible children of a state are ordered thanks to their heuristic evaluation, whether they come from the application of an helpful action or not.
- OBFS (Optimistic Best First Search): identical to BFS, with the use of a flag indicating whether the actions attached to a node are helpful or rescue. A node containing helpful actions is always preferred over a node containing rescue actions.
- LOBFS (Lookahead Optimistic Best First Search): identical to OBFS, with the use of lookahead states and plans.

We use nine different domains[9]: the Logistics domain, the Mprime and Mystery domains created for the 1st IPC, the Freecell domain created for the 2nd IPC, and five domains created for the 3rd IPC (Rovers, Satellite, DriverLog, Depots, ZenoTravel). Problems for Logistics are 30 selected problems from the 2nd IPC (from the smallest to the biggest) and those for Freecell come from the 3rd IPC.

We classified these domains into three categories, in accordance with the way LOBFS solves them: easy ones (Rovers, Satellite, Logistics, DriverLog, ZenoTravel), medium difficulty ones (Mprime, Freecell), and difficult ones (Depots, Mystery).

Our objectives for these experiments are the following:

1. *To test the efficiency of our planning system over a state-of-the-art planner, FF*. Indeed, FF is known for its distinguished performances in numerous planning domains and successes in the 2nd and 3rd IPC. Although generally not as fast as FF in the BFS and OBFS settings, our planner compares well to FF.
2. *To evaluate the suitability of the optimistic search strategy*. This strategy allows us to use helpful actions in complete search algorithms. This is in contrast to their use in the local search part of FF. We will see in particular that the OBFS strategy is better than BFS in almost all the problems.
3. *To demonstrate that the use of a lookahead strategy greatly improves the performances of forward heuristic search*. Even more, we will see that our planner can solve problems that are substantially bigger than what other planners can handle (up to 10 times more atoms in the initial state and 16 times more goals in the last DriverLog problem).

All tests have been performed on a Pentium II - 450MHz machine with 512Mb of memory running Debian GNU/Linux 3.0. The maximal amount of time devoted to all planners for each problem was fixed to one hour. Problems in the various graphs are ordered following LOBFS running time for a better readability.

Easy problems

As original problems from the competition sets are solved very easily by LOBFS, we created 10 more problems in each domain with the available generators. The 20 first problems are the original ones, and the 10 following are newly created.

In order to fully understand the results we present here, it is very important to remark that: *difficulty (measured as the number of atoms in the initial state and the number of goals) between successive new created problems numbered from 21 to 30, increases much more than difficulty between original problems*. Indeed, the last problem we created in each problem is the largest one that can be handled by LOBFS within the memory constraints. As the solving time remains reasonable, larger problems could surely be solved in less than one hour with more memory.

As a consequence, the graphs representing plan length are divided into two parts: plan length for new created problems increases much more than for original ones. We also show in Table 1 some data about the largest problems solved by FF, OBFS and LOBFS, in order to realize the progress accomplished in the size of the problems that can be solved by a STRIPS planner.

For the five domains we presented in this section, the superiority of LOBFS over all planners and the superiority of OBFS over BFS are clearly demonstrated, while OBFS and FF have comparable performances (see Figure 3). Similar results have been observed in several other domains (Ferry, Gripper, Miconic-10 elevator).

The only drawback of LOBFS is sometimes a degradation of plan quality, but this remains limited: the trade-off between speed and quality tends without any doubt in favor of our lookahead technique.

---

[6]http://www.informatik.uni-freiburg.de/∼hoffmann/ff.html

[7]http://www.cril.univ-artois.fr/∼vidal/yahsp.html

[8]Objective Caml is a strongly-typed functional language from the ML family, with object oriented extensions (http://caml.inria.fr/).

[9]All domains and problems used in our experiments can be downloaded on the YAHSP home page.

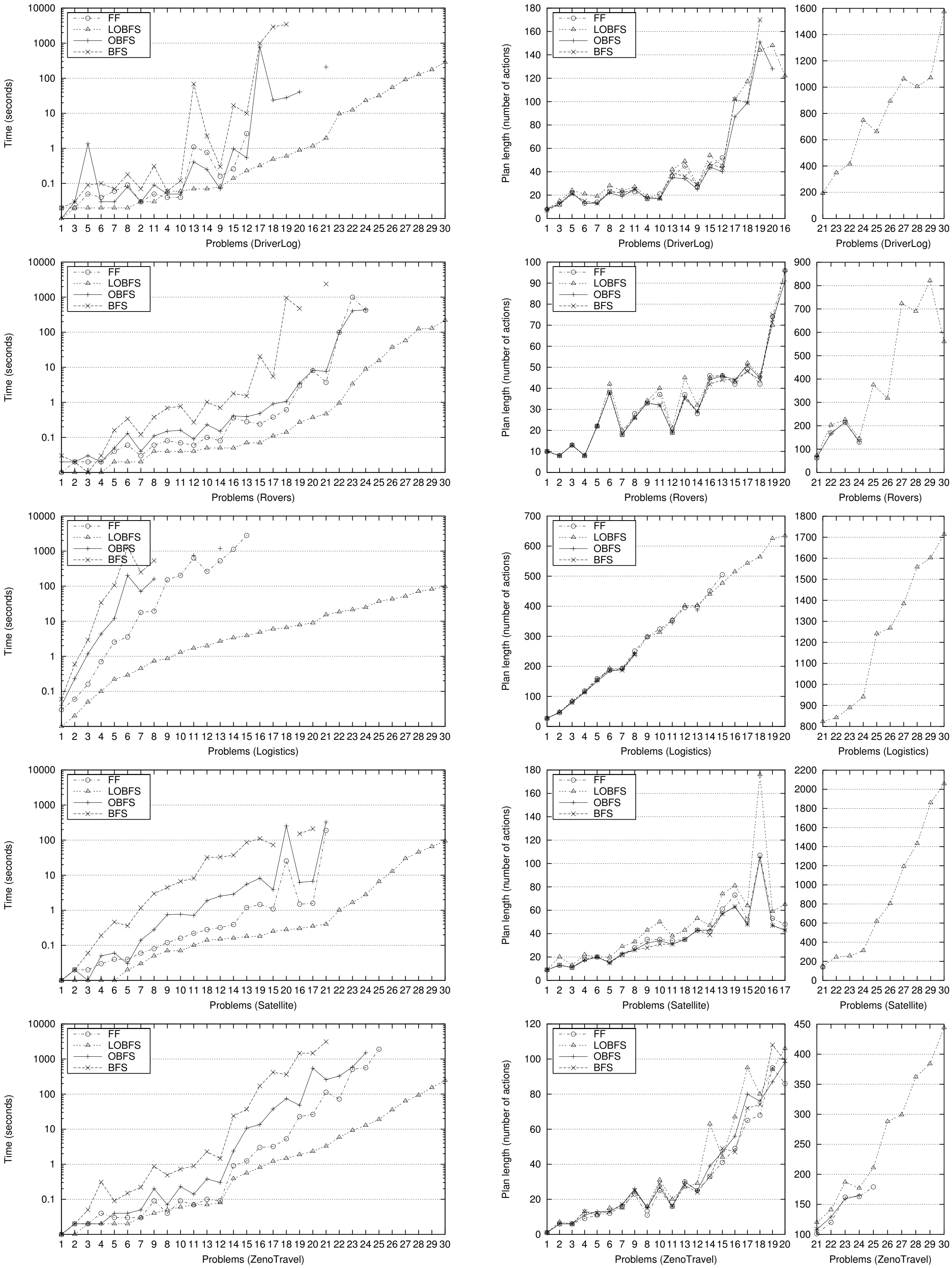

Figure 3: Easy domains

	Driver 15			Driver 21		Driver 30	Rovers 24			Rovers 30
Init atoms	227			607		2130	5920			35791
Goals	10			38		163	33			127
	FF	OBFS	LOBFS	FF	LOBFS	LOBFS	FF	OBFS	LOBFS	LOBFS
Plan length	44	44	54	184	193	1574	130	133	145	560
Evaluated nodes	161	273	4	3266	8	38	3876	2114	9	24
Search time	0.21	0.84	0.02	207.89	0.45	93.92	418.32	430.95	1.97	44.35
Total time	0.26	0.97	0.14	209.40	1.96	284.65	422.48	437.92	8.87	219.13
	Logistics 13			Logistics 15		Logistics 30	Satellite 21			Satellite 30
Init atoms	320			364		1140	971			10374
Goals	65			75		200	124			768
	FF	OBFS	LOBFS	FF	LOBFS	LOBFS	FF	OBFS	LOBFS	LOBFS
Plan length	398	387	403	505	477	1714	140	125	151	2058
Evaluated nodes	16456	16456	4	45785	4	5	22385	20370	5	5
Search time	527.21	1181.95	0.29	2792.51	0.42	16.64	188.69	328.42	0.12	33.73
Total time	528.10	1184.35	2.68	2793.82	3.88	96.69	188.82	328.70	0.40	94.24
	Zeno 24			Zeno 25		Zeno 30				
Init atoms	166			183		353				
Goals	45			49		100				
	FF	OBFS	LOBFS	FF	LOBFS	LOBFS				
Plan length	163	165	177	179	211	444				
Evaluated nodes	3481	5271	15	8714	16	20				
Search time	562.09	1496.81	4.15	1898.26	6.45	59.67				
Total time	564.07	1505.43	12.80	1901.03	18.98	247.06				

Table 1: Largest problems in easy domains

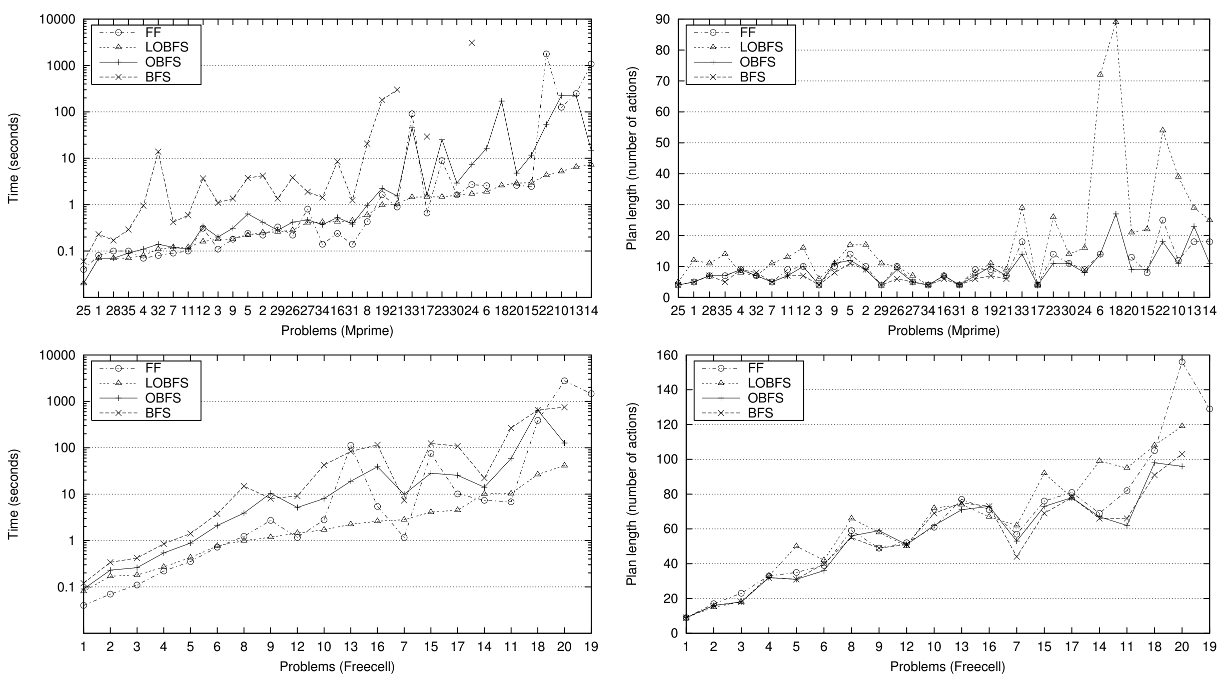

Figure 4: Medium difficulty domains

It is to be noted that FF had already good performances in these domains, that are for the most part transportation domains; but the time required for solving problems from these domains and the size of problems that can be handled have been considerably improved.

Medium difficulty problems

Although not as impressive as for the five first domains we studied, the improvements obtained using the lookahead technique are still interesting for these two domains, as LOBFS has much better performances than OBFS (see Figure 4). This last compares well to FF, and is more effi-

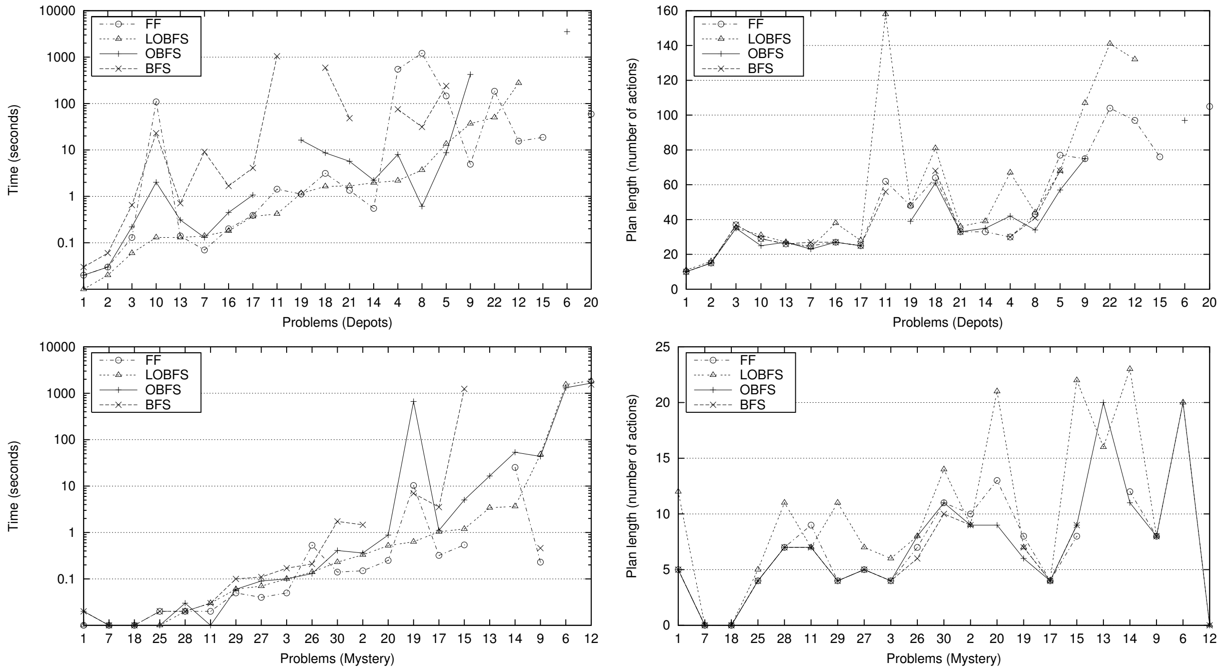

Figure 5: Difficult domains

cient than BFS. The loss in quality of solution plan observed for LOBFS remains limited to a small number of problems, and for example in Mprime domain where LOBFS solves all problems in less than 10 seconds, we could use LOBFS for getting a solution as fast as possible and then another planner to get a better solution. We can remark that when FF is faster than LOBFS, it is less than an order of magnitude; and when LOBFS is faster than FF, it is often more than one order of magnitude.

Difficult problems

Due to the loss in plan quality, the use of the lookahead technique is less interesting than in previous studied domains; it however allows to find plans for problems where OBFS fails to do so, and to get a better running time for a lot of problems (see Figure 5). Problems not solvable for any planner are not reported in the graphs. Further developments of the ideas presented in this paper should concentrate on improving the behavior of LOBFS for such domains, where there are a lot of subgoal interactions as in the Depots domain, or limited resources as in the mystery domain.

Lookahead utility

In order to try to characterize the effectiveness of the lookahead strategy, we studied the impact of the *lookahead utility* on the performance of our planner. The lookahead utility can be defined as the percentage of actions of the relaxed plans that are effectively used in lookahead plans. For example, a lookahead utility of 60 for a given problem means that on the average, 60% of the actions in the computed relaxed plans are used in lookahead plans. The graphs in Figure 6 represent the acceleration of the running time between LOBFS and OBFS (i.e. the running time of LOBFS divided by the running time of OBFS), for the lookahead utility of each problem.

We can remark that the highest utility does not lead to the best improvements in a given domain. For example in ZenoTravel, the best improvements can be found around an utility of 30%, while between 60% and 90%, the improvements are much more modest. This also happens in Freecell, where an utility below 40% is better than above 70%. This suggests that the strategy we defined (trying to use as most actions of relaxed plans as possible, that is maximize the utility), is perhaps not the best; or at least, producing better relaxed plans could improve the process. We can also observe that in domains where the lookahead strategy gives the best results for both time and quality, e.g. in Logistics domain, utility is grouped in a short interval (between 70% and 90%).

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. We then improved this search algorithm by using helpful actions in a way different than FF, that preserves completeness of the search algorithm in a strategy we called “optimistic”. 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 (Rovers, Logistics, DriverLog, ZenoTravel, Satellite), our planner can solve problems that

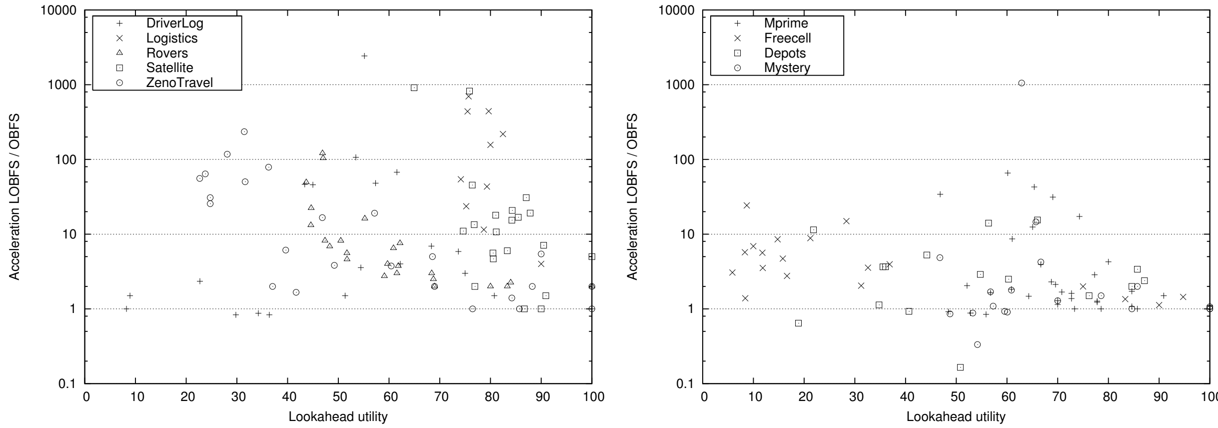

Figure 6: Improvement of the running time between OBFS and LOBFS in relation to the lookahead utility

are up to ten times bigger (in number of actions of the initial state) than those solved by FF or by the optimistic best-first search without lookahead. The efficiency for problems solved by all planners is also greatly improved when using the lookahead strategy. In domains that present more difficulty for all planners (Mystery, Depots), the use of the lookahead strategy can still improve performances for several problems. There are very few problems for which the optimistic search algorithm is better without lookahead. 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.

This work can be extended in a number of ways. Amongst them are improving the lookahead technique for domains containing many subgoal interactions (which could benefit a lot from the work about landmarks of (Porteous, Sebastia, & Hoffmann 2001)), and interpreting the results presented here in the light of recent works about complexity of planning benchmarks (Hoffmann 2001; 2002; Helmert 2003).

Acknowledgments

Thanks to Pierre Régnier for his help and numerous discussions. Thanks to Héctor Geffner for useful discussions, and who first suggested the measure of lookahead utility. Thanks also to the anonymous reviewers for their very helpful comments.

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. 1999. Planning as heuristic search: New results. In *Proc. ECP-99*, 360–372.

Bonet, B., and Geffner, H. 2000. HSP: Heuristic search planner. *AI Magazine* 21(2).

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

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *Proc. AAAI-97*, 714–719.

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

Helmert, M. 2003. Complexity results for standard benchmark domains in planning. *Artificial Intelligence* 143(2):219–262.

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

Hoffmann, J. 2001. Local search topology in planning benchmarks: An empirical analysis. In *Proc. IJCAI-2001*, 453–458.

Hoffmann, J. 2002. Local search topology in planning benchmarks: A theoretical analysis. In *Proc. AIPS-2002*, 379–387.

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.

Korf, R. 1993. Linear-space best-first search. *Artificial Intelligence* 62:41–78.

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.

Pearl, J. 1983. *Heuristics*. San Mateo, CA: Morgan Kaufmann.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *Proc. ECP-2001*, 37–48.

Vidal, V. 2003. A lookahead strategy for solving large planning problems (extended abstract). In *Proc. IJCAI-03*, 1524–1525.