Optimal Heuristic Planning using Search Space Splitting

Yacine Zemali and **Patrick Fabiani**
ONERA / DCSD - Centre de Toulouse
2 avenue Edouard Belin
BP 4025, 31055 Toulouse cedex 4 - FRANCE
{zemali, fabiani}@onera.fr

Abstract

Heuristic search has been widely applied to classical planning and has proven its efficiency in finding a "good and feasible" plan as quickly as possible. Finding an optimal solution remains an NP-hard problem for which even the computation of an informative admissible heuristic remains a challenge. Compared to already proposed approaches, the originality in this paper is to compute an admissible heuristic by taking into account, via search space splitting, a controlled amount of dependences between sub-sets of the reachable space. Search space splitting consists in reintroducing a controlled tree structure in a disjunctive plan graph. The splitting strategy, that selects which actions should split the search space, is automatically determined in a pre-processing stage. Experiments are presented and discussed.

Keywords : domain-independent classical planning, search space splitting, heuristic planning, disjunctive planning.

Introduction

Heuristic search has been widely applied to classical planning and has proven its efficiency in finding a "good and feasible" plan as quickly as possible. Finding the optimal solution (either in terms of plan length or of other criteria) remains an NP-hard problem (Bylander 1994). Complete and optimal planners require an exponential computation time in the worst case, while a number of heuristic planners propose efficient approximative schemes (Hoffmann 2002): those planners are often neither complete, nor optimal. GraphPlan (Blum & Furst 1997) is complete and optimal: it finds the shortest parallel plan in `STRIPS` problems. However, it can be interpreted as a heuristic planner (Bonet & Geffner 1999): GraphPlan performs an approximate reachability analysis by neglecting some constraints, except for the binary *mutex* relations. In terms of heuristic search, the planning graph building phase allows to assess a **lower bound of the length** of this shortest parallel plan. The computed heuristic is admissible, but less informative than the non-admissible heuristic used in (Hoffmann 2001). Good heuristics can generally be computed by solving a relaxed problem, but it may be difficult to take into account enough constraints with a quick computation method: the relaxed problem should not make too strong assumptions about the independence of subgoals. There seems to exist a **compromise** to be achieved between computational efficiency and quality, namely **admissibility and accuracy**, of the heuristic. We propose and formalize *search space splitting* as a general framework allowing to neglect or take into account a **controlled** part of the problem constraints (e.g. negative interactions) (Zemali, Fabiani, & Ghallab 2001), and use it in order to compute a variety of *informative admissible* heuristics, then applicable in an A^* informed search algorithm for **optimal planning**.

We first present related recent developments in heuristic planning. We formalize *State space splitting* as a flexible planning scheme and present how to use it in order to compute a range of *informative admissible heuristics*. Finally, we compare *splitting strategies* in terms of the computational cost of obtaining an informative admissible heuristic and also in terms of the benefit obtained from using that heuristic at search stage with an $A*$ algorithm.

Issues

A planning heuristic can be computed by considering a relaxed problem that is much quicker to solve. `HSP` (Bonet & Geffner 1998) computes its heuristic by assuming that subgoals are independent: it does not take any interaction into account. `HSPr` (Bonet & Geffner 1999) takes some negative interactions into account while computing its heuristic and uses a WA^* algorithm performing a backward search from the goal to find a valid plan. `HSPr*` (Haslum & Geffner 2000) computes an admissible heuristic using a shortest path algorithm. It finds an optimal solution using an IDA* algorithm to search the regression space. `HSP 2.0` (Bonet & Geffner 2001) is a planner that allows to switch between several heuristics with a WA^* search algorithm. `FF` (Hoffmann 2001) uses a relaxed GraphPlan (without mutex propagation nor backtrack) to compute an **approximation of the length** of the solution, i.e. the length of a solution to the relaxed problem: this heuristic takes into account all positive interactions, but no negative effects: therefore it is **not a lower bound** and **not an admissible heuristic**. The search is performed using an *enforced hill-climbing* algorithm, switching to WA^* in tougher problems.

Inferring state invariants can help reducing the planning complexity: invariants are properties that hold in all the states that are reachable from an initial state. (Rintanen 1998) propose a polynomial iterative fix-point algorithm to infer 2-literals invariants, such as permanent mutex relations. TIM (Fox & Long 1998) computes GraphPlan's permanent mutex relations as a pre-processor for `STAN` (Long & Fox 1999) : it generates finite state machines for all the possible transitions for each single object in the domain. Yet, the benefit of pre-computing permanent mutex relations appears as not significative enough in GraphPlan-like approaches, such as in (Fox & Long 2000) or (Meiller & Fabiani 2001). A generalization to other invariants is proposed in (Rintanen 2000) and in the `DISCOPLAN` package (Gerevini & Schubert 2000) which uses syntactic properties of the operators to verify whether the truth of an invariant is preserved. Such invariants or properties of the domains can rather be used in order to speed up heuristic computation. "Hybrid" planners efficiently combine both approaches : `AltAlt` (Nigenda, Nguyen, & Kambhampati 2000) computes its heuristic using the planning graph produced by `STAN`, and searches from the goal with `HSPr`'s regression algorithm.

States and facts

The duality between state-based and fact-based representations is classical: a planning problem can be described either with a model based on enumerated states or with a model based on facts that can be true of false. Both representations characterize the same state space: enumerating all the possible combinations of truth values for the facts gives the set of all possible states. Each fact corresponds to the set of states in which it is true. Each list of facts corresponds to the set of states in which all the facts are true (the intersection of the sets of states corresponding to each individual fact).

State-based model

In a state-based model, actions are defined as transitions between individual states: in each state, an action can either be executable, leading to another state $\omega \longrightarrow^{a} \omega'$, or not. The state-based model of a planning problem is defined by:

- a finite possible states set Ω: $\Omega = \{\omega_1, \omega_2, \ldots, \omega_n\}$
- a finite possible actions set $\mathcal{A}$: $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$
- a transition function $T_s : \Omega \text{x} \mathcal{A} \longrightarrow \Omega \cup \{\star\}$ associating to each couple (state, action) (ω, a):
  - either an impossibility symbol $\star$ if the action a is not executable in the state ω: $T_s(\omega, a) = \star$
  - either the state $\omega' \in \Omega$ resulting from the execution of action a in state ω : $T_s(\omega, a) = \omega'$.

Fact-based model

In a fact-based model, actions are defined as operators that make some fact true and other fact false, provided a number of facts that have to be true for the action to be executable. Each action is defined by :

- a list of preconditions, that is to say the facts that have to be true for the action to be applicable ;
- a list of deletion, that is to say the facts that are made false by the application of the action ;
- a list of creation, that is to say the facts that are made true by the application of the action .

An operator defines a macro-transition from a list of facts (a set of states) to another list of facts (another set of states): $\{f_0^0, f_1^0, \ldots, f_f^0\} \longrightarrow^{a} \{f_0^1, f_1^1, \ldots, f_{f'}^1\}$.
The fact-based model of a planning problem is defined by:

- a finite set of facts describing the properties of the possible states $F = \{f_1, f_2, \ldots, f_k\}$
- a finite possible actions set $\mathcal{A}$: $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$
- a transition function $T_f : 2^F \text{x} \mathcal{A} \longrightarrow 2^F \cup \{\star\}$ associating each couple (facts list, action) $(\{f_0^0, f_1^0, \ldots, f_f^0\}, a)$:
  - either to an impossibility symbol $\star$ if a's preconditions do not hold in $\{f_0^0, f_1^0, \ldots, f_f^0\}$: $T_f(\{f_0^0, f_1^0, \ldots, f_f^0\}, a) = \star$
  - either to the list of facts resulting from the execution of action a in the states where facts $\{f_0^0, f_1^0, \ldots, f_f^0\}$ are true : $T_f(\{f_0^0, f_1^0, \ldots, f_f^0\}, a) = \{f_0^1, f_1^1, \ldots, f_{f'}^1\}$.

Search space

The planning can either be done from the problem definition or after an exploration phase that builds the graph of the *search space*. Depending on how it handles facts or states and how it develops the transition relation between them, a planner will not build the same search space nor explore it in the same way. The classical duality between facts and states is the basis of our formalization of a variable state space splitting in the building and exploration of the reachable states at planning time.

Tree search space at state level

In a tree search space, all the nodes n of the explored tree correspond exactly to individual states $n = n_\omega$. The nodes are connected by the possible transitions between states. The transition relation TR_s between the explored nodes in an tree search space is directly given by the transition function T_s of the state-based model. $n_{\omega'}$ is the successor of n_ω in the search tree and in the transition relation TR_s, if and only if there exists an action that realizes a transition between these two corresponding states. Developing such a transition relation gives the tree of a *Forward State Space* search.

State-based transition relation

$$(n_\omega, n_{\omega'}) \in TR_s \Leftrightarrow \exists a \in \mathcal{A} / T_s(\omega, a) = \omega'$$

Disjunctive search space

In a disjunctive planning approach, the nodes of the *search space* are the levels, which correspond to the notion of depth in a tree: there is only one node by depth. The successor node $\mathcal{L}_{k+1}$ at level $k+1$ is the list of facts $\{f_0^{k+1}, f_1^{k+1}, \ldots, f_{f'}^{k+1}\}$ built by developing all the transitions that are applicable from the list of facts of its predecessor node $\mathcal{L}_k = \{f_0^k, f_1^k, \ldots, f_f^k\}$ at level k (taking some mutexes into account). The transition relation TR_f is deduced from the transition function T_f of the fact-based model by neglecting some constraints :

Disjunctive transition relation

$$(\mathcal{L}_k, \mathcal{L}_{k+1}) \in TR_f \;\Leftrightarrow \forall f_{i'}^{k+1} \in \mathcal{L}_{k+1},\; \exists a \in \mathcal{A},$$

$$\exists \{f_{i_0}^k, f_{i_1}^k, .., f_{i_f}^k\} \subset \mathcal{L}_k \Big/ \; f_{i'}^{k+1} \in T_f(\{f_{i_0}^k, f_{i_1}^k, .., f_{i_f}^k\}, a)$$

Search space splitting

The idea of *search space splitting* is to reintroduce a tree structure in a disjunctive search by grouping the facts at each level k in different nodes instead of one undistinguished list of facts $\mathcal{L}_k$. For that purpose, it is necessary to *split* the level into different **nodes** constitutive of the tree structure.

This can be done in many ways over the set $2^{\mathcal{L}_k}$ of subsets of $\mathcal{L}_k$, and in particular the splitting *need not* be a partition. On the contrary, useful splitting strategies are more likely to let each fact belong to a list of nodes, or classes:

Classes : at each level k, the facts of $\mathcal{L}_k$ are grouped into a finite number of subsets that we call *classes* $C_i^k \in 2^{\mathcal{L}_k}$, with $i \in I$ as indices. Each class C_i^k in a level k correspond to a **node** $n_{C_i^k}$ of depth k in the tree of the search space.

The set of all classes at a level k is denoted $\mathcal{C}^k = \{C_i^k\}_{i \in I}$. $\mathcal{C}^k$ is a **covering** of $\mathcal{L}_k$. A *splitting function* $\mathcal{F}_{split}^k : \mathcal{L}_k \longrightarrow 2^{\mathcal{C}^k}$ describes how facts are grouped into *classes*, associating each fact $f \in \mathcal{L}_k$ to the set of classes in $\mathcal{C}^k$ it belongs to: $\mathcal{F}_{split}^k(f) = \{C_{1,f}^k,\ C_{2,f}^k,\ \ldots\ C_{l,f}^k\}$
Such a definition is descriptive rather than constructive.

The **splitting strategy** defines the rules for creating, propagating or merging classes, thus associating each class at a given level with its successor classes at the following level. In that sense, the splitting strategy is the "branching rule" of the tree structure of the search space. The splitting strategy can be defined either as $k - dependent$ or not.

Classes are defined while building the search space as a tree, necessarily combining the splitting strategy and the transition relation T_f: as a matter of fact, the splitting also depends on actions. Some classes of the level k are simply propagated to the level $k+1$ just in a similar way as facts can be propagated by *no-ops* in *GraphPlan*.

Splitting strategy: $\forall \mathbf{k},\ \mathcal{S}_{\mathbf{split}}^{\mathbf{k}} : \mathcal{C}^{\mathbf{k}} \mathrm{x} \mathcal{A} \longrightarrow \mathbf{2}^{\mathcal{C}^{\mathbf{k+1}}}$

$$\forall (C_i^k, a),\ \ \mathcal{S}_{split}^k(C_i^k, a) = \{C_{i,1}^{k+1,a}, C_{i,2}^{k+1,a}, .., C_{i,m^i}^{k+1,a}\}$$

Once such subclasses have been created, it may be useful to merge them according to some coherence-based, or similarity-based **merging strategy**, e.g. merging $C_{i,n^i}^{k+1,a}$, $C_{i',n^{i'}}^{k+1,a'}$,.., $C_{i^l,n^{il}}^{k+1,a^l}$ into a single new coherent class renamed C_j^{k+1} in the resulting set of classes of level $k+1$. All the images $(T_f(C_i^k))$ through T_f of the sets of facts C_i^k at level k are by construction covered at level $k+1$ by $\mathcal{C}^{k+1} = \{C_1^{k+1},\ C_2^{k+1},\ \ldots\ C_m^{k+1}\}$ and thus: $\forall i,\ T_f(C_i^k) \subset \bigcup_{j=\{1,\ldots,m\}} C_j^{k+1}$.

The resulting transition relation in the search space connects two nodes, i.e. two classes of successive levels, if and only if, for all facts in the arrival class:

- there exists an action whose preconditions are all in a same departure class and that has this fact as an effect in the arrival class;
- the application of this specific action in the departure class effectively leads to the arrival class according to the splitting strategy.

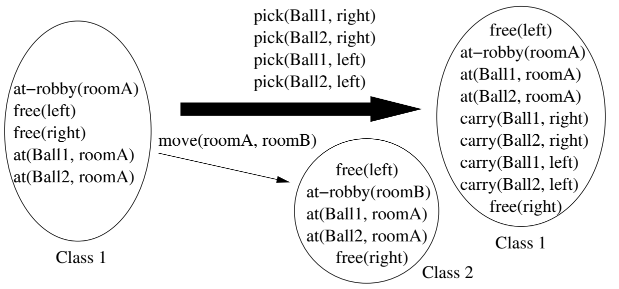

Figure 1: Splitting process for the gripper domain

This transition relation not only requires that the facts in each class are connected by the fact-based transition relation T_f, but also that this is coherent with the splitting strategy. In practice, classes and facts are coherently managed by a same algorithm.

Splitted transition relation:

$$(n_{C_i^k}, n_{C_{i'}^{k+1}}) \in TS_f \;\Leftrightarrow \forall f_{i'}^{k+1} \in n_{C_{i'}^{k+1}},\; \exists a \in \mathcal{A},$$

$$\exists \{f_{i_0}^k, f_{i_1}^k, .., f_{i_f}^k\} \subset n_{C_i^k} \Big/ \; f_{i'}^{k+1} \in T_f(\{f_{i_0}^k, f_{i_1}^k, .., f_{i_f}^k\}, a)$$

Getting informative admissible heuristics

In GraphPlan, the number of levels of the planning graph (when the goals are first reached) may be a very poor estimation of the exact length of the solution: in this building phase, GraphPlan relaxes a number of constraints. It groups together incompatible facts into sets (levels) that contains a number of incoherences.

On the contrary, a full splitting approach (a tree exploration) will separate each individual state during the graph construction. The depth of the planning tree when the goals are first reached, will give the exact length of an optimal solution. GraphPlan's approximation is too coarse (too short). A tree exploration will compute a perfect heuristic, but in an exponential time.

We use *state space splitting* in order to achieve a **compromise** between those two extremes: our heuristic is classically computed by solving a relaxed problem, i.e. having neglected some constraints, but uses the number of levels of the *splitted planning graph*.

Splitting the planning graph allows us to better approximate the real plan length (see also (Kambhampati, Parker, & Lambrecht 1997)), taking more negative effects into account in a controlled way : a number of incoherent sets of states can be splitted, and more constraints of the problem can be taken into account. As a consequence, some operators will not be introduced as early as in a GraphPlan-like building phase. Therefore, the length of the planning graph (when it first reaches the goals) is **increased** getting closer to the real optimal plan length. Yet, we are still underestimating the exact plan length because not all the constraints are taken into consideration. The length of the splitted planning graph is **a more informative admissible heuristic**.

Splitting on the number of permanent mutexes

Our splitting strategy is based on mutual incoherence, or interactivity, between facts. At each level, facts are grouped according to their level of mutual coherence with respect to the problem constraints: facts created by operators that are not mutex with each other can be grouped in a same class. In practice, operators with a big interaction degree (this is the number of instantiated operators with which the current operator is permanently mutex) are marked as "class creator".

A number of algorithms allow to compute invariants features such as permanent mutexes in a planning problem. We designed an algorithm, devoted to the detection of permanent mutex relations, which considers facts as resources (Meiller & Fabiani 2001): if two operators destroy the same fact, they are mutex because they try to consume the same resource, which is not possible. Our algorithm takes advantage of this by coloring the facts. Two facts with a common color are mutex because they result from the consumption of a common resource. This marking allows to know if two facts need competitive resources to be produced. The two first steps of the algorithm are to determine the number of colors per fact, and to create the initial colors which will be used to create new color combinations. The creation of the first (primary) colors is deduced from the "initial state" of the problem. Thus, we can give a (or several) different color to each fact of the initial state because those facts are not permanent mutex between each other. Our algorithm then propagate the primary colors until all facts are colored. We do not detail the propagation process. From the mutex relations between facts, we deduce the mutex relations between operators to compute the interaction degrees.

Implementation details

In our planner, the user has to provide a splitting level. This is a percentage which determines how many operators will be *class creators*. Those operators are then determined by the *splitting strategy*.

At each call of the heuristic function h, the algorithm needs to assess sets of states. During the successive calls, the algorithm may assess several times the same set. To avoid this, we need to store intermediate valuations of sets obtained during h computations. A set of states is described by *facts*: if n is the number of facts of the problem, each set is described by a vector $v \in \{0,1\}^n$, where $v[i]$ gives the truth value of the i^{th} fact. The vectors are stored in a *decision tree* (DT) where leaves contain the h intermediate valuations. Formally, we have $DT : \{0,1\}^n \rightarrow \mathbb{N}$. Thus, we can read and write intermediate values in a constant time (n). This mechanism allows us not to recompute h from scratch.

If during h computation, the algorithm reaches a branch where it has to assess a set which is already being assessed at a lower depth, the branch will be cut. To detect such branches, we also use a method based on *decision trees*.

Thanks to those mechanisms, our planner behaves like a dynamic programming algorithm when used with a *full splitting*. This allows to study a continuum between the resolution with a relaxed GraphPlan heuristic (*no splitting*) and with a dynamic programming approach. With the intermediate splitting levels, more or less constraints are taken into account in the heuristic computation.

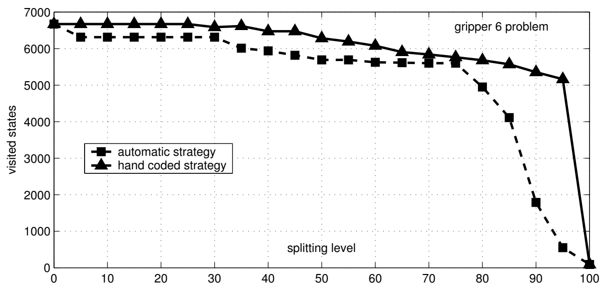

Figure 2: visited states for the gripper problem: instance with 6 balls

Results on the Gripper domain

We solved this problem with two splitting strategies: the automatic one, described above, and a hand coded one, in order to study the impact of the strategy. We tried a range of splitting percentages from 0% (then the heuristic computation is equivalent to a relaxed GraphPlan) to 100%. As expected, the number of visited states decreases when the splitting level increases. Figure 2 shows the relation between the splitting level and the number of visited states for both strategies. The automatic splitting strategy (dashed edge) is deduced from the pre-computed permanent mutexes. The operators with the greatest interaction level are the *"move"* operators. As a result, they are preferentially "class creators" (fig 1). Our hand-coded strategy does exactly the opposite: *"pick"* and *"drop"* operators are preferentially responsible for splitting. The automatically deduced strategy visits the search space in a more efficient way. This is due to the fact that when we do not split on *move* operators, we propagate a lot of incoherence, because *move* operators are very interactive and generate numerous mutex relations.

Results on the Blocks World domain

This time, the automatic strategy is equivalent to a *random splitting* because each operator has the same interaction degree. As a result, there is no preferentially chosen operator to be responsible for splitting. By hand, we privileged the *"pick"* operators to be responsible for splitting. The hand coded strategy is here the best one. This is because this strategy manages to capture l-literals constraints between facts with $l \geq 2$ (*mutex* only correspond to $l = 2$).

Concerning overall CPU time, on a five blocks instance, we can see that the two extremes (relaxed GraphPlan and dynamic programming) give the best results (fig 3). For a splitting level between 0% and 50%, the number of visited states decreases, but the heuristic computation time increases: we do not have any benefit from the splitting. From 50%, the overall CPU time starts to decrease. This is due to our intermediate results storage. With such splitting levels (50% to

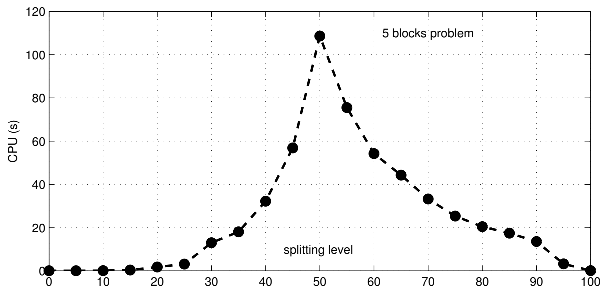

Figure 3: CPU time for the 5 blocks problem

100%), the heuristic becomes very informative. The search algorithm then focuses in one direction. Therefore, previous stored results will be reused. With too small splitting levels, there are too much changes of direction for the intermediate storage to be used efficiently. We verified this intuition empirically by studying the ratio *number of elements stored in the DT / number of elements read in the DT*. This ratio reflects the utility of our storage. For the 5 blocks problem, it remains lower than 1 up to 50%, it then slowly increases from 1 to 3 for a 100% splitting.

Conclusion and Perspectives

We proposed and formalized *search space splitting*. We presented its application to compute admissible heuristics for optimal planning. The *search space splitting* aims at increasing the quality of the heuristic. The more we split, the more the heuristic gives a length close to the reality. One of our future work is to optimize the splitting level in order to minimize the global complexity of the planning process: this level is a parameter given by the user, and we are working on how to deduce it automatically.

We presented an automatic splitting strategy which appears to be efficient (in terms of reducing the size of the search space) in the general case. However, the blocks-world example proves us the existence of better strategies in certain cases. We are working on a way to automatically adapt the splitting strategy to the domain.

Our method is efficient with extreme splitting levels (close to 0 or 100) on small dimension problems. With bigger dimensions, too small splitting levels give a poor estimation, thus leading the search algorithm to make a too vast exploration. Our first experiments show that problems with high dimensions should be tackled with a high splitting level. Of course, we can not handle those problems with a full splitting, because it would be equivalent to solve them with a dynamic programming algorithm, which is not applicable with huge dimensions. Quite high splitting levels (60 to 90) could relax the problem enough to break the complexity while giving an informative heuristic. The search algorithm would then concentrate its exploration on a few privileged directions, because it is well informed. But, this leads to storage problems: there are too much intermediate results to store in the DT. To handle this, we envision using tree reduction methods issued from the MTBDD[1] domain.

We expect those methods to help us dealing with tougher problems: the splitting provides a mean to control the accuracy of the heuristic which could help to explore greater search spaces in an efficient manner, or to deal with optimization criteria richer than the plan length (`PDDL 2.1`).

References

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

Bonet, B., and Geffner, H. 1998. Hsp: Heuristic search planner. In *Planning Competition of AIPS-98*.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *ECP-99*, 360–372.

Bonet, B., and Geffner, H. 2001. Heuristic search planner 2.0. *AI Magazine* 22(3):77–80.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1-2):165–204.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in tim. In *JAIR*, 367–421.

Fox, M., and Long, D. 2000. Utilizing automatically inferred invariants in graph construction and search. In *AIPS-00*, 102–111.

Gerevini, A., and Schubert, L. K. 2000. Discovering state constraints in DISCOPLAN: Some new results. In *AAAI/IAAI*, 761–767.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *AIPS-00*, 140–149.

Hoffmann, J. 2001. FF: The fast-forward planning system. *AI Magazine* 22(3):57–62.

Hoffmann, J. 2002. Local search topology in planning benchmarks: A theoretical analysis. In *AIPS-02*.

Kambhampati, S.; Parker, E.; and Lambrecht, E. 1997. Understanding and extending graphplan. In *ECP-97*, 260–272.

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

Meiller, Y., and Fabiani, P. 2001. Tokenplan ; a planner for both satisfaction and optimization problems. *AI Magazine* 22(3):85–87.

Nigenda, R.; Nguyen, X.; and Kambhampati, S. 2000. Altalt: Combining the advantages of graphplan and heuristic state search. In *KBCS-00*.

Rintanen, J. 1998. A planning algorithm not based on directional search. In *KR-98*. 617–624.

Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In *AAAI/IAAI*, 806–811.

Zemali, Y.; Fabiani, P.; and Ghallab, M. 2001. Using state space splitting to compute heuristic in planning. In *PLANSIG-01*.

[1]Multi Terminal Binary Decision Diagram