

Approaches to Integrating Abstractions In Graphplan-based Planning Systems

Greg Barish
University of Southern California
Computer Science Department
CS 541 Final Project
December 11, 1998

In this paper, I describe various ways of combining abstractions and Graphplan-style of planning to accomplish highly efficient problem solving. Graphplan-based planners can efficiently construct solution spaces by focusing on the rapid construction of a planning graph. Abstractions represent a means for reducing and optimizing the search for a solution, by parceling complex problems into smaller sub-problems, which are easier and more efficiently solved. Combining both approaches thus provides a means for both rapid solution space construction as well as efficient search in that solution space. This paper relates three approaches to implementing abstractions in IPP, a Graphplan-based planner. One method involves modifying the internal search algorithm of IPP. Another approach involves creating a preprocessor for domain facts and operators, such that domain abstractions can be used to encourage or bias the IPP search in a favorable manner. Finally, a third approach involves implementing an abstraction hierarchy processor within IPP, which reduces a problem into a series of sub-problems and has IPP solve each of the sub-problems independently, for each abstraction level, by inferring relevant state information for these sub-problems through the mutual exclusion information IPP accumulates from previous problems.

1 Introduction

Graphplan-style planners are unique from other planners in how they identify solution spaces which contain valid plans to solve a particular problem. Conventional forward chaining planners iteratively apply operators and progressively search for a solution while creating a state space graph. In contrast, Graphplan rapidly creates a graph which represents the plan space and, when a newly added layer is found to potentially contain a solution to problem, employs backward chaining to identify the exact plan. To avoid cases where the simultaneous use of operators at a given step in a plan can lead to a violation of mutual exclusion, Graphplan inserts *no-op* operators as necessary.

Abstractions are another mechanism to improve the efficiency of planning, by reducing a large, complex problem into a series of smaller sub-problems and then solving those sub-problems independently. Intuitively, this approach seems near optimal: if domains can be constructed such that complex problems can be neatly broken into sub-problems, we can likely force the search algorithm to only consider the minimal number

of steps in a given sub-problem. Searches for solutions of individual sub-problems are indeed efficient because one needs to consider far fewer operators. Specifically, the idea of abstractions is to parcel a given complex problem into a set of independent “abstract spaces” wherein individual sub-problems can be identified and solved. Intermediate states at a given level in an abstraction hierarchy are used to identify the goal states in the next lower layer.

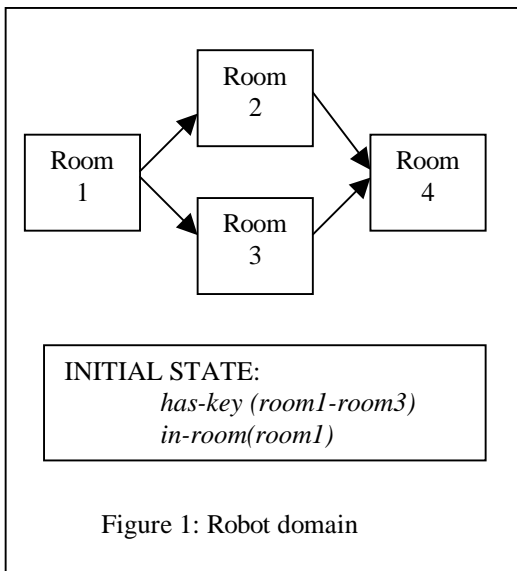
In this paper, I will describe how I explored three different methods for integrating abstractions in IPP [Koehler97], a Graphplan-style planner. First, in section 2, I will discuss three general challenges associated with implementing abstractions in any planning system. Section 3 describes the various approaches I took, and their different merits and drawbacks. In Section 4, I present results from two of these approaches, and in the remaining sections, I discuss possibilities for future work.

2 Challenges of Abstractions

There are unique challenges associated with implementing abstractions. One immediate issue has to deal with how to identify the proper abstract spaces for a given domain. Algorithms for accomplishing

this task are given in [Knoblock91], but as the author notes, there are questions as to the domain independence of such abstraction identification. In short, the question is: how easy is it to identify an abstraction hierarchy in problem spaces which do not easily lend themselves to such analysis (such as the Towers of Hanoi)?

Another issue with abstractions has to do with the need to occasionally backtrack between layers in the abstraction hierarchy. It is important to note that, in a global context, when a given sub-problem has multiple solutions, not all of these solutions will continue be valid. For example, consider a robot domain where there are three operators: *move*, *open-door*, and *unlock-door*. Also suppose that, as depicted in figure 1, the initial state consists of a robot which is in room1, has the *room1-room3* key, both *room1-room2* and *room1-room3* doors are locked, and the goal is to have the robot get to room4.



This problem may be thought of as containing two abstraction layers: one involving moving between rooms, the other involving the unlocking and opening of doors. When solving the moving-between-rooms sub-problems, a valid plan at that level involves moving from room1 to room2 to room4. However, during plan refinement at lower layers in the abstraction hierarchy, we find that moving from room1 to room2 – while possible at a higher abstraction layer – is not possible as we approach the ground state. We find this out because facts (or the lack of facts) at lower abstraction layers indicates that

we cannot refine the plan in at that lower layer: specifically, in our example, it is impossible to open the door which connects room1 to room2 since we do not have a key. Still, it is obvious that this inability to refine at a lower layer can be resolved by going back to the upper layer in the hierarchy and choosing a new solution (*move-room room1 room3*) that can then be refined properly at lower layers. This repair process is known as *hierarchical backtracking* and it is important that any implementation of abstractions support such a mechanism.

A final problem associated with implementing abstractions has to do with the nature of the underlying planning system, and whether it produces totally ordered or partially ordered plans. With total ordered plans, which was the case for PRODIGY (on which Hierarchical PRODIGY, or H-PRODIGY, was implemented), it is easy to identify the intermediate steps between an initial state and a goal. This is important because these intermediate states serve as the basis for the goal states of the lower abstraction spaces, and thus their ordering can be critical. Since Graphplan (and IPP specifically) produce partially ordered plans, what are the implications of enforcing some sort of total order on these partially ordered results? Or, can we somehow maintain the partial order property of these planners and simply merge in refinements of parallel tasks at a common time point?

For example, in the robot domain described earlier, if some abstract plan consisted of the robot unlocking a door with 2 keys (no matter as to the order of applying the keys, the door only opens when both keys have been used), can we refine the plan such that picking up both keys can also be specified in a partially ordered manner (no matter as to which order the keys are picked up, as long as they are both picked up)?

3 Approaches to Abstractions in IPP

For my project, I explored three different approaches to implementing abstractions in IPP, each of which had its advantages and disadvantages, which I will now describe.

Throughout this section, I will refer to the Towers of Hanoi example, since it is the domain on which I did all of my testing. The Towers of Hanoi is an attractive problem to demonstrate abstraction processing since it contains all of the attractive properties for abstraction identification [Knoblock91]. The Towers of Hanoi 3x3 problem refers to the idea of moving 3 disks from peg 1 to peg 3, as shown in Figure 2a. Likewise, as shown in Figure 2b, the Towers problem 4x4 refers to a case where there are 4 disks and 4 pegs.

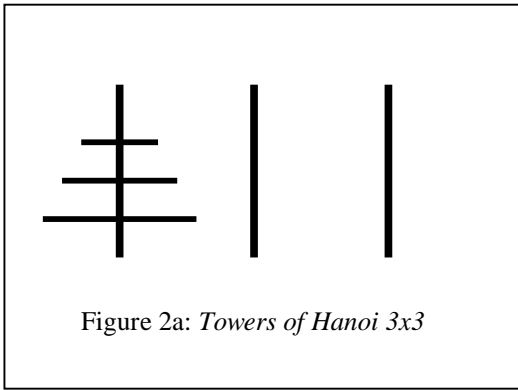


Figure 2a: *Towers of Hanoi 3x3*

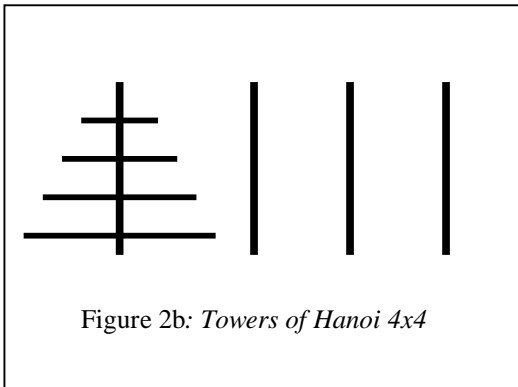


Figure 2b: *Towers of Hanoi 4x4*

3.1 Re-engineering IPP Search

The initial approach I took was to attempt to re-engineer the search algorithm in the IPP source code.

Currently, IPP works by progressively creating layers of a planning space graph until it identifies that it has enabled all of the goal conditions. Once it reaches this state, it attempts to produce a valid plan by backwards chaining among the operators in the plan space. When it cannot form a valid plan (usually because of mutual exclusions between operators), it continues to append layers to the planning space graph and revisits the search for a plan at each successive layer. As [Blum95] points out, the “leveling off” nature of appending layers to the planning graph does ensure that planning is complete, sound, and that it will indeed terminate when no solution can be found.

In examining the IPP search in more detail, the search for a valid plan at a given state of the planning graph is accomplished in a breadth-first manner. Thus, IPP recursively tries to satisfy all of the operators at a given time step, only moving backwards to the next time step after identifying which of the potential conditions at

an earlier time step will enable the operators at the current (later) time step.

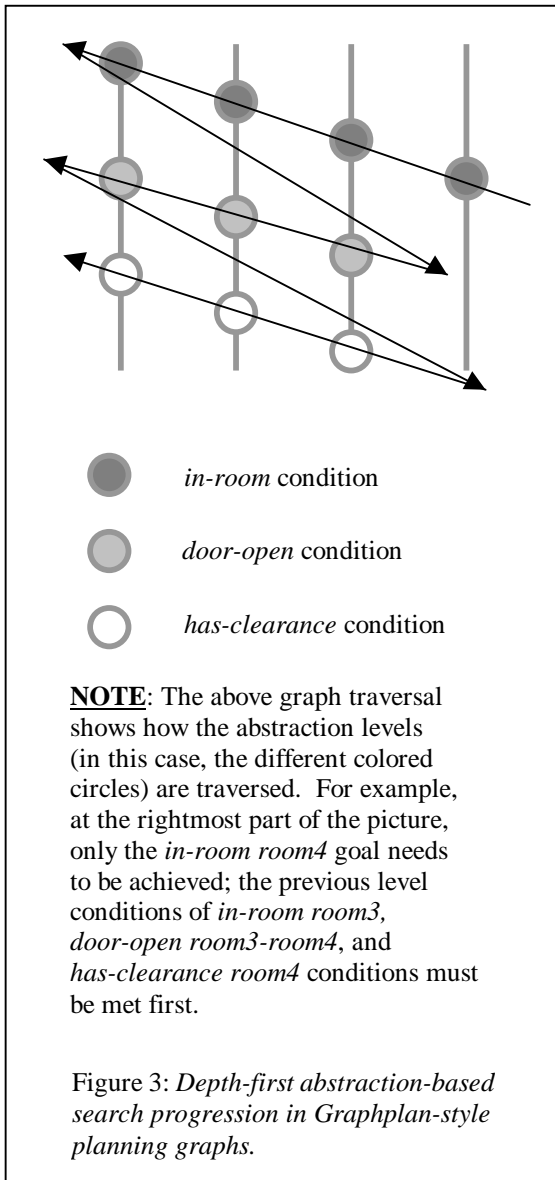
This breadth-first style of search is counter-productive when implementing abstractions, since the desired effect is to completely solve a problem at a given abstract layer n before solving sub-problems for layer $n-1$. Thus, a depth-first search would be more natural, where IPP would, for example, have to solve the moving of disk 3 from peg 1 to peg 3 before moving on to solving the sub-problems associated with moving disks 2 and 3 both from peg 1 to peg 3. In effect, this means starting the search for a solution to level 3 at the latest graph layer and then returning to that layer when beginning the search for solutions to the sub-problems of the next lower layers. In short, this leads to iterated calls to the IPP processing at the right-most (latest time) layer, until each level in the abstraction hierarchy has been processed.

However, there are still problems associated with that approach. Recall that Graphplan style planners may identify a situation at time step m where the goal conditions are reached, but a valid plan turns out to not be achievable (hereafter, I refer to this as the “false alarm effect”). Eventually, Graphplan will locate a valid plan – if there is one – at some later time step $m+c$. However, this property of Graphplan planners may in fact lead to more search than is really necessary when implementing abstractions, as well as raise the complexity of implementation.

Consider, for example, what can happen in the Towers of Hanoi 3x3 problem. For each layer in the planning graph, IPP will attempt to identify that the goal conditions can be reached. In fact, this first occurs at time step 4. At this point, IPP will try to locate a valid plan. However, since the integration of abstractions will effect not only how the search is accomplished, but the order of which pre-conditions are satisfied before others, such situations may result in more searching than would otherwise be necessary in “standard IPP”. This is because mutual exclusions which may be quickly obvious when searching breadth first become more slowly revealed when searching in an iteratively depth-first manner. In general, the implementation of abstractions in Graphplan planners needs to avoid these false alarm situations without incurring overly wasteful additional search.

Assuming that this can be accomplished, the next step is to consider the complexity associated with plan refinement. Recall that the nature of abstraction processing is to solve sub-problems at a higher layer and then use the intermediate states of these solutions as input for the goal states of lower layers. When thinking about this in terms of search in Graphplan

planners, it becomes apparent that the search will reveal a movement like that depicted in Figure 3:



Essentially, the search will move from right to left (does this once, only to establish the solution for the outermost abstraction layer), and then from left to right, iteratively among the abstraction layers, until a plan is found. But, such searching also needs to take into account hierarchical backtracking, which means that, during refinement, choosing a different operator at a higher level may be necessary, thus interrupting the normal flow of search. While this is certainly achievable, it does increase the complexity of such a solution.

Still, the overall approach appears sound and does include a method for dealing with backtracking. Unfortunately, at the time of this writing, I am still investigating an easy way to implement this without re-engineering too much of the search (since IPP includes several optimizations to search which may assume that its recursive, breadth first approach is maintained).

3.2 Developing a Domain Preprocessor

Another, less complex approach to implementing abstractions in IPP, is to pre-process the domain fact files so that (a) preconditions of various operators are re-ordered in terms of abstraction relevance, (b) the instantiated operators themselves are ordered by abstraction level, and (c) the goals are reordered by abstraction relevance.

We can rely on the natural feature of IPP that involves its search for potential operators which fulfil a given condition. If we re-order preconditions and instantiated operators, we can implicitly induce IPP to both address conditions in the order of abstraction space, and also choose operators - in a similar order - to enable those conditions. This has the overall effect of “guiding the IPP search” with a bias towards abstraction levels. Finally, altering the goal ordering is much like altering the pre-condition ordering, in the sense that induces a bias as to which operators are considered first.

Still, without modifying the fundamental breadth-first search nature of IPP, the pre-processor approach will only have a coarse-grain effect of improving the search. We will not be able to solve individual sub-problems, like we would want to in a pure abstractions-based approach, because IPP still sees all facts of the problem at all graph layers, albeit in a different, optimal order.

I was able to implement such a preprocessor, and did find many of the desired effects. These results are described in section 4. Assuming that abstractions can be automatically identified, such a pre-processor should be able to be domain-independent.

3.3 Emulating Hierarchical Prodigy

A final approach to implementing abstractions in IPP had to do with essentially emulating the way H-PRODIGY processed abstractions. This process consisted of simply iterating through the abstraction space levels, solving the individual sub-problems at that level and continuing the process on downwards until all of the abstraction spaces had been exhausted. Then, an aggregate plan could be formed by concatenating the plans for the sub-problems together.

H-PRODIGY also supported the ability to backtrack, which as mentioned earlier, is essential when implementing abstraction processing.

When looking at the details of what needs to be done to solve abstract problems at a given level, one should realize that essentially what is happening is that a planner is given a series of individual planning problems, each with their own initial state and goal state, and asked for a solution. The initial state and goal states must be accurately defined in order to produce a valid plan. For example, consider the first goal in level 2 of the Towers of Hanoi problem, where the goal is to get disk3 to peg3, starting with an initial state of *on-peg disk2 peg1* and *on-peg disk3 peg1*. Notice that it would be a mistake to communicate *the entirety* of what the picture implies about the goal state, namely: *empty-peg peg1* and *empty-peg peg2*. If that were the case, there would be no way for the planner to solve the sub-problem at level 2 of the hierarchy (no way for 2 pegs to be empty and disk3 to be the only disk on a peg when there exists more than 1 disk in the system).

Upon closer examination, the most important facts to retain from the goal state are (a) *empty-peg peg1* and (b) *on-peg disk3 peg3*. When thinking about the abstract space above (the disk3 space), this makes sense, since it would only intuitively seem relevant to describe what has changed since the initial state, namely that peg1 is now empty and the disk3 is on peg3. The point being made by this example is that the determination of initial and goal states is both critical and complex, especially when those states are based on the results of intermediate states from higher layers in the abstraction stack. H-PRODIGY circumvented this problem by inducing PRODIGY to be in the given state desired, simply by re-applying the operators returned for the previous plan (or higher layer plan).

State-inducement is not as easy in IPP. The internals of the code are such that other data structures get in the way of simply applying a known set of operations to get the planner in a given state, and then asking to go further, to yet another state. Recall that Graphplan planners are not totally-ordered, forward-chaining, state-space planners, so it is not easy, nor intuitive to “induce a state” for systems which based their decisions on plan graphs, not state graphs. Where state does come into play, however, is in

the initial and goal specifications. Consequently, I looked for an approach which could allow me to accurately determine these states for the various sub-problems.

The algorithm I developed reasons about the relevant state information for any given intermediate or goal state after a plan has been established for a particular problem, so that this state information could be recorded and then used for the next sub-problem (or those at lower layers). The algorithm takes advantage of internal IPP data structures which label mutual exclusive operations. Based on the data in these structures, it is able to derive the state at any given point in a resulting plan. The same data structures should be common to all Graphplan-style planners, since mutex processing is one of the key enabling mechanisms behind the way such planners locate valid plans in a given solution space.

The algorithm consists of the following:

```
FactList
Derive-The-State(
    FactList a_Facts,
    int      a_AbLevel)
{
    FactList newFacts = copyFacts($a_Facts);

    for i=1 to NumStepsInLastPlan {
        curFacts = TrueFacts(
            $FactTable[$a_AbLevel][$i]);
        newFacts = $newFacts +
            $curFacts -
            MutexFacts($curFacts);
    }

    return newFacts;
}
```

NOTE: TrueFacts() is needed since IPP contains fact tables for each time step in the plan, but also includes facts which were considered but not used - only TRUE facts should be used.

NOTE: MutexFacts() identifies those older facts which conflict with any fact in the Exclusive Set for each newly added fact.

Example:

Consider level 3 of the 3x3 Towers of Hanoi problem. Here, Derive-The-State() will be called to determine the goal state for the first sub-problem of level 2 as well as the initial state for the second sub-problem. The first call will be Derive-The-State(NULL, 3), which will result in processing

whereby newFacts will contain only the newly TRUE facts $\{empty\text{-peg } peg1, on\text{-peg } disk3\ peg3\}$, which is exactly what we want for the goal state for the first sub-problem of level 2. Derive-The-State(L2-Init-State-Facts, 3) will also be called so that the newly added facts $\{empty\text{-peg } peg1, on\text{-peg } disk3\ peg3\}$ will still be added, but they will be added to the existing list for the initial state (which also includes key information such as *bigger-than disk3 disk2*) and will mutex out facts which are no longer true, namely $\{on\text{-peg } disk3\ peg1\}$ and $\{on\text{-peg } disk2\ peg1\}$.

4 Results

As mentioned previously, results were only obtained for the preprocessing and H-PRODIGY emulation approach, since modification of the search algorithm in IPP proved to more complex than first envisioned.

4.1 Preprocessor Approach

The preprocessor for these experiments was written strictly for the Towers of Hanoi domain, but could be generically written for any domain, assuming that an abstraction hierarchy generator was also employed.

Three different factors of control were permitted by the pre-processor approach, allowing the user to re-order the operator preconditions (OPPRE), operator interleaving (OPINT), and modified goal ordering (GOAL). For purposes of this experiment, the last control was administered manually.

Results for these approaches on a 5 disk, 3 peg Towers of Hanoi problem, compared with normal IPP, produced the following results, shown in Table 1. Each configuration was measured against the logical opposite (most sub-optimal) configuration. For example, when measuring GOAL ordering effects, one case consisted of the goals labeled from *disk5* to *disk1* (optimal) while the other consisted of the goals labeled from *disk1* to *disk5*.

APPROACH	Best Planning Time (sec)	Worst Planning Time (sec)
OPPRE	138.24	140.11
OPINT	138.67	146.25
GOAL	139.25	603.33

Table 1: Planning Time (IPP vs GIPP)

Obviously, the most important factor had to do with goal-ordering. Interestingly, precondition re-ordering did not show as much of an effect. It would seem that precondition ordering would have the same or better effect than goal ordering, since it really amounts to goal ordering at each graph layer. However, IPP may do some optimized ordering of pre-conditions.

4.2 Hierarchical Approach

Under this approach, I compared a custom, H-PRODIGY-like emulation version of IPP (called "HIPP", for *Hierarchical IPP*) against standard IPP. I tested both versions against the 3x3, 4x4, and 5x5 Towers of Hanoi problem. The results appear in Tables 2a and 2b, below.

PROBLEM	IPP	HIPP
Towers 3x3	0.11	0.03
Towers 4x4	0.65	0.03
Towers 5x5	242.80	0.04

Table 2: Planning Time (IPP vs HIPP)

PROBLEM	IPP	HIPP
Towers 3x3	0.21	2.01
Towers 4x4	1.14	3.54
Towers 5x5	245.36	19.59

Table 2: Overall CPU Time (IPP vs HIPP)

NOTE: The goal ordering for the domains was optimal for the standard IPP runs.

As is obvious, the overall planning time is far more efficient than standard IPP (especially in the case where it only took 0.04 cumulative planning time versus over 200 seconds for IPP!). It should be noted that these results are probably slightly inaccurate, but not by much, since there is probably some round-off

error under 0.01 which is not reflecting the true cumulative planning time. Even so, the worst case planning time for the 5x5 problem will still be well under 1 second, which is 200 times faster than what normal IPP can produce.

Also of note is the fact that the overall time is better for standard IPP for simple problems, but far better for HIPP in complex problems. This is likely due to the fact that the increased overhead for HIPP, involving graph construction and data copying, on simple problems has a significant effect.

5 Future Work

Work should continue on analyzing how to re-engineer the IPP search so that a single planning graph can be used in combination with abstractions for efficient, minimal search. Although this a more complex route than the other approaches described, it is more elegant and achieves the goal of integrating abstractions with Graphplan-style planners.

It is also useful to explore applying the H-PRODIGY emulation approach to other domains, to see how well the state-reasoning algorithm holds up in domains which do not lend themselves so easily to abstractions. It would seem that derivation of state could be a generally useful function to call from a plan-space planner, if some other type of state-based analysis or optimization needed to be done at particular point in planning (just like plan step refinement).

While the results are also encouraging from the preprocessor approach, I see the main merits of this approach to be related to the coarse grain approach of goal/pre-condition reordering. Optimal orderings can vastly improve performance, but they will still reach an upper bound, constrained by the breadth-first nature of the Graphplan style of search during backwards chaining.

6 Discussion

In this paper, I have described options for integrating abstractions with Graphplan-based planning systems. Such integration is attractive because of the complimentary performance gains of each approach. Graphplan planning is quick largely because of how rapidly it can construct the solution (plan) space. Abstractions can be useful for efficiently searching that space.

Some of the issues involved in integrating abstractions with Graphplan include: support for hierarchical backtracking, automatic identification of abstract hierarchies in existing domains, and the approach towards iterative plan refinement under planning systems which generate partially ordered plans. The first two issues are native to the abstractions approach, in general, while the third issue is one which really is raised by the integration of these two approaches.

The three solutions described in this paper each have their strengths and weaknesses. Re-engineering IPP search seems like the most elegant solution, but requires heavy reorganization of the search code, since the nature of search through abstraction space is a nested depth-first, breadth first one, not strictly a breadth-first approach. The second approach, implementing a preprocessor, showed significant performance improvement, but such a solution appears to be a coarse grained approach and does not offer the ability to truly have IPP work on distinct sub-problems at various levels of abstraction. It merely guides IPP to do this type of processing, but cannot ensure it. Finally, the H-PRODIGY emulation approach (HIPP) was the most successful to date, in terms of performance, and contains some useful algorithms for reasoning about state at a given point in the solution search phase. However, iteratively calling IPP to solve these sub-problems may turn out to be less efficient and elegant than optimizing the search through the IPP plan graph, which would thus consist of the generation of only a single graph.

7 References

- [Knoblock91] Knoblock, C.A. *Automatically Generating Abstractions for Problem Solving*. PhD Thesis. Carnegie Mellon, 1991.
- [Blum95] K Blum, A. and Furst, M.L. *Fast planning through planning graph analysis*. Proc. IJCAI-95, Montreal, Canada.
- [Koehler97] Koehler, J. Nebel, B. Hoffman, J. and Dimpoulous, Y. Extending planning graphs to an ADL subset. In Proc. ECP-97, Toulouse, France, 1997.