A lookahead strategy for heuristic search planning

Vincent Vidal
IRIT – Université Paul Sabatier
118 route de Narbonne
31062 Toulouse Cedex 04, France
email: vvidal@irit.fr

Technical report IRIT/2002-35-R

Abstract

The planning as heuristic search framework, initiated by the planners ASP from Bonet, Loerincs and Geffner, and HSP from Bonet and Geffner, lead to some of the most performant planners, as demonstrated in the two previous editions of the International Planning Competition. We focus in this paper on a technique introduced by Hoffmann and Nebel in the FF planning system 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. 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, more information is derived from the planning graph and its solution, namely the helpful actions which permit FF to concentrate its efforts on more promising ways, forgetting the other actions in a local search algorithm. We introduce a novel way for extracting information from the computation of the heuristic and for tackling with helpful actions, by considering the high quality of the plans computed by the heuristic function 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, that will often bring us closer to a solution state. The lookahead state thus calculated is then added to the list of nodes that can be chosen to be developed following the numerical value of the heuristic. We use this lookahead strategy in a complete best-first search algorithm, modified in order to take into account helpful actions by preferring nodes that can be developed with such actions over nodes that can be developed with actions that are not considered as helpful. We then provide an empirical evaluation which demonstrates that in numerous planning benchmark domains, the performance of heuristic search planning and the size of the problems that can be handled have been drastically improved, while in more "difficult" domains these strategies remain interesting even if they sometimes degrade plan quality.

1 Introduction

Planning as heuristic search has proven to be a successful framework for 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 [BF95, BF97], Satplan [KS96, KMS96] and their descendants Blackbox [KS99], IPP [KNHD97], LCGP [CRV01], STAN [LF99], SGP [WAS98], ... Although most of these planners 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 such as planning as model checking, more 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 [BLG97], HSP and HSPr [BG01], lead to some of the most performant planners, as demonstrated in the two previous editions of the International Planning Competition with planners such as HSP2 [BG01], FF [HN01] and AltAlt [NK00, NK02]. 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 on a technique introduced in the FF planning system 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 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 actions which are executable in that state and produce fluents that where found to be goals 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 informations from the computation of the heuristic and for tackling 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 define in this paper an algorithm for combining some actions from each relaxed plan, in order to find the beginning of 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 developed following the numerical value of the heuristic. The best strategy we (empirically) found is to use as much actions as possible from each relaxed plans 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 can simply consist in augmenting the list of nodes to be developed (the open list) with some new nodes computed by the lookahead algorithm. The branching factor is slightly

[1] The 2^{nd} IPC home page can be found at `http://www.cs.toronto.edu/aips2000/`.

[2] The 3^{rd} IPC home page can be found at `http://www.dur.ac.uk/d.p.long/competition.html`.

increased, but the performances are generally better and completeness is not affected. In addition to this lookahead strategy, we propose a new way of 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. Once a state S is evaluated, two new nodes are added to the open list: one node that contains the helpful actions, which are the actions belonging to the relaxed plan computed for S and executable in S, and one node that contains all actions applicable in S and that do not belong to the relaxed plan (we call them *rescue* actions). A flag is added to each node, indicating whether the actions attached to it are helpful or rescue actions. We then add a criterium to the node selection mechanism, that always gives preference in developing a node containing helpful actions over a node containing rescue actions, whatever the heuristic estimates of these nodes are. As no action is lost and no node is pruned from the search space as in FF, completeness is preserved.

Our empirical 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 performant, while the lookahead strategy makes it able to solve very large problems in several domains. One drawback of our lookahead strategy is sometimes a degradation of plan quality, which we found to be critical for a few problems. But the trade-off between speed and quality, even in some "difficult" domains where solutions for some problems are substantially longer when using the lookahead strategy, seems to always tend in favor of it.

After giving classical definitions and notations in Section 2, we explain the main ideas of the paper and give theoretical issues in Section 3. We then give all details about the algorithms implemented in our planning system in Section 4, and illustrate them with an example from the well-known Logistics domain in Section 5. We finally present an experimental evaluation of our work in Section 6 before some related works in Section 7 and our conclusions in Section 8.

2 Definitions

Operators are STRIPS-like operators, without negation in their preconditions. We use a first order logic language L, constructed from the vocabularies V_x, V_c, V_p that respectively denote finite disjoint sets of symbols of variables, constants and predicates.

Definition 1 (operator) *An* operator, *denoted by o, is a triple $\langle pr, ad, de \rangle$ where pr, ad and de denote finite sets of atomic formulas of the language L. $Prec(o)$, $Add(o)$ and $Del(o)$ respectively denote the sets pr, ad and de of the operator o.*

Definition 2 (state, fluent) *A* state *is a finite set of ground atomic formulas (i.e. without any variable symbol). A ground atomic formula is also called a* fluent.

Definition 3 (action) *An* action *denoted by a is a ground instance $o\theta = \langle pr\theta, ad\theta, de\theta \rangle$ of an operator o which is obtained by applying a substitution θ defined with the language L such that $pr\theta$, $ad\theta$ and $de\theta$ are ground. $Prec(a)$, $Add(a)$, $Del(a)$ respectively denote the sets $pr\theta$, $ad\theta$, $de\theta$ and represent the preconditions, adds and deletes of the action a.*

Definition 4 (planning problem) *A* planning problem *is a triple $\Pi = \langle A, I, G \rangle$ where A denotes a finite set of actions (which are all the possible ground instantiations of a given set of operators defined on L), I denotes a finite set of fluents that represent the initial state, and G denotes a finite set of fluents that represent the goals.*

Definition 5 (relaxed planning problem) *Let* $\Pi = \langle A, I, G \rangle$ *be a planning problem. The* relaxed planning problem $\Pi' = \langle A', I, G \rangle$ *of* Π *is such that*

$$A' = \{\langle Prec(a), Add(a), \emptyset \rangle \mid a \in A\}$$

Definition 6 (plan) *A* plan *is a sequence of actions* $\langle a_1, \ldots, a_n \rangle$*. Let* $\Pi = \langle A, I, G \rangle$ *be a planning problem. The set of all plans constructed with actions of* A *is denoted by* $Plans(\Pi)$*.*

Definition 7 (First, Rest, Length, concatenation of plans) *We define the classical functions* $First$ *and* $Rest$ *on non-empty plans as* $First(\langle a_1, a_2, \ldots, a_n \rangle) = a_1$ *and* $Rest(\langle a_1, a_2, \ldots, a_n \rangle) = \langle a_2, \ldots, a_n \rangle$*, and* $Length$ *on all plans as* $Length(\langle a_1, \ldots, a_n \rangle) = n$ *(with* $Length(\langle\rangle) = 0$*). Let* $P_1 = \langle a_1, \ldots, a_n \rangle$ *and* $P_2 = \langle b_1, \ldots, b_m \rangle$ *be two plans. The* concatenation *of* P_1 *and* P_2 *(denoted by* $P_1 \oplus P_2$*) is defined by* $P_1 \oplus P_2 = \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$*.*

Definition 8 (application of a plan) *Let* S *be a state and* P *be a plan. The impossible state, which represents a failure in the application of a plan, is denoted by* $\perp$*. The* application *of* P *on* S *(denoted by* $S \Re P$*) is recursively defined by:*

$S \Re P =$
  if $P = \langle\rangle$ ***or*** $S = \perp$
  then S
  else /* *with* $P = \langle a_1, a_2, \ldots, a_n \rangle$ */
    if $Prec(a_1) \subseteq S$
      then $[(S - Del(a_1)) \cup Add(a_1)] \Re \langle a_2, \ldots, a_n \rangle$
      else $\perp$.

Definition 9 (valid plan, solution plan) *Let* $P = \langle a_1, \ldots, a_n \rangle$ *be a plan.* P *is valid for a state* S *iff* $S \Re P \neq \perp$*.* P *is a* solution plan *of a planning problem* $\Pi = \langle A, I, G \rangle$ *iff* $P \in Plans(\Pi)$ *and* $G \subseteq I \Re P$*.*

Definition 10 (reachable state) *Let* $\Pi = \langle A, I, G \rangle$ *be a planning problem. Let* S *and* S' *be two states.* S' *is* reachable *from* S *in* Π *iff there exists a plan* $P \in Plans(\Pi)$ *such that* $S \Re P = S'$*.*

3 The ideas

In this section, we expose the two main ideas of the paper (lookahead plans and use of helpful actions in a complete algorithm) in a very general way, without entering the details of their computation which will be given in the next section. For each of these ideas, we describe the main intuitions, we briefly explain how to obtain them in the context of a forward heuristic search planner based on FF [HN01] and we explain how to use them in a complete heuristic search algorithm.

3.1 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 a vertex starting from that node represents the application of one action to this state. For completeness issue, all actions that can be applied to one state must be considered. For example, let S be planning state and $\{a_1, \ldots, a_n\}$ be all the actions that can be applied to that state. Figure 1 represents the development of a node related to the state S, with each action $a_1, \ldots, a_n$ leading respectively to the states $S_1, \ldots, S_n$, in a complete search algorithm. The order in which these states will then be considered for development depends on the overall search strategy: depth-first, breadth-first, best-first...

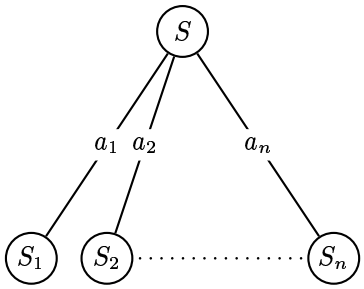

Figure 1: Node development

The main problem of forward heuristic search planning is of course the selection of the best node to be developed next. Difficulties appear when heuristic values for such nodes are very close to each other: no state can be distinguished from other states.

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 direct descendants of S. It could then be interesting to apply P to S, and use the resulting state S' as a knew node in the search. This state could be simply considered as a new descendant of S, as represented in the Figure 2: b_1 is an action that can be applied to S, b_2 is an action that can be applied to $S \Re b_1$, and so on until the application of b_m: the resulting state S' is such that $S' = S \Re \langle b_1, \ldots, b_m \rangle$. We have then two kind of vertices 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 but are considered in the search tree as direct descendants of the nodes they are connected to. Plans labeling vertices that lead to lookahead states will be called *lookahead plans*. Once a goal state is found, the solution plan is then the concatenation of single actions for classical vertices and lookahead plans for the other vertices.

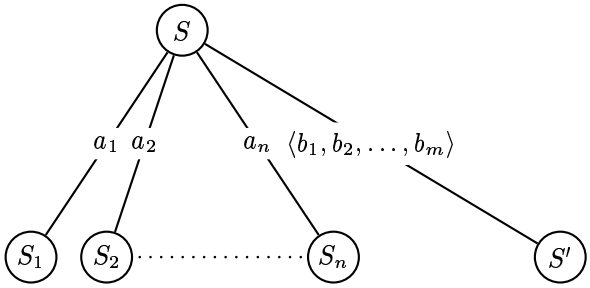

Figure 2: Node development with lookahead state

The computation of a heuristic for each states as is done in the FF planner offers a way to get such lookahead plans. FF creates a planning graph for each encountered state S, for the relaxed problem obtained by ignoring deletes of actions and using S as initial state. A relaxed plan is then extracted from these planning graphs in polynomial time and space. The length in number of actions of these relaxed plans corresponds to the heuristic evaluation of the states for which they are calculated. The relaxed plan for a state S is in general 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 in that they contain a lot of actions that belong to solution plans. We propose a way of computing lookahead plans from these relaxed plans, by trying as most actions as possible from them and keeping the ones that can be collected into a valid plan. The details of the algorithms are given in Section 4.

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

Even more, this lookahead strategy can preserve optimality of algorithms such as A* or IDA* when used with admissible heuristics, although some preliminary experiments we conducted did not gave interesting results (we tried to use a heuristic similar to the one used in [HG00], which corresponds to the length in number of levels of the relaxed planning graph, while still extracting a relaxed plan for computing a lookahead plan). The condition for preserving optimality is to consider that the cost of a vertex is the number of actions attached to it: 1 for classical vertices, and the length of the lookahead plan for vertices added by the lookahead algorithm. The evaluation function $f = g + h$ for a state S, with g being the cost for reaching S from the initial state and h being the estimation of the cost from S to the goal, thus takes into account the length of the lookahead plans within the value of g. This cost, that takes into account the length of lookahead plans, must also be considered by the way the state loop control (if used) is handled. Indeed, avoiding to explore already visited states increases a lot the performances of forward state-space planners [VR99]: a state S encountered at a depth d in the search tree does not need to be developed again when encountered again at a depth $d' \geq d$. In order to preserve optimality, we must consider the cost of a node as defined above, and not simply its depth: a state S encountered with a cost g in the search tree need not to be developed again when encountered again with a cost $g' \geq g$.

3.2 Using helpful actions: the "optimistic" 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 a 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 [HN01], with the concept of *helpful actions*. Once the planning graph for a state S is computed and a relaxed plan is extracted, 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 goals at the first level of the planning graph, during the extraction of the relaxed plan. The main drawback of this strategy, as used in FF, it 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 search when no solution is found by the local search algorithm.

We present in this paper 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 informations returned by the computation of the heuristic. The principle is the following:

1. 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.

2. 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[3]). For each class, one node is created for the state S, that contains the actions of that class returned by the heuristic function.

3. When a node is selected to be developed, the class of the actions attached to it is taken into account: in our "optimistic" algorithm, it is the first criterium: nodes containing helpful actions are always preferred over nodes containing rescue actions, whatever their numerical heuristic values are.

Once again, completeness and correctness are preserved: no information is lost. The way nodes are developed is simply modified: a state S is developed first with helpful actions, some other nodes are developed, and then S can potentially be developed 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, no action is lost. But optimality for use with admissible heuristics cannot of course be preserved: nothing guarantees that helpful actions always produce the shortest path to the goal.

4 Description of the algorithms

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

LOBFS(): this is the main function (see Figure 3). 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 from 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 develops it until a plan is found or the open list is empty. In contrast to standard search algorithms, the actions chosen to be applied to the node state are already known, as they are part of the informations 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.

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

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 criterium is the value of the flag f: $helpful$ is preferred over $rescue$. When two flags are equal, the second criterium minimizes the heuristic value $f(S) = W \times h + Length(P)$, as in WA^* algorithm. In our current implementation, we use $W = 3$. When two heuristic values are equal, the third criterium minimizes the length of the plan P that lead to S.

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 3). 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 has a lowest 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 (actions that delete a goal which is not in the initial state): this heuristic proved to be useful in some benchmark domains. Once created, the relaxed planning graph is then searched backward for a solution.

In our current implementation, there is 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 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 time i, but its preconditions are required at time 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 the 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 plan. a is ordered after a_1 iff: the level of the goal a was selected for is greater or equal than the level of the goal a_1 was selected for, 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 [HN01].

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 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 4). Several strategies can be imagined: searching plans with a limited number of actions, returning several possible plans, ... From our experiments, the best strategy we found is to search one plan, containing as most actions as possible from the relaxed plan, and to try to replace an action of RP which is not applicable by another action when no other choice is possible.

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 is 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. 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 updated 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 found in 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 A) applicable in S, which achieves an add effect of the action of $failed$ we observe, this add effect being a precondition 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**). 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 deleted from the current relaxed plan. This repairing technique is also completely heuristic, but gave good results in our experiments.

5 An example

Let us now illustrate the main ideas of the paper and the algorithms by the resolution of a small problem issued from the well-known Logistics benchmark domain. We provide here exactly what prints our planning system.

```
let Π = ⟨A, I, G⟩ ;    /* planning problem */
let GA = {a ∈ A | ∀f ∈ Del(a), f ∉ (G \ I)} ;    /* goal-preferred actions */
let open = ∅ ;    /* open list: nodes to be developed */
let close = ∅ ;    /* close list: already developed 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 ℜ ⟨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, A) ;
          if RP ≠ fail then
              open ← open ∪ {⟨S, P, Length(RP), H ∪ R, rescue⟩}
          endif
      endif
  endif
end
```

Figure 3: Lookahead Optimistic Best-First Search algorithm

```
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 ℜ ⟨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 possible_actions = {a ∈ A | f ∈ Add(a) ∧ Prec(a) ⊆ S} ;
                      if possible_actions ≠ ∅ then
                          let a = choose_best(possible_actions) ;
                          continue ← true ;
                          S ← S ℜ ⟨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 4: Lookahead algorithm

This Logistics world contains two cities (Paris and Toulouse), each one subdivided into two districts (post office and airport). Each city contains a truck that can move between districts of this city. There is one airplane, which can move between airports. Trucks and airplane can load and unload packages in every location they can go. In the initial state, three objects are located at Paris post office, each truck is located at its respective city post office, and the airplane is located at Paris airport. The goal is to move two of these objects to Toulouse post office, the third object staying at Paris post office. This problem can be described in typed PDDL [MGH+98] by the following:

```
(define (problem small-french-logistics)
 (:domain logistics)
 (:objects
  Paris Toulouse - city
  pa-po tlse-po - location
  pa-apt tlse-apt - airport
  A320 - airplane
  pa-truck tlse-truck - truck
  obj1 obj2 obj3 - package)
 (:init
  (in-city pa-po Paris) (in-city pa-apt Paris)
  (in-city tlse-po Toulouse) (in-city tlse-apt Toulouse)
  (at A320 pa-apt) (at pa-truck pa-po) (at tlse-truck tlse-po)
  (at obj1 pa-po) (at obj2 pa-po) (at obj3 pa-po))
 (:goal
  (and (at obj1 tlse-po) (at obj2 tlse-po) (at obj3 pa-po))))
```

The open and close lists are initiated to the empty set:

$open = \{\ \}$

$close = \{\ \}$

As the `in-city` predicate is only used for constraining trucks to move between districts of the same city, we can remove fluents using it from the initial state. They can be handled in a separate way, typically by the instantiation procedure. The initial state I can then be described by the following:

$I = \{$ `(at A320 pa-apt)`, `(at pa-truck pa-po)`,
`(at tlse-truck tlse-po)`, `(at obj1 pa-po)`, `(at obj2 pa-po)`,
`(at obj3 pa-po)` $\}$

The function **compute_node** is then called on the initial state I and the empty plan. As I does not belong to the close list, it is evaluated by the heuristic function **compute_heuristic** which returns a relaxed plan P_1, a set of helpful actions H_1 and a set of rescue actions R_1. The action `(LOAD-TRUCK obj3 pa-truck pa-po)` is not considered as helpful because it is not used into the relaxed plan. The union of H_1 and R_1 represents all actions applicable in I.

$P_1 = \langle$ (LOAD-TRUCK obj1 pa-truck pa-po),
(LOAD-TRUCK obj2 pa-truck pa-po),
(DRIVE-TRUCK pa-truck pa-po pa-apt Paris),
(UNLOAD-TRUCK obj2 pa-truck pa-apt),
(UNLOAD-TRUCK obj1 pa-truck pa-apt),
(LOAD-AIRPLANE obj2 A320 pa-apt),
(LOAD-AIRPLANE obj1 A320 pa-apt),
(FLY-AIRPLANE A320 pa-apt tlse-apt),
(DRIVE-TRUCK tlse-truck tlse-po tlse-apt Toulouse),
(UNLOAD-AIRPLANE obj1 A320 tlse-apt),
(UNLOAD-AIRPLANE obj2 A320 tlse-apt),
(UNLOAD-TRUCK obj1 tlse-truck tlse-po),
(LOAD-TRUCK obj1 tlse-truck tlse-apt),
(UNLOAD-TRUCK obj2 tlse-truck tlse-po),
(LOAD-TRUCK obj2 tlse-truck tlse-apt) $\rangle$

$H_1 = \{$ (LOAD-TRUCK obj1 pa-truck pa-po),
(LOAD-TRUCK obj2 pa-truck pa-po),
(DRIVE-TRUCK pa-truck pa-po pa-apt Paris),
(FLY-AIRPLANE A320 pa-apt tlse-apt),
(DRIVE-TRUCK tlse-truck tlse-po tlse-apt Toulouse) $\}$

$R_1 = \{$ (LOAD-TRUCK obj3 pa-truck pa-po) $\}$

As a relaxed plan has been found, the open list is augmented with two new nodes, one for the helpful actions and one for the rescue actions. The heuristic value 15 corresponds to the number of actions of the relaxed plan P_1. The initial state I is added to the close list.

$open = \{ \langle I, \langle\rangle, H_1, 15, helpful\rangle, \langle I, \langle\rangle, R_1, 15, rescue\rangle \}$

$close = \{ I \}$

At that point, a classical search algorithm would have selected and developed a node of the open list. The preferred node for our algorithm would be the one with the flag set to $helpful$. But as the relaxed plan has been found with only using goal-preferred actions, a lookahead plan will be computed from the initial state.

This is motivated by the fact that the relaxed plan P_1 has a very good quality. We can remark that the eleven first actions of P_1 give the beginning of an optimal plan that solves the problem: the two objects that must move to Toulouse are loaded into the truck at Paris post office, the truck goes to Paris airport, the two objects are unloaded from the truck and boarded into the airplane, the airplane flies from Paris to Toulouse while one truck goes from Toulouse post office to Toulouse airport, and the two objects are unloaded from the airplane.

There is a problem with the following four actions: as delete lists of actions are not taken into account, the fact that one truck is still at Toulouse post office remains true while extracting a relaxed plan: the action of driving the truck from the airport to the post office does not appear. For the same reason, the final actions of loading objects and unloading them are badly ordered.

The function **lookahead** returns $\langle S_1, P_1' \rangle = \text{lookahead}(I, P_1)$, with $S_1 = I \Re P'1$:

$P'1 = \langle$ (LOAD-TRUCK obj1 pa-truck pa-po),
(LOAD-TRUCK obj2 pa-truck pa-po),
(DRIVE-TRUCK pa-truck pa-po pa-apt Paris),
(UNLOAD-TRUCK obj2 pa-truck pa-apt),
(UNLOAD-TRUCK obj1 pa-truck pa-apt),
(LOAD-AIRPLANE obj2 A320 pa-apt),
(LOAD-AIRPLANE obj1 A320 pa-apt),
(FLY-AIRPLANE A320 pa-apt tlse-apt),
(DRIVE-TRUCK tlse-truck tlse-po tlse-apt Toulouse),
(UNLOAD-AIRPLANE obj1 A320 tlse-apt),
(UNLOAD-AIRPLANE obj2 A320 tlse-apt),
(LOAD-TRUCK obj1 tlse-truck tlse-apt),
(LOAD-TRUCK obj2 tlse-truck tlse-apt) $\rangle$

$S_1 = \{$ (at A320 tlse-apt), (at pa-truck pa-apt),
(at tlse-truck tlse-apt),(in obj1 tlse-truck),
(in obj2 tlse-truck), (at obj3 pa-po) $\}$

The eleven first actions have been taken without any change, and the algorithm found that the two actions of loading objects into Toulouse truck can be executed without the two actions of unloading these objects from the truck, which were present in the relaxed plan. The repairing part of the lookahead algorithm is not useful here, because no action in the global set of actions can have the same effects that unloading the objects at Toulouse post-office, which cannot be executed here.

The lookahead plan P'_1 being more than one action long, the function **compute_node** is recursively called on the state S_1 and the plan P'_1 that lead to it. As the state S_1 does not belong to the close list, it is evaluated by the heuristic function **compute_heuristic** which returns a relaxed plan P_2, a set of helpful actions H_2 and a set of rescue actions R_2:

$P_2 = \langle$ (UNLOAD-TRUCK obj1 tlse-truck tlse-po),
(DRIVE-TRUCK tlse-truck tlse-apt tlse-po Toulouse),
(UNLOAD-TRUCK obj2 tlse-truck tlse-po) $\rangle$

$H_2 = \{$ (DRIVE-TRUCK tlse-truck tlse-apt tlse-po Toulouse) $\}$

$R_2 = \{$ (UNLOAD-TRUCK obj1 tlse-truck tlse-apt),
(UNLOAD-TRUCK obj2 tlse-truck tlse-apt),
(FLY-AIRPLANE A320 tlse-apt pa-apt),
(DRIVE-TRUCK pa-truck pa-apt pa-po Paris)
(LOAD-TRUCK obj3 pa-truck pa-po) $\}$

A relaxed plan has been found this time again, so the open list is augmented with two new nodes: one for the helpful actions and one for the rescue actions. There is now four nodes in the open list, as none of them has been developed yet, and S_1 is entered into the close list:

$open = \{ \langle I, \langle\rangle, H_1, 15, helpful\rangle, \langle I, \langle\rangle, R_1, 15, rescue\rangle, \langle S_1, P'1, H_2, 3, helpful\rangle,$
$\langle S_1, P'1, R_2, 3, rescue\rangle \}$

$close = \{ I, S_1 \}$

Once again, the relaxed plan has been computed with only using goal-preferred actions. So, a lookahead plan is searched from the state S_1. In the case of a choice by the function

pop_best_nodes, the nodes would have been ordered by decreasing interest as follows:

1. $\langle S_1, P'1, H_2, 3, helpful\rangle$: the flag is $helpful$ and its heuristic value evaluates to $f(S_1) = 3 \times 3 + Length(P'1) = 9 + 13 = 22$,
2. $\langle I, \langle\rangle, H_1, 15, helpful\rangle$: the flag is $helpful$ and its heuristic value evaluates to $f(I) = 15 \times 3 + Length(\langle\rangle) = 45$,
3. $\langle S_1, P'1, R_2, 3, rescue\rangle$: the flag is $rescue$ and its heuristic value evaluates to $f(S_1) = 3 \times 3 + Length(P'1) = 9 + 13 = 22$,
4. $\langle I, \langle\rangle, R_1, 15, rescue\rangle$: the flag is $rescue$ and its heuristic value evaluates to $f(I) = 15 \times 3 + Length(\langle\rangle) = 45$.

The function **lookahead** returns $\langle S_2, P'_2\rangle = \text{lookahead}(S_1, P_2)$, with $S_2 = S_1 \Re P'_2$:

$P'_2 = \langle$ `(DRIVE-TRUCK tlse-truck tlse-apt tlse-po Toulouse),`
`(UNLOAD-TRUCK obj1 tlse-truck tlse-po),`
`(UNLOAD-TRUCK obj2 tlse-truck tlse-po)` $\rangle$

$S_2 = \{$ `(at A320 tlse-apt), (at pa-truck pa-apt),`
`(at tlse-truck tlse-po),`
`(at obj1 tlse-po),(at obj2 tlse-po), (at obj3 pa-po)` $\}$

The lookahead plan P'_2 being more than one action long, the function **compute_node** is recursively called on the state S_2 and the plan $(P'_1 \oplus P'_2)$ that lead to it. The state S_2 does not belong to the close list, but contains the goals of the problem: a solution plan is found and printed out, and the search is stopped. The solution plan is then:

$P'_1 \oplus P'_2 = \langle$ `(LOAD-TRUCK obj1 pa-truck pa-po),`
`(LOAD-TRUCK obj2 pa-truck pa-po),`
`(DRIVE-TRUCK pa-truck pa-po pa-apt Paris),`
`(UNLOAD-TRUCK obj2 pa-truck pa-apt),`
`(UNLOAD-TRUCK obj1 pa-truck pa-apt),`
`(LOAD-AIRPLANE obj2 A320 pa-apt),`
`(LOAD-AIRPLANE obj1 A320 pa-apt),`
`(FLY-AIRPLANE A320 pa-apt tlse-apt),`
`(DRIVE-TRUCK tlse-truck tlse-po tlse-apt Toulouse),`
`(UNLOAD-AIRPLANE obj1 A320 tlse-apt),`
`(UNLOAD-AIRPLANE obj2 A320 tlse-apt),`
`(LOAD-TRUCK obj1 tlse-truck tlse-apt),`
`(LOAD-TRUCK obj2 tlse-truck tlse-apt),`
`(DRIVE-TRUCK tlse-truck tlse-apt tlse-po Toulouse),`
`(UNLOAD-TRUCK obj2 tlse-truck tlse-po),`
`(UNLOAD-TRUCK obj1 tlse-truck tlse-po)` $\rangle$

As a conclusion to the treatment of this example, here is a comparison between its resolution by our planning system with three different settings: classical Best First Search (BFS), Optimistic Best First Search (OBFS), and Lookahead Optimistic Best First Search (LOBFS):

Developed nodes: BFS and OBFS develop 16 nodes, while LOBFS develops 0 nodes: LOBFS solves this problem without entering the real search part of the algorithm.

Evaluated nodes: BFS evaluates 65 states, OBFS evaluates 42 states. As the plan is 16 actions long, both of these algorithms evaluate unuseful states. The difference corresponds to the fact that OBFS uses only actions that are considered as helpful: rescue actions are in nodes whose flag is $rescue$, which take no part in node development as a solution is found before using them. LOBFS evaluates only 2 nodes.

Computed nodes: BFS computes 129 nodes, OBFS computes 43 nodes. The difference between evaluated and computed nodes for these algorithms is due to the fact that OBFS focuses faster on a solution state on the last stage of search. LOBFS computes only 16 states, which is only performed in the lookahead algorithm and never in the development of nodes, as none of them is developed.

6 Experimental evaluation

6.1 Planners, benchmarks and objectives

We compare four planners. The first one is the FF planning system v2.3 [HN01][4], the highly optimized heuristic search planner (implemented in C) that won the 2^{nd} International Planning Competition and was generally the top performer in the STRIPS and simple numeric tracks of the 3^{rd} International Planning Competition. The other planners consist in three different settings of our planning system called YAHSP (which stands for Yet Another Heuristic Search Planner, see[5]) implemented in Objective Caml[6] 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, with the differences detailed in Section 4.
- 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 the lookahead algorithm described in Section 4. A lookahead plan is searched each time the computation of a relaxed plan can be performed with goal-preferred actions.

We use nine different benchmark domains[7]: the classical Logistics domain, the Mprime and Mystery domains created for the 1^{st} IPC, the Freecell domain created for the 2^{nd} IPC, and the five STRIPS domains created for the 3^{rd} IPC (Rovers, Satellite, ZenoTravel, DriverLog, Depots). Problems for Logistics are those of the 2^{nd} IPC and problems for Freecell are those of the 3^{rd} IPC.

We classified these domains into three categories, in accordance with the way LOBFS solves them: easy problems in Section 6.2 (Rovers, Satellite, ZenoTravel, Logistics, DriverLog), medium difficulty problems in Section 6.3 (Mprime, Freecell), and difficult problems in Section 6.4 (Depots, Mystery).

Our objectives for these experiments are the following:

[4] The FF home page can be found at:
`http://www.informatik.uni-freiburg.de/~hoffmann/ff.html`.

[5] The YAHSP home page can be found at:
`http://www.irit.fr/recherches/RPDMP/vincent/yahsp.html`.

[6] Objective Caml is a strongly-typed functional language from the ML family, with object oriented extensions. Its home page can be found at `http://caml.inria.fr/`.

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

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 2^{nd} and 3^{rd} 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 enforced hill-climbing algorithm of FF which is not complete, and falls back into a classical complete best-first search when hill-climbing fails to find a plan. 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). The main drawback of our lookahead strategy is the degradation in plan quality (number of actions); however, we will see that this degradation remains generally very limited.

All tests have been performed in the same experimental conditions, on a Pentium II - 450MHz machine with 512Mb of memory running Debian GNU/Linux 3.0. The maximal amount of time allocated to all planners for each problem was fixed to one hour.

6.2 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 problem generators. The 20 first problems (numbered from 1 to 20) are the original ones, and the 10 following are newly created ones.

In order to fully understand the results we present here, it is very important to remark the following: *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 added a table for each domain that shows 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.

6.2.1 Rovers domain

This domain is inspired by planetary rovers problems. A collection of rovers navigate a planet surface, analyzing samples of soil and rock, and taking images of the planet. Analysis and images are then communicated back to a lander. Parallel communication between rovers and the lander is impossible.

OBFS and FF perform nearly the same, FF being slightly more efficient than OBFS (see Figure 5). They are much more faster than BFS and solve 24 problems, while BFS solves only 20 problems. OBFS is up to 300 times faster than BFS, on problem 21. LOBFS is much more efficient than all other planners, and solves all the problems. The largest problem solved by LOBFS contains 6 times more atoms in the initial state and 4 times more

goals than the largest problem solved by FF and OBFS (see Table 1). Plans found by LOBFS contain sometimes slightly more actions than plans found by other planners, but this remains limited (see Figure 6).

What is remarkable in this domain is that LOBFS never develops any node: plans are found by recursive calls between node computation and lookahead plan application, as in the example of the Logistics domain we studied in Section 5. Such a behavior will be encountered again for numerous problems in other domains, and is a reason of the good performances of LOBFS.

6.2.2 Satellite domain

This domain is inspired by space-applications and is a first step towards the "Ambitious Spacecraft" described by David Smith at AIPS-2000. It involves planning and scheduling a collection of observation tasks between multiple satellites, each equipped in slightly different ways. Satellites can point to different directions, supply power to one selected instrument, calibrate it to one target and take images of that target.

Running time curves for OBFS and FF are very similar, FF being generally more efficient than OBFS but at most around 10 times faster (see Figure 7). They are much more efficient than BFS and solve 21 problems against 19 for BFS. OBFS can be up to 30 times faster than BFS, on problem 17. LOBFS outperforms all other planners, and solves all the problems. The largest problem solved by LOBFS contains 10 times more atoms in the initial state and 6 times more goals than the largest problem solved by FF and OBFS (see Table 2). Plans found by LOBFS contain often slightly more actions than plans found by other planners, but this remains limited (see Figure 8) with the except of one problem for which it finds a plan containing 176 actions against 105 actions for OBFS.

In this domain, LOBFS develops only one node in 8 problems and develops no node in all other problems.

6.2.3 ZenoTravel domain

This is a transportation domain which involves transporting people around in planes. It was more specifically designed for numeric tracks of the competition, with different modes of movement: fast and slow. The fast movement consumes fuel faster than the slow once, making the search for a good quality plan (one using less fuel) much harder. In its STRIPS version, the fast movement gives no advantage, as quality is measured by plan length; but as more fuel is consumed, it still brings a penalty. The best behavior for STRIPS planner is then to avoid using the fast movement.

In this domain, FF is often more efficient than OBFS but at most around 10 times faster (see Figure 9). Both are much more efficient than BFS. FF solves 25 problems, against 24 for OBFS and 21 for BFS. OBFS is up to 30 times faster than BFS, on problem 19. LOBFS is much more efficient than all other planners, and solves all the problems. The largest problem solved by LOBFS contains 2 times more atoms in the initial state and 2 times more goals than the largest problem solved by OBFS (see Table 3). Plans found by LOBFS contain often slightly more actions than plans found by other planners, but this remains limited (see Figure 10) with the except of 2 or 3 problems where it finds up to 2 times more actions.

In this domain, LOBFS develops only 5 nodes in one problem, one node in 2 problems and develops no node in all other problems.

6.2.4 Logistics domain

This is the classical domain, as described in Section 5.

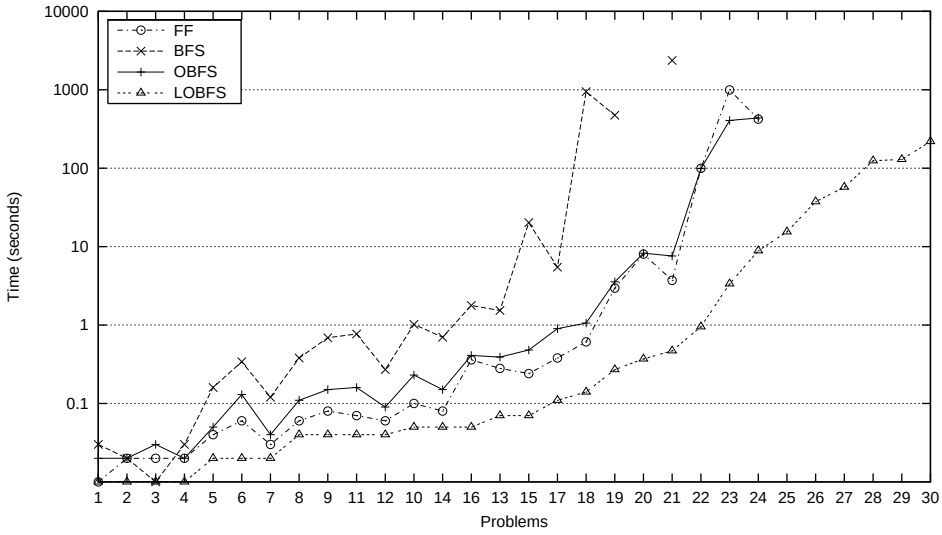

Figure 5: Rovers domain (CPU time)

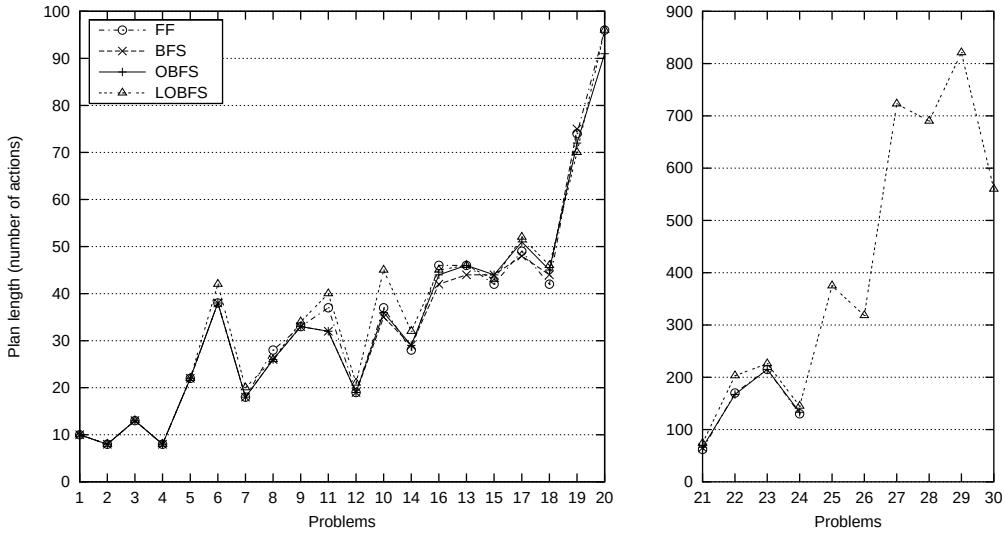

Figure 6: Rovers domain (plan length)

	Problem 24			Problem 30
Rovers	26			66
Way-points	80			200
Objectives	26			66
Cameras	16			40
Goals	26			66
Init atoms	5920			35791
Goals	33			127
	FF	OBFS	LOBFS	LOBFS
Plan length	130	133	145	560
Evaluated nodes	3876	2114	9	24
Search time	418.32	430.95	1.97	44.35
Total time	422.48	437.92	8.87	219.13

Table 1: Rovers domain (largest problems)

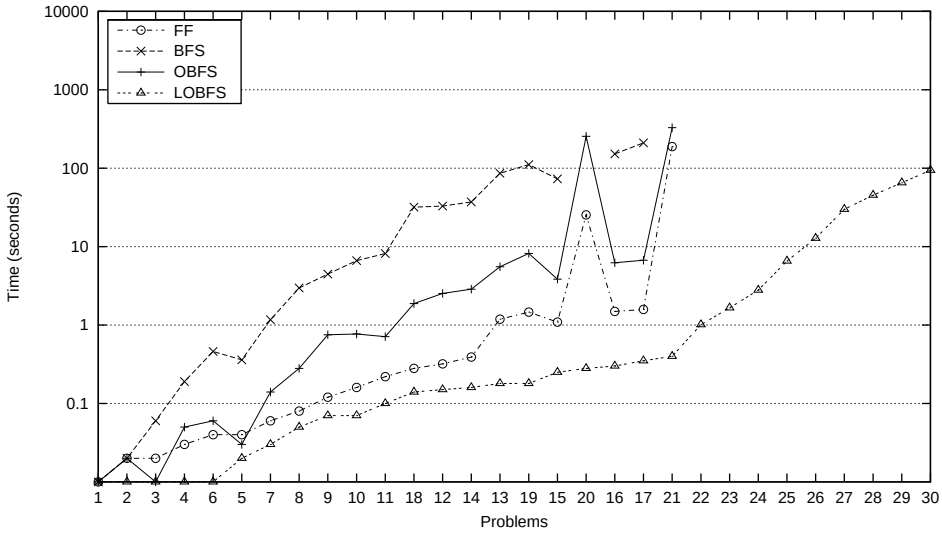

Figure 7: Satellite domain (CPU time)

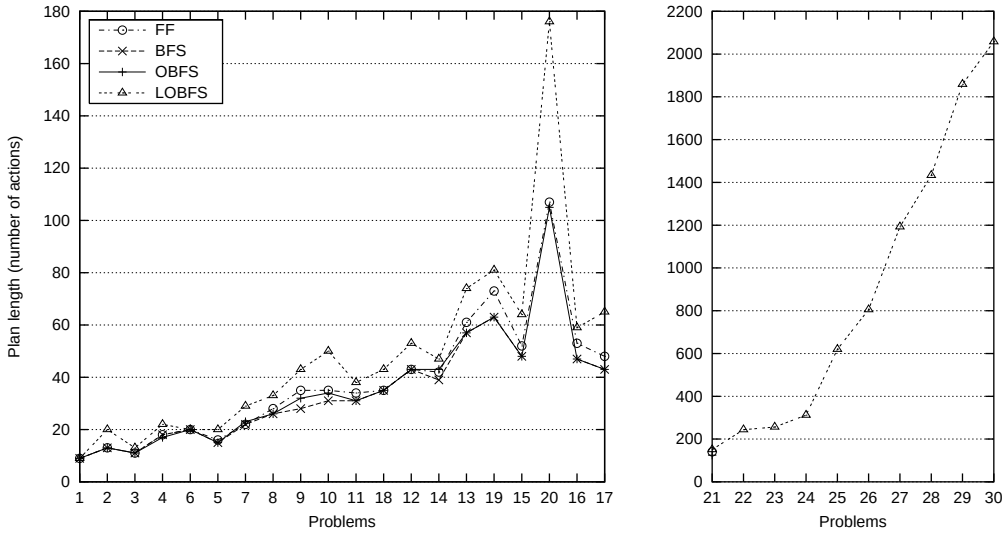

Figure 8: Satellite domain (plan length)

	Problem 21			Problem 30
Satellites	6			45
Instruments/sat.	12			50
Modes	12			50
Targets	6			25
Observations	25			100
Init atoms	971			10374
Goals	124			768
	FF	OBFS	LOBFS	LOBFS
Plan length	140	125	151	2058
Evaluated nodes	22385	20370	5	5
Search time	188.69	328.42	0.12	33.73
Total time	188.82	328.70	0.40	94.24

Table 2: Satellite domain (largest problems)

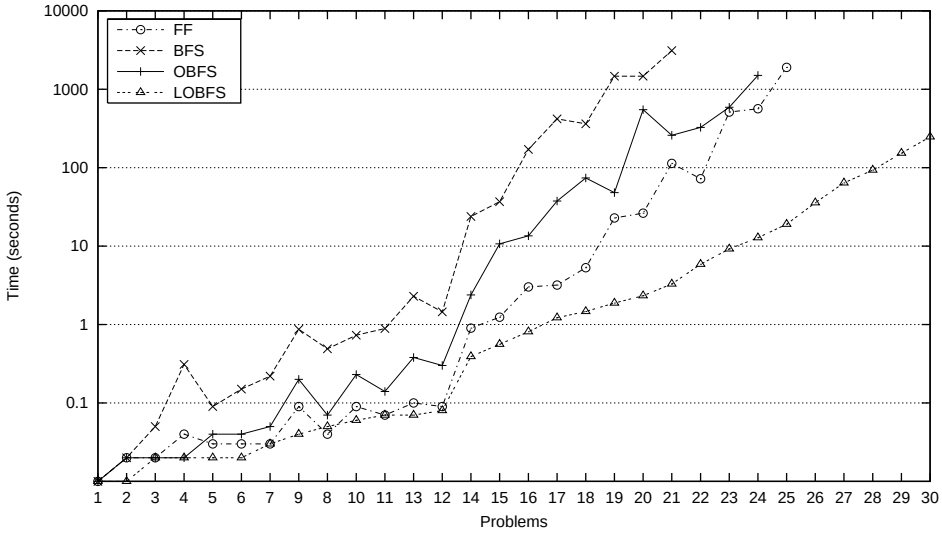

Figure 9: ZenoTravel domain (CPU time)

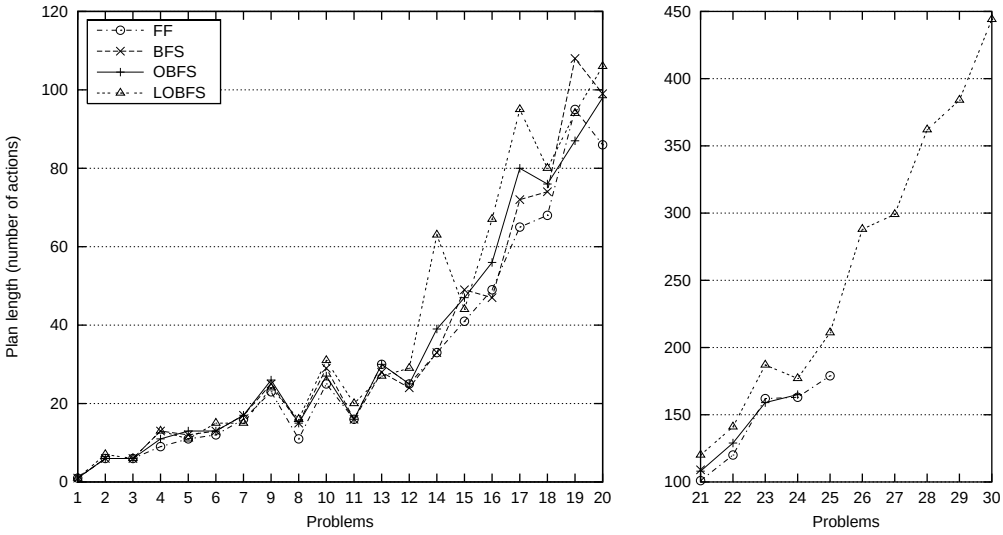

Figure 10: ZenoTravel domain (plan length)

	Problem 24			Problem 25		Problem 30
Cities	36			40		80
Planes	9			10		20
People	45			50		100
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 3: ZenoTravel domain (largest problems)

In this domain, FF is much more efficient than OBFS: up to 57 times faster in problem 6 (see Figure 11). FF solves 15 problems against 10 for OBFS. This last remains however more efficient than BFS, which solves only 8 problems. LOBFS is outperforms all other planners, and solves all the problems. The largest problem solved by LOBFS contains 3.5 times more atoms in the initial state and 3 times more goals than the largest problem solved by OBFS (see Table 4). What is remarkable in this domain is that all planners give plans of approximately the same length, with only a few actions of difference (see Figure 12).

In this domain, LOBFS never develops any node.

6.2.5 DriverLog domain

This domain involves driving trucks around delivering packages between locations. The difference with a more classical transportation domain such as Logistics is that the trucks require drivers who must walk between trucks in order to drive them. The paths for walking and the roads for driving form different maps on the locations.

In this domain, OBFS is often more efficient than FF (see Figure 13). Both are much more efficient than BFS, but this last solves more problems than FF: FF solves 15 problems, BFS solves 18 problems, and OBFS solves 20 problems. LOBFS is much more efficient than all other planners, and solves all the problems. The largest problem solved by LOBFS contains 3.5 times more atoms in the initial state and 4 times more goals than the largest problem solved by OBFS (see Table 5). This problem contains also 9 times more atoms in the initial state and 16 times more goals than the largest problem solved by FF. Plans found by LOBFS contain sometimes slightly more actions than plans found by other planners, but this remains very limited (see Figure 14).

Problems from this domain seem to be a little bit more difficult for LOBFS than problems from previous domains: 13 problems require nodes to be developed. But there are still few of such nodes: 4 problems develop between 10 and 75 nodes, and 9 problems develop between 1 and 9 nodes.

6.2.6 Concluding remarks

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.

The only drawback of LOBFS is sometimes a slight degradation of plan quality, but this remains very limited: the trade-off between speed and quality tends without any doubt in favor of our lookahead technique. Several known benchmarks not presented here such as Ferry, Gripper, Miconic-10 elevator, are also solved very easily by LOBFS.

It is to be noted that the FF planner 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.

6.3 Medium difficulty problems

The results we present for domains in this category are not as advantageous for LOBFS as in the previous section, but still compares well to the results of the other planners.

6.3.1 Mprime domain

In this domain, OBFS and FF have very close performances (FF being slightly faster), with the except of problem 18 solved by OBFS but not by FF and two problems (22 and 14) solved much faster by OBFS (see Figure 15). BFS solves only 24 of the total 35 problems,

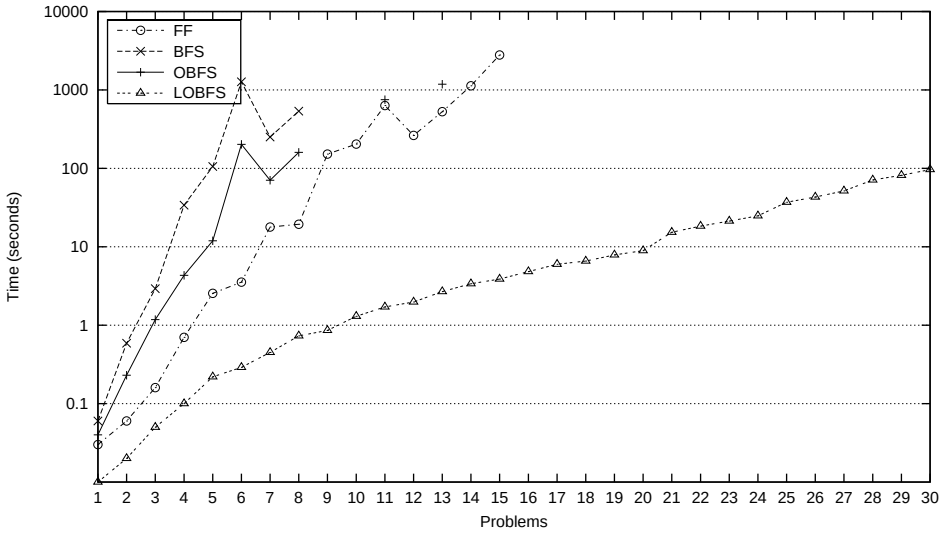

Figure 11: Logistics domain (CPU time)

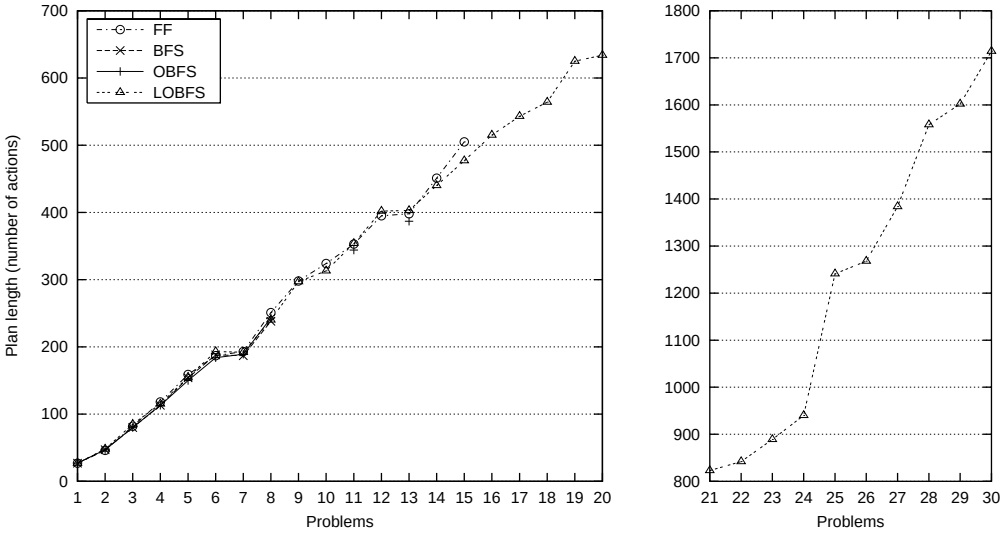

Figure 12: Logistics domain (plan length)

	Problem 13			Problem 15		Problem 30
Airplanes	6			7		20
Cities	22			25		50
City size	2			2		5
Packages	66			75		200
Init atoms	320			364		1140
Goals	65			75		200
	FF	OBFS	LOBFS	FF	LOBFS	LOBFS
Plan length	398	387	403	505	477	1714
Evaluated nodes	16456	16456	4	45785	4	5
Search time	527.21	1181.95	0.29	2792.51	0.42	16.64
Total time	528.10	1184.35	2.68	2793.82	3.88	96.69

Table 4: Logistics domain (largest problems)

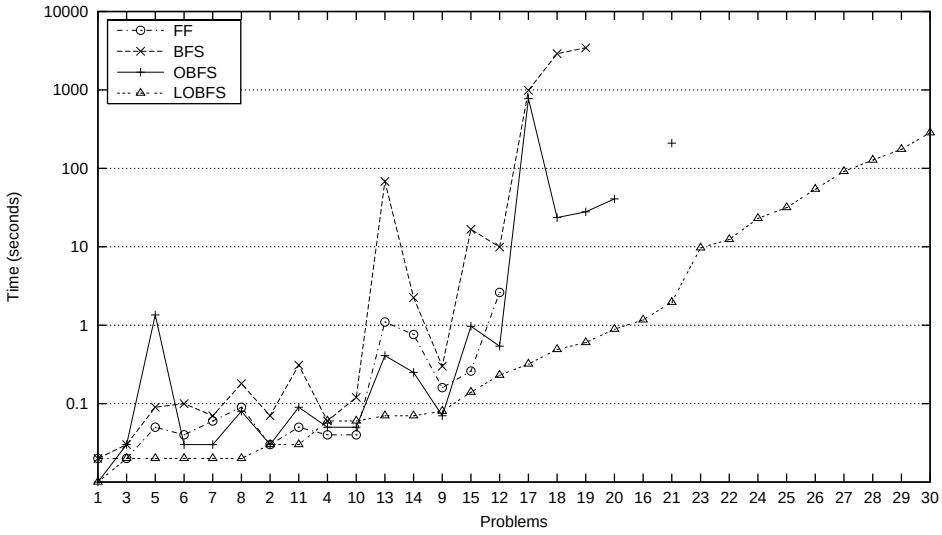

Figure 13: DriverLog domain (CPU time)

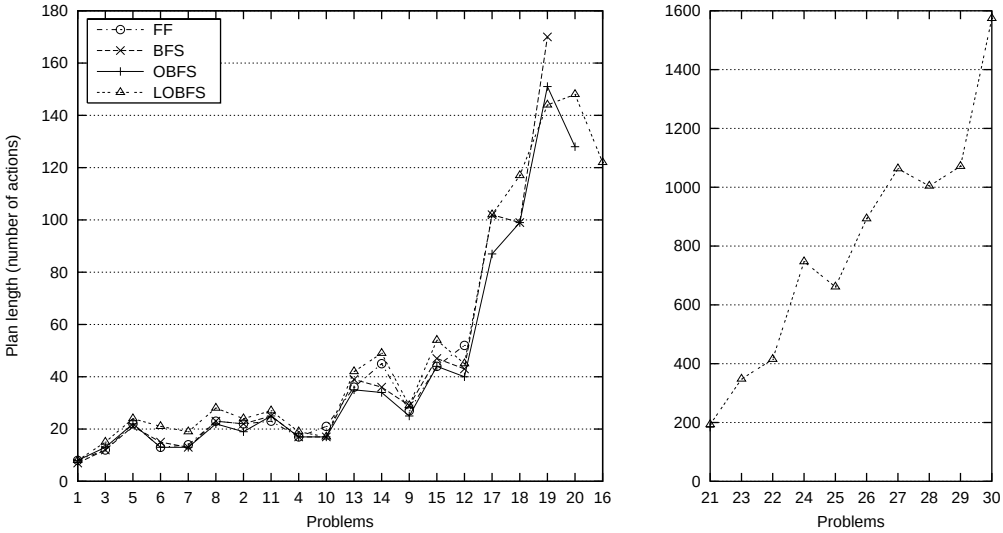

Figure 14: DriverLog domain (plan length)

	Problem 15			Problem 21		Problem 30
Road junctions	12			28		90
Drivers	4			10		40
Packages	8			30		120
Trucks	4			7		30
Init atoms	227			607		2130
Goals	10			38		163
	FF	OBFS	LOBFS	OBFS	LOBFS	LOBFS
Plan length	44	44	54	184	193	1574
Evaluated nodes	161	273	4	3266	8	38
Search time	0.21	0.84	0.02	207.89	0.45	93.92
Total time	0.26	0.97	0.14	209.40	1.96	284.65

Table 5: DriverLog (largest problems)

with running times much more important than the other planners ones. LOBFS solves all the problems and is generally faster. The problem of LOBFS in this domain is about plan quality: several plans found by LOBFS contain much more actions than plans found by other planners (see Figure 16). However, when looking closely to these results, we can make the following observation: problems for which LOBFS returns a very long plan compared to other planners are often the ones that cause difficulty to these planners. For instance, problem 22 is solved in 1770 seconds by FF, in 53 seconds by OBFS, but in only 4 seconds by LOBFS. The returned plans are 25 actions long for FF, 18 actions long for OBFS, and 54 actions long for LOBFS.

LOBFS develops only one node in 8 problems, and never develops any node in all other problems.

6.3.2 Freecell domain

It is the familiar solitaire game found on many computers, involving moving cards from an initial tableau, constrained by tight restrictions, to achieve a final suit-sorted collection of stacks.

In this domain, FF is more efficient than OBFS, although this last is faster for 3 problems (up to 23 times for problem 20, see Figure 17). It is interesting to note that the enforced hill-climbing strategy of FF fails to find a solution for these problems, and FF must switch to a complete best-first search algorithm. OBFS does not have this problem, as the optimistic best-first search algorithm is able to handle helpful actions until a solution is found or the state space is completely explored without finding a plan. OBFS is generally more efficient than BFS, except for two problems where BFS is slightly faster (the difference is not really significant). LOBFS is more efficient than BFS and OBFS, and tends to be faster than FF: up to 67 times for problem 20. However, problem 19, solved by FF, is not solved by any of our planners. Plan quality is equivalent for all planners, plans found by LOBFS being often slightly longer (see Figure 18).

Some problems seem to be more difficult for LOBFS than problems from domains we have studied so far, because LOBFS develops more nodes: five problems require between 70 and 250 nodes to be developed, while the other problems require less than 3 nodes. This is however limited compared to the number of nodes developed by OBFS: for instance in problem 18, OBFS develops 6915 nodes and LOBFS only 250.

6.3.3 Concluding remarks

Although not as impressive as for the five first domains we studied, the improvements obtained by using our lookahead technique are still interesting for these two domains, as LOBFS has much better performances than OBFS. This last compares well to FF, and is more efficient than BFS.

The loss in quality of plan solution 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.

Other techniques within our planner can also be imagined; for example we could let it run when a solution is found and try to get a better one in an anytime way. We have some preliminary results of such a technique that are not yet enough convincing and will not be presented here.

6.4 Difficult problems

The domains we test here are very difficult for all the planners we use. While still upgrading the efficiency of OBFS, our lookahead technique seems to be less useful than in previous

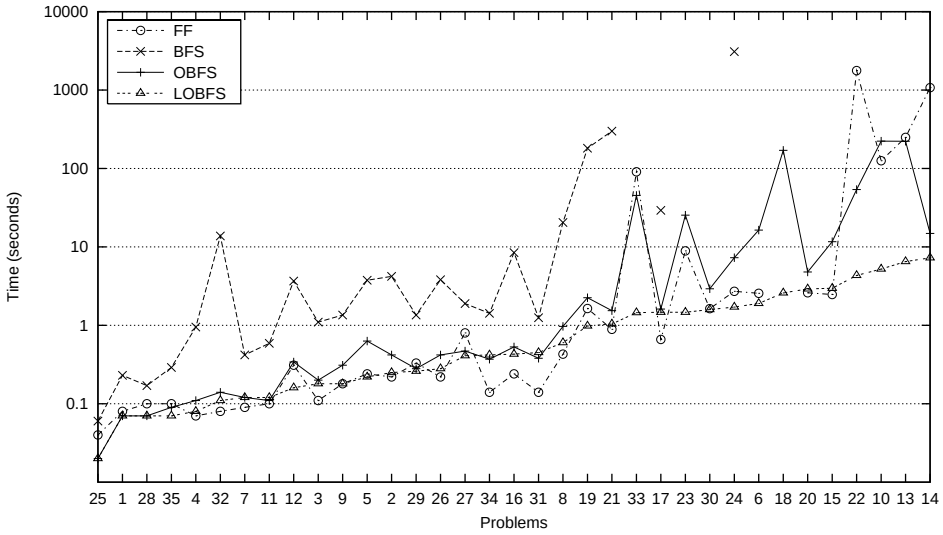

Figure 15: Mprime domain (CPU time)

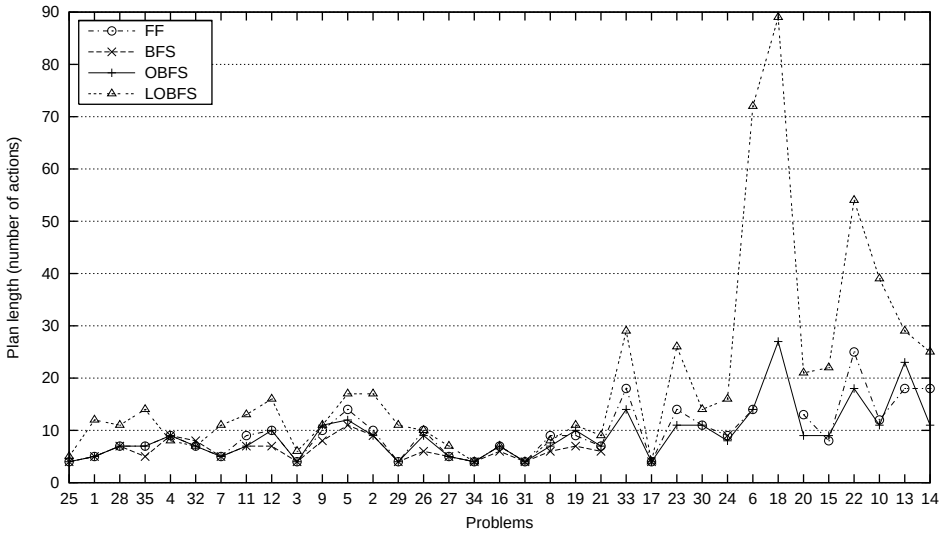

Figure 16: Mprime domain (plan length)

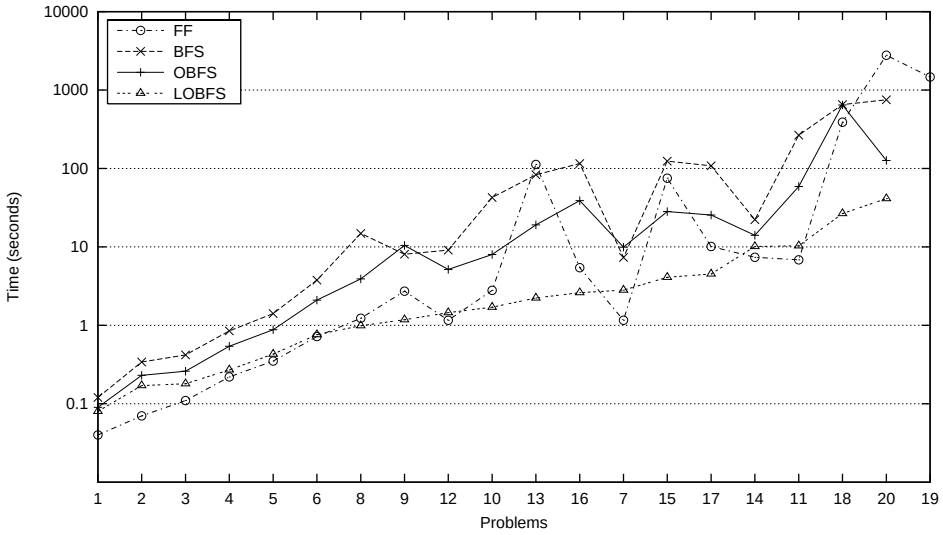

Figure 17: Freecell domain (CPU time)

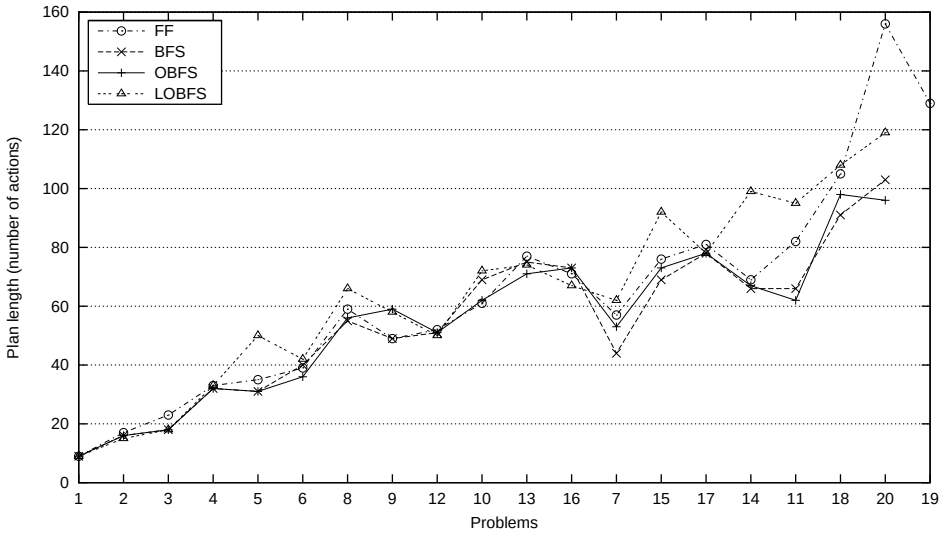

Figure 18: Freecell domain (plan length)

domains. The trade-off between speed and quality is not so clearly in favor of LOBFS.

6.4.1 Depots domain

This domain was devised in order to see what would happen if two previously well-researched domains were joined together. These were the logistics and blocks domains. They are combined to form a domain in which trucks can transport crates around and then the crates must be stacked onto pallets at their destinations. The stacking is achieved using hoists, so the stacking problem is like a blocks-world problem with hands. Trucks can behave like "tables", since the pallets on which crates are stacked are limited.

Problems from this domain are difficult for all planners (see Figure 19). FF solves more problems than the other planners: it solves 21 problems on 22, LOBFS solves 19 problems, OBFS solves 17 problems and BFS solves 14 problems. FF is generally more efficient then OBFS, with the except of 4 problems solved much faster by OBFS: it is up to 2000 times faster on problem 8. As in the Freecell domain, 3 of these problems (problems 4, 8 and 10) require that FF switches to a complete best-first search algorithm, unable to handle helpful actions contrary to OBFS. BFS is only slower than OBFS, while LOBFS is generally faster but in 2 problems and one problem solved by OBFS and not by LOBFS. This last compares well to FF and is often faster, but misses two problems more than FF. The plans returned by LOBFS are also significantly longer than plans returned by other planners in 7 problems (see Figure 20).

In this domain, LOBFS develops a lot of nodes for several problems: between 9 and 6659 nodes for 10 problems, and less than 3 for 9 problems. But it still develops very fewer nodes than OBFS: for example in problem 9, OBFS develops 48629 nodes while LOBFS develops only 6659 nodes.

The interest of our lookahead technique is debatable for this domain because of plan quality, but seems to be still interesting: 2 more problems are solved and LOBFS is generally faster than OBFS.

6.4.2 Mystery domain

What is very difficult to handle for heuristic search planners in this domain is the fact that some problems do not admit a solution plan. It is some time trivial, and can be determined by the instantiation procedure; but it more often requires to exhaustively explore the search space.

This domain is the most difficult one we used in our experiments, for all planners (see Figure 21). Among the 30 problems tested, only 16 are solved by FF, 14 are solved by BFS, and 18 by OBFS and LOBFS. BFS is the slowest planner, except on problem 9 that it solves much better than OBFS and LOBFS. FF is generally faster than OBFS and LOBFS, but solves less problems. OBFS and LOBFS have very close performances, with the except of 4 problems solved much better by LOBFS (up to 1050 times faster on problem 19). The main drawback of LOBFS, as in the Depots domain, is the quality of plans that it returns (see Figure 22): 9 plans found by LOBFS are significantly longer than plans returned by the other planners.

LOBFS develops less than 3 nodes in 15 problems, but develops a lot of nodes for the 3 other problems. On problem 12, which admits no solution plan, OBFS and LOBFS develop exactly the same number of nodes (1032977 nodes); but on problems 6 and 9, LOBFS develops slightly more nodes than OBFS: 15729 against 15663 for problem 6 and 3976 against 3960. The running time overhead due to the lookahead computation in LOBFS remain limited: LOBFS solves problem 6 in 1530 seconds against 1314 seconds for OBFS.

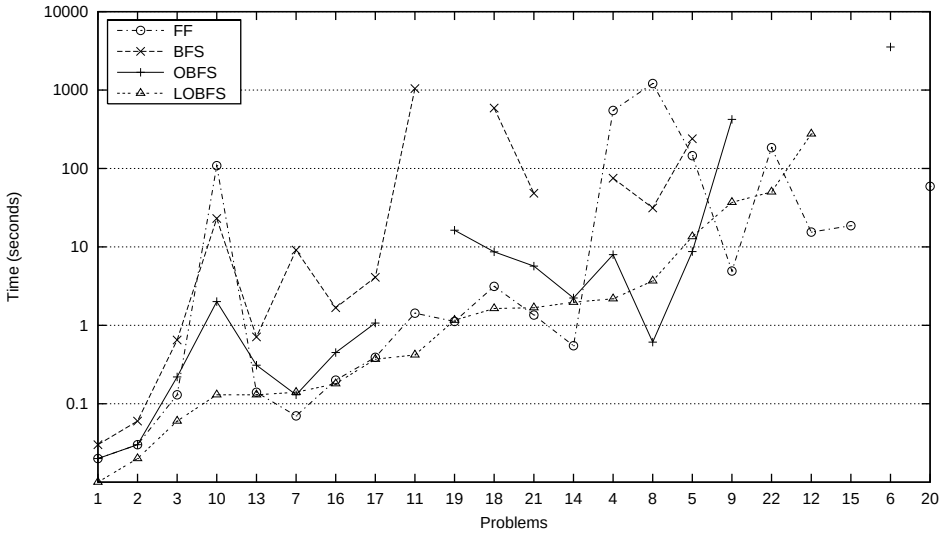

Figure 19: Depots domain (CPU time)

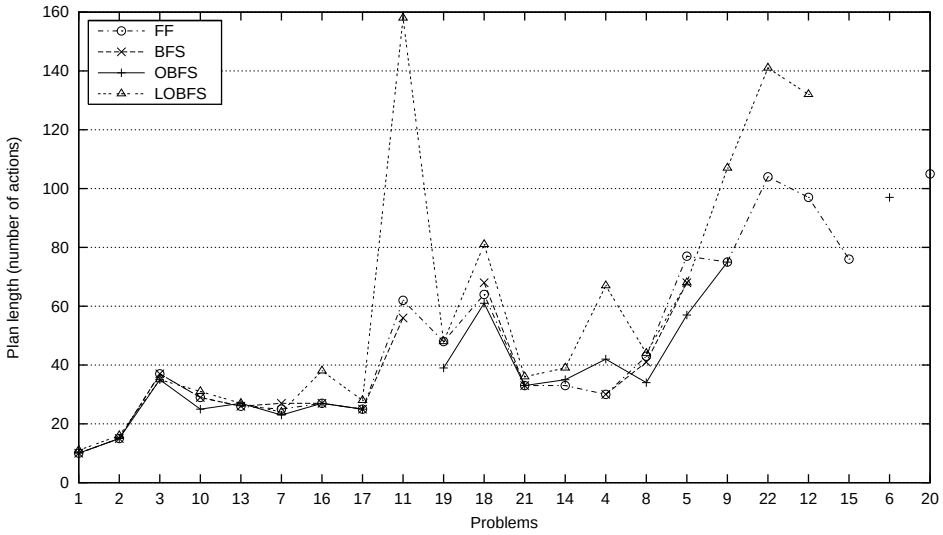

Figure 20: Depots domain (plan length)

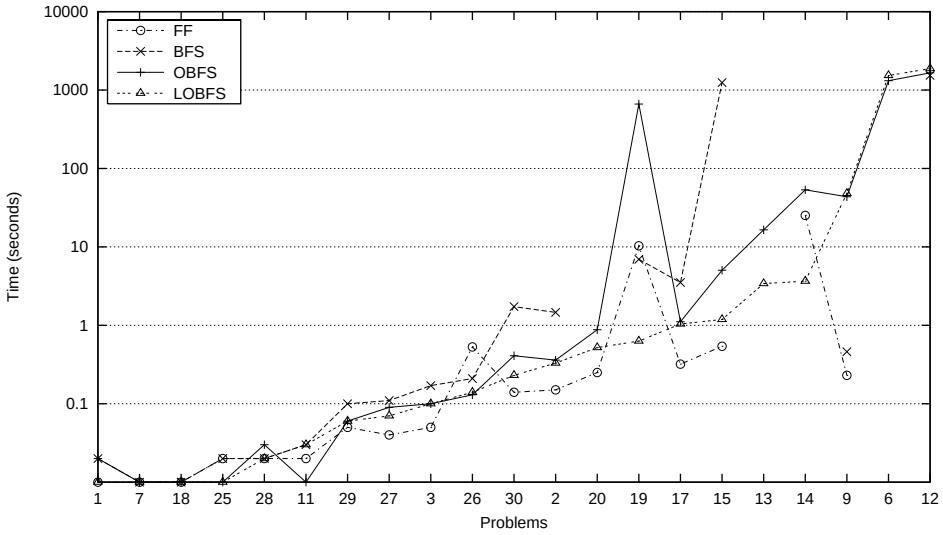

Figure 21: Mystery domain (CPU time)

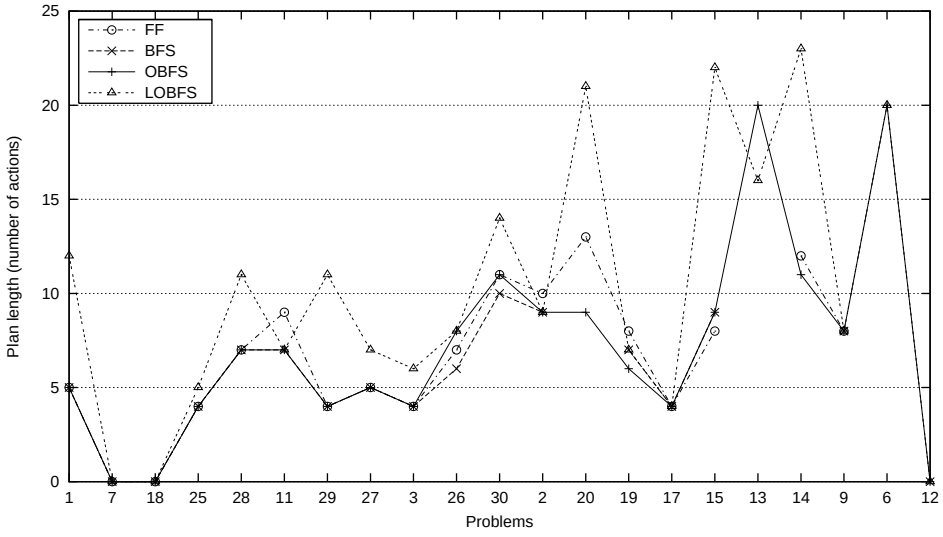

Figure 22: Mystery domain (plan length)

6.4.3 Concluding remarks

Due to the loss in plan quality, the use of our 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 some problems. 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.

6.5 Lookahead utility

7 Related works

8 Conclusion

Plans extracted from relaxed planning graphs are used in the FF planning system for providing an estimation of the length of the plans that lead to goal states. We presented a new method for deriving information from these relaxed plans, by the computation of lookahead plans. A lookahead plan, calculated by a polynomial algorithm, is composed of as most actions as possible from a given relaxed plan. It is valid for the state for which the relaxed plan is extracted in the sense that its application to this state is possible and leads to another state called lookahead state. We then expose how to employ the produced lookahead plans in a modified version of a classical best-first search algorithm to be used by a forward state-space planner. For each state evaluated by the heuristic function, a lookahead plan is computed and the state that it leads is added to the list of nodes to be developed, in the same way the states computed by using the actions executable in that state are added to that list: lookahead states thus produce supplementary nodes in the search tree. The difference with classical algorithms is that the vertices that lead to lookahead states are labeled by a plan and not by only one action. We then improved this search algorithm by using helpful actions in a way different than in FF, that preserves completeness of the search algorithm in a strategy we called “optimistic”. Instead of using them in a local search, and loosing actions that are not considered as helpful, we introduce a criterium for choosing nodes that always prefers to develop states that can be developed with helpful actions over states that cannot. 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…), 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 the optimistic best-first search without lookahead (we got plans with more than 2000 actions long). The efficiency for problems solved by all planners is also greatly improved when using our 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 is very few problems for which the optimistic search algorithm is better without lookahead. The price to pay for such improvements in the performances and in the size of the problems that can be handled resides in the quality of solution plans that can be in some cases severely damaged. However, there are few of such plans and quality remain generally very good compared to FF.

Acknowledgments

To be included…

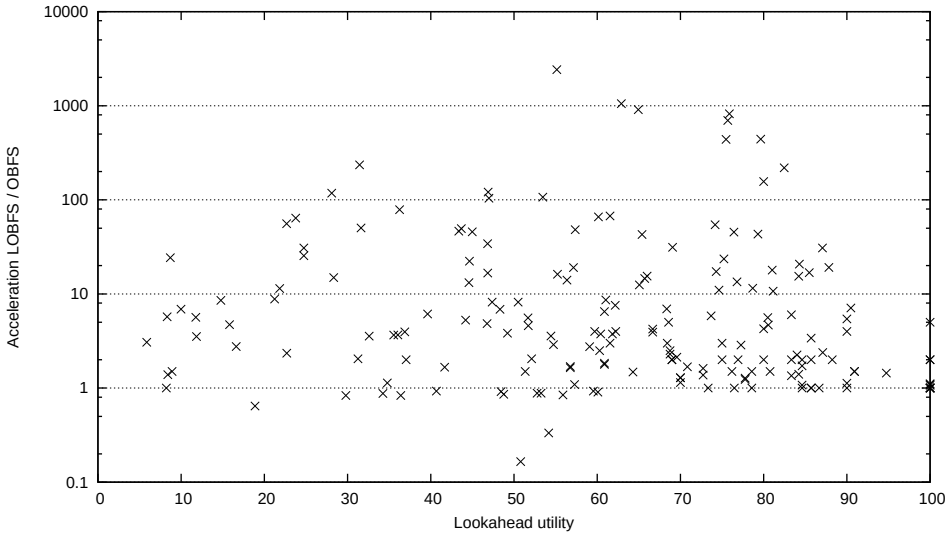

Figure 23: Utility

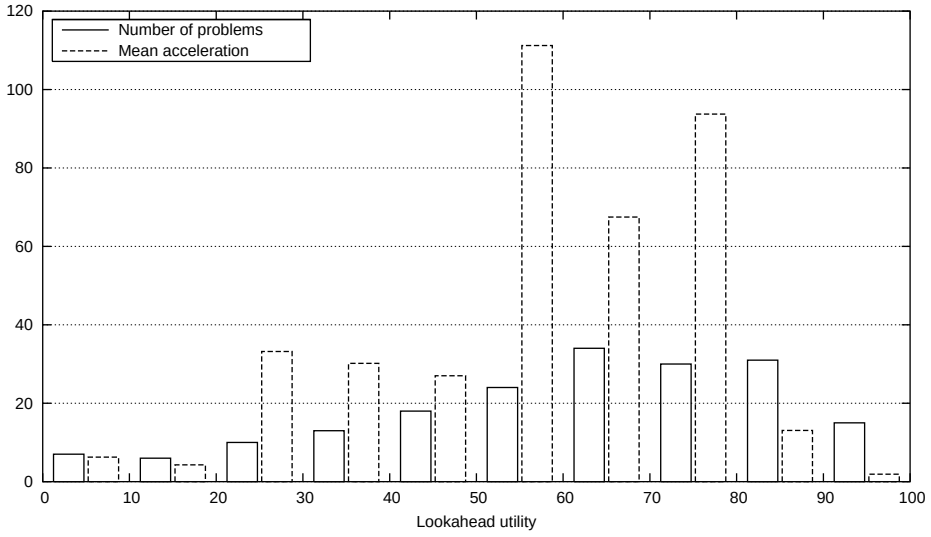

Figure 24: Utility

References

[BF95] A. Blum and M. Furst. Fast planning through planning-graphs analysis. In *Proc. IJCAI-95*, pages 1636–1642, 1995.

[BF97] A. Blum and M. Furst. Fast planning through planning-graphs analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[BG01] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[BLG97] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proc. AAAI-97*, pages 714–719, 1997.

[CRV01] M. Cayrol, P. Régnier, and V. Vidal. Least commitment in Graphplan. *Artificial Intelligence*, 130(1):85–118, 2001.

[HG00] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Proc. AIPS-2000*, pages 140–149, 2000.

[HN01] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[KMS96] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proc. KR-96*, pages 374–384, 1996.

[KNHD97] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning-graphs to an ADL subset. In *Proc. ECP-97*, pages 273–285, 1997.

[KS96] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proc. AAAI-96*, pages 1194–1201, 1996.

[KS99] H. Kautz and B. Selman. Unifying SAT-based and Graph-based planning. In *Proc. IJCAI-99*, pages 318–325, 1999.

[LF99] D. Long and M. Fox. The efficient implementation of the plan-graph in STAN. *JAIR*, 10:87–115, 1999.

[MGH+98] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, and D. Weld. *The Planning Domain Definition Language*. 1998.

[NK00] X.L. Nguyen and S. Kambhampati. Extracting effective and admissible state space heuristics from planning-graph. In *Proc. AAA1-2000*, pages 798–805, 2000.

[NK02] X.L. Nguyen and S. Kambhampati. Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *Artificial Intelligence*, 135(1-2):73–123, 2002.

[VR99] V. Vidal and P. Régnier. Total order planning is better than we thought. In *Proc. AAAI-99*, pages 591–596, 1999.

[WAS98] D.S. Weld, C.R. Anderson, and D.E. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *Proc. AAAI-98*, pages 897–904, 1998.