# AltAlt-p: Online Parallelization of Plans with Heuristic State Search

Romeo Sanchez Nigenda & Subbarao Kambhampati
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
Telephone Number: 480 965 2735
Email: {rsanchez,rao}@asu.edu

ASU CSE TR02-002

**Abstract**

Despite their near dominance, heuristic state search planners still lag behind disjunctive planners in the generation of parallel plans in classical planning. The reason is that directly searching for parallel solutions in state space planners would require the planners to branch on all possible subsets of parallel actions, thus increasing the branching factor exponentially. We present a variant of our state search planner *AltAlt* called *AltAlt-p* which generates parallel plans by using greedy online parallelization of partial plans. The greedy approach is significantly informed by the use of novel distance heuristics that *AltAlt-p* derives from a graphplan-style planning graph for the problem. While this approach is not guaranteed to provide optimal parallel plans, empirical results show that *AltAlt-p* is capable of generating good quality parallel plans without losing also the quality in terms of the total number of actions in the solution at a fraction of the time cost incurred by the disjunctive planners.

Keywords:Domain-independent planning, scalability in planning, heuristic state search planning, parallel plans, and partial planning.

## 1 Introduction

In most of real world applications it is necessary to find optimal plans given a cost measure. Sequential plans can be seen as a special case of parallel plans, where each action is represented by an atomic execution. It is more realistic to represent plans that are optimized given a cost measure (e.g. time), and in which each step in the plan encodes a set of independent actions. The main motivation for this representation is that most real world scenarios could have a subset of non conflicting actions, and exploiting their concurrent execution to minimize time is an important requirement.

Despite their near dominance, heuristic state search planners still lag behind disjunctive planners in the generation of parallel plans in classical planning. The reason is that directly searching for parallel solutions in state space planners would require the planners to branch on all possible subsets of parallel actions, thus increasing the branching factor exponentially. *AltAlt* is based on two of the main approaches: disjunctive planners as exemplified by Graphplan[2], and heuristic state space planners exemplified by UNPOP[11], HSP[4] and HSP-R[3].

Graphplan based planners are able to support optimal parallelism by grouping as many as non conflicting actions at each time step of the search. In the case of heuristic state search planners, they have proved to perform very well in solving large deterministic planning problems[3, 4], but they have lagged behind disjunctive planners in generating parallel plans. However, we can use three main techniques to find such plans in state space search:

1. Search in the total space of parallel plans.

2. Post-Process a sequential plan in order to minimize the number of parallel steps.

3. Use a greedy technique to incrementally parallelize the plan.

The first approach is basically infeasible, because we would need to consider all partial subsets of applicable actions exponentially increasing our branching factor [12, 5]. The second approach is extensively discussed in [14], in which reorderings and deorderings of sequential plans are done offline to minimize the number of parallel steps. The main drawback of this approach is that it does not minimize the number of parallel steps given the overall problem, but only given the current plan. In other words, there is no guarantee that the given plan will encode the best ordering to minimize the number of parallel steps because it was not computed for that purpose.

Our approach falls in between, and aims to incrementally parallelize the partial plan during the regression search. The main advantage for this kind of approach is that it can obtain on the fly parallel information of the partial plan, and it does not incur in the cost of searching the whole space. Moreover, our approach is able also to maintain quality in terms of the total number of actions in the plan. We implement this in *AltAlt-p* which is based on the implementation of *AltAlt* planning system[8]. *AltAlt-p* uses a greedy approach that makes use of its heuristics to consider only a subset of the most promising parallel actions at each level of the search, iteratively adding more actions later, and rearranging the partial plan to maximize its parallelism. Empirical results show that our approach not only maintains the quality of the plans in terms of the number of parallel steps, but also in terms of the number of actions contained in the plan. *AltAlt-p* provides a good tradeoff between the quality based on the number of parallel steps in the plan, and length of the solution. This is partly because not only commits to parallelize the plan but also to insert the best possible actions at each stage of the search reducing the number of actions in the final solution. Because of this ability, *AltAlt-p* remains the only viable option where competing approaches can not even solve the problems. Our costs in terms of space and time for finding parallel plans with our approach are negligible. Results also show that *AltAlt-p* incurs very little additional overhead over *AltAlt.* Finally, we also show that the quality of the parallel plans generated by our approach is superior to that generated by a post-processing approach, but it is not optimal with respect to the Graphplan-based solutions.

In the rest of this paper, we discuss the implementation and evaluation of our approach to generate parallel plans with *AltAlt-p.* Section 2 starts by providing the background on *AltAlt* the base planner and explains how differs from *AltAlt-p.* Section 3 and 4 describes the generation of parallel plans in *AltAlt-p* and constraints this approach with alternate approaches (e.g. post-processing). Next, Section 5 presents extensive empirical evaluation of our implementation that demonstrates the effectiveness and limitations of our approximation. This section also presents experiments to study the quality and efficiency tradeoffs between Graphplan based planners and *AltAlt-p* in the computation of parallel plans. Finally, Section 6 discusses some related work and Section 7 summarizes our contributions.

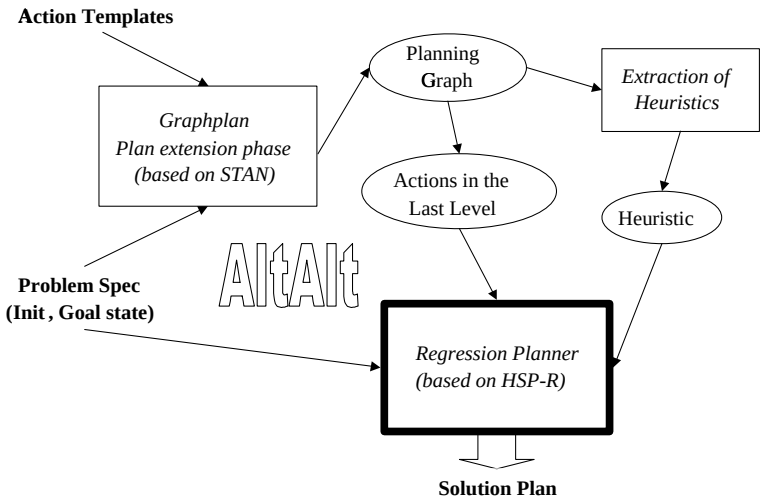



Figure 1: Architecture of *AltAlt*

## 2 Background on *AltAlt*

*AltAlt* system is based on a combination of Graphplan and heuristic state space search technology. *AltAlt* extracts powerful heuristics from a planning graph data structure to guide backwards the state search. The highlevel architecture of *AltAlt* is shown in Figure 1. The problem specification and the action template description are first fed to a Graphplan-style planner, which constructs a planning graph for that problem in polynomial time. We assume the reader is familiar with the Graphplan algorithm [2].This planning graph structure is then fed to a heuristic extractor module that is capable of extracting a variety of effective and admissible heuristics, based on the theory that we have developed in our recent work [8, 9]. This heuristic, along with the problem specification, and the set of ground actions in the final action level of the planning graph structure (see below for explanation) are fed to a regression state-search planner.

To explain the operation of *AltAlt* at a more detailed level, we need to provide some further background on its various components. We shall start with the regression search module. This module starts with the goal state and regresses it over the set of relevant action instances from the domain. An action instance $a$ is considered relevant to a state $S$ if the effects of $a$ give at least one element of $S$ and do not delete *any element* of $S$. The result of regressing $S$ over $a$ is then $(S\ eff(a)) \cup prec(a)$–which is essentially the set of goals that still need to be achieved before the application of $a$, such that everything in $S$ would have been achieved once $a$ is applied. For each relevant action $a$, a separate search branch is generated, with the result of regressing $S$ over that action as the new state in that branch. Search terminates with success at a node if every literal in the state corresponding to that node is present in the initial state of the problem.

The crux of controlling the regression search involves providing a heuristic function that can estimate the relative goodness of the states on the fringe of the current search tree and guide the search in most promising directions. The critical issue in designing heuristics is to take positive and negative interactions among subgoals into account. However, taking interactions into account in a principled way turns out to present several technical challenges. Fortunately, our recent work [8, 9] provides an interesting way of leveraging the Graphplan technology to generate very effective heuristics. In the

next subsection, we introduce one of the best heuristics.

### 2.1 $h_{AdjSum2M}$ Heuristics

One of the best heuristics according to our analysis in [8, 9] was a heuristic called $h_{AdjSum2M}$. We adopted this heuristic as the default heuristic in *AltAlt* and *AltAlt-p.* In the following, we briefly describe this heuristic.

The basic idea of $h_{AdjSum2M}$ is to adjust the sum heuristic [8] to take positive and negative interactions into account. This heuristic approximates the cost of achieving the subgoals in some set $S$ as the sum of the cost of achieving $S$, while considering positive interactions and ignoring negative interactions, plus the penalty for ignoring the negative interactions. The first component can be computed as the length of a "relaxed plan" for supporting $S$, which is extracted by *ignoring all the mutex relations.* And, the second component as the largest interacting value among any pair of subgoals in $S$. In summary, we have:

**Heuristic 1 (AdjustedHeuristic2M)** $h_{AdjSum2M}(S) := length(Relaxedplan(S)) + max_{p,q \in S}\ \delta(p,q)$

This heuristic not only provides quality in terms of the number of actions contained in the plan, but also in the final number of parallel steps. The computation of the heuristic captures the interaction among pairwise independent subgoals, providing useful information to the planner. As we will see in our evaluation Section 5 this heuristic provides a good tradeoff between quality in terms of the number of actions in the plan and the number of parallel steps, and efficiency.

One of the main differences between *AltAlt* and *AltAlt-p* is the computation of the heuristic. In *AltAlt* the heuristics were extracted from a serial planning graph[8], in which every pair of non-noop actions at the same level are marked mutex to make the optimality holds in terms of number of actions. However, in *AltAlt-p* we are interested also in encoding the information about the total number of steps, where each step can have multiple actions. So, the computation of the heuristics on *AltAlt-p* is based on a parallel planning graph rather than on a serial one. This procedure give us more effectiveness not only in the total number of parallel steps but also in the final number of actions in the plan. In Section 5 we provide empirical results of our intuition.

## 3 Representation of Parallel Plans in *AltAlt-p*

One of the main motivations in representing parallel plans is that most real world applications require to find plans given a cost measure [14]. One of these cost measures is execution time, where a plan is represented using time steps instead of sequential actions. Each time step could itself contain a number of independent actions that can be executed together, without constraining to a specific order between them. To represent parallel plans in *AltAlt-p,* we need to add incrementally to a partial plan a set of independent actions:

**Definition 1 (Independent Actions)** *Two actions $a$ and $a_1$ are considered independent in our context if and only if the resulting state $S'$ after applying both actions simultaneously is the same state obtained by applying $a$ and $a_1$ sequentially with any of their possible linearizations.*

The last definition implies that $a$ and $a_1$ can be done in parallel if their execution does not overlap at all [14]. In other words, if the preconditions and effects of each of the actions do not interact.

**Definition 2 (Interaction)** *Two actions $a$ and $a1$ do not interact with each other if* $(((prec(a) + effects(a)) \cap (prec(a_1) + effects(a_1))) = 0$

In other words, neither of the action deletes atoms from the precondition and effect lists of the other [5]. So, we consider a parallel step as one containing a set of actions which are pairwise independent. *AltAlt-p* changes the branching strategy of *AltAlt*, and it has two main alternate steps, the selection of the most promising parallel node and the rearranging of the plan. We will discuss our overall procedure in the next subsection 3.1.

### 3.1 Introducing Parallel Nodes into the Search Space

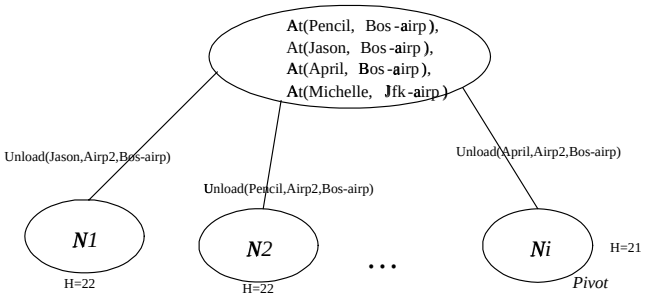
Recall from section 2 that *AltAlt-p* searches the state space of the problem in a backward direction, guided by powerful heuristics extracted from the planning graph data structure. In serial domains the execution cost of a plan is based on the occurrences of actions rather than time steps. In other words, each action represents a time step. In our parallel space, the cost is in terms of time steps that may contain a parallel set of independent actions. *AltAlt-p* will try to limit the branching factor of the problem by considering only individual actions given a state at each time step, and by creating parallel nodes, which contain the *most promising set of independent actions determined by the heuristic at each time step.* Initially, only one parallel node is created at each level of the search. The main motivation for this technique is that we do not want to explore the exponential subsets of applicable actions, and at the same time want a good approximation of the best parallel nodes during the search. Our approach picks the single best action, as recommended by the heuristic, and adds pairwise independent actions to that branch. The obvious way to parallelize the chosen branch is to add the maximum possible number of pairwise independent actions to that branch. Applying the maximum parallel set does not work well in practice, this technique does not make use of the heuristic information at all, increasing the number of actions in the plan, and consequently increasing the number of parallel steps. Just because, a set of actions could be applied in parallel does not imply that they should had been applied at that time. The main drawback is that some actions in the maximum set could take us away from the initial state, necessitating the introduction of more actions.

We specifically identify the best individual choice at each time step, we call this action the $pivot$. Using this $pivot$, we will set up a threshold for the rest of the actions. Actions that produce nodes with less heuristic cost than this threshold will be considered to create the parallel node. We do not consider independent actions whose introduction worsen the heuristic cost of the $pivot$. Basically, we identify the possible set of independent actions within the threshold and with the current $pivot$, and create the parallel node. This parallel node becomes an additional branch in the current state being evaluated, and its cost will be the same as the cost of the $pivot$ node.

It is easy to see that we may lose some relevant actions in the computation of the parallel nodes. These are the actions that could have been independent and part of the parallel node, but were discarded by the $pivot$ threshold. In order to solve this problem, and get a better approximation to the optimal parallel plan, the algorithm will identify these actions if they are selected later in the search tree for expansion. Once they are identified, our algorithm tries to rearrange the partial plan. Specifically, before expanding such actions, the algorithm will go through the current partial plan checking if there exists a node which contains only pairwise independent actions with respect to the current selection. If such a node exists, the current action is added to that node and the changes are propagated down through the partial plan to avoid losing the progress of the search. This computation is not expensive

```
GREEDYSEARCH
        pivot = Goal-State
        WHILE pivot <> NULL
          children = EXPAND( pivot )
          IF solution is in children THEN RETURN it
          OTHERWISE
                pivot = bestNodeIn ( children )
                IF pivot = NULL THEN
                   pivot = bestOpenList ();
                IF pivot <> NULL
                   pivot = pushUP ();
        CONTINUE
END
```

(a) Greedy Search Algorithm




(b) Greedy Search Tree

Figure 2: Search Algorithm and Search Tree

because it is just linear on the number of ancestors of the current node. We call this procedure $pushUp$, and it is discussed with an example in the next section.

# 4 A detailed example

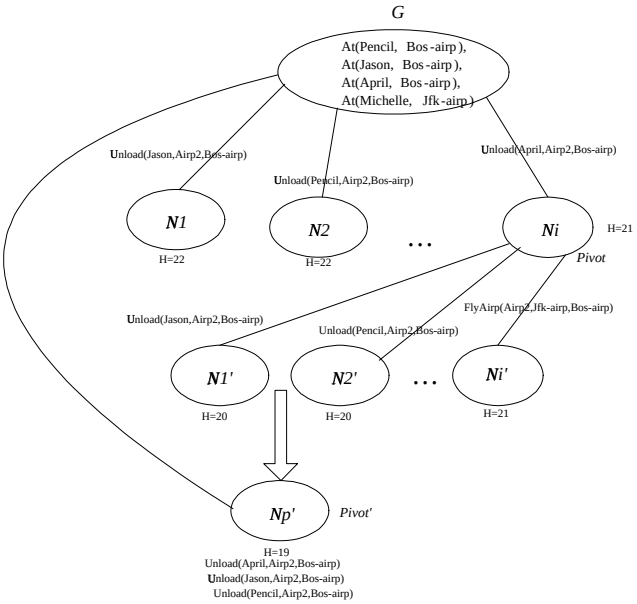
We will illustrate the details of our algorithm with an example. We can observe both the selection mechanisms in Figure 2(a), and an example of the search tree of a problem from the Logistics domain in Figure 2(b). Our goal in Figure 2(b) is to have $Jason$, $April$, and $Pencil$ at $Bos - airp$, while $Michelle$ needs to be at $Jfk - airp$. There are two planes $airp1$ and $airp2$ to carry out the plan. The picture shows the first level of the search after $G$ has been expanded. It also shows the $pivot$ node being chosen by our algorithm. At this stage of the search the call to the $pushUP$ procedure does not affect the partial plan, since the $pivot$ action "$Unload(April, airp2, Bos - airp)$" can not be pushed up to higher levels of the tree. Basically, the two core components of the search procedure as introduced in Section 3.1, are the expansion of the nodes at each time step as well as the maximization of the parallelism by $pushingUp$ actions on the partial plan.

During the expansion phase (see Figure 3(a)) a set $A$ of applicable actions will be considered for expansion. Next, given the heuristic value of the current node being expanded ($pivot$), a new set $B$ of actions is derived from $A$. In Figure 3(b), the set $A$ led to the nodes $N_1..N_{i'}$ on the tree. $B$ contains only actions from nodes $N_{1'}$ and $N_{2'}$, because those are the ones that satisfy the heuristic threshold of the curren $pivot$ $N_i$. Using $B$, we find the set of pairwise independent actions $P$, which will be used to create the parallel node. In our example $P$ and $B$ are the same. $P$ contains two new actions that are pairwise parallel among themselves and with $N_i$, and a new parallel node $N_{p'}$ is created with this subset. The procedure $Validate - Costs - Links$ from Figure 3(a) will update the necessary heuristic and cost information of the new parallel node $N_{p'}$. In our example $N_{p'}$ has been linked to the pivot's parents and its cost has been set to the pivot's cost. So, the father of $N_{p'}$ in the partial plan will be the root $G$, and its cost will be one. Finally, by setting the $pivot$ pointer to $N_{p'}$ we are indicating that only one time step has occurred in our partial plan, and this time step has three parallel actions.

We see the pseudocode procedure of $pushUP$, and an example on its operation on Figure 4(a) and

```
EXPAND pivot
        A = get_applicable_actions ( pivot );
        For a IN A
            createChildNode (a)
        B = get_best_actions ( A, pivot )
        P = get_parallel_actions ( B, pivot )
        createParallelNode ( P )
        Validate-Costs-Links ( )
    RETURN
END
```

(a) Node Expansion Algorithm




(b) Generation of Parallel Nodes

Figure 3: Node Expansion and Illustrative Example

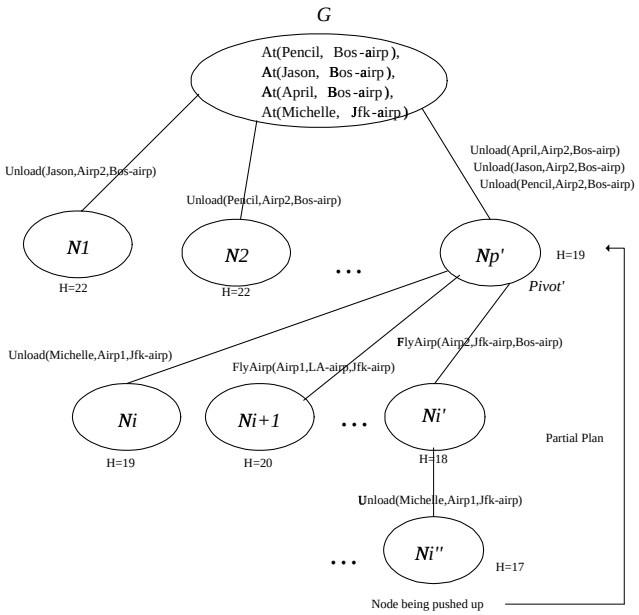
Figure 4(b) respectively. Given the current node $N_{i''}$ for expansion and the actions that originated it, we will try to see if the node's actions may have been part of earlier nodes, but were ignored because of a heuristic threshold. Following our example on Figure 4(b), node $N_{i''}$ could be pushed up because it has been left out by the initial threshold on the computation of the parallel node $N_{p'}$. We check the ancestors of $N_{i''}$ until an interacting node is found. The current action can not be pushed up further on the tree without affecting the applicability of the actions above the interacting node. In our example the root is that node because we can not regress anymore, but it could have been any node containing an action that interacted with node $N_{i''}$. Finally, the direct child of $G$ is updated by appending the action $Unload(Michelle, AIp1, Jfk - airp)$ to $N_{p'}$. This idea finds better approximations to optimal parallel plans without much additional overhead.

# 5 Evaluating the Performance of *AltAlt-p*

We implemented *AltAlt-p* on top of *AltAlt.* We have tested our implementation on a suite of parallel problems that were used in the 2000 AIPS competition [1], as well as other benchmark problems. We have compared the performance of our planner with the results obtained by running the latest version of blackbox [7] charged with a new powerful SAT solver, and STAN [10] , which is an improved version of the graphplan algorithm that reasons with invariants and symmetries to reduce the size of the problem. Both of these planners are graphplan based and are designed to produce optimal parallel plans. Our experiments were all done on a Linux system running on a 500 MHZ pentium III CPU with 256 MB of RAM. Unless noted otherwise, *AltAlt-p* was run with the $h_{AdjSum2M}$ family of heuristics

```
PUSHUP  startNode
        currentNode  = getAncestor  ( startNode  );
        For  a IN startNode
                WHILE  currentNode  < > NULL
                        IF parallel  (a,currentNode  ) AND applicable  (a,currentNode  )
                                currentNode  = getAncestor  ( currentNode  );
                        ELSE
                                newNode  = rearrangePlan  (a,currentNode  +1);
                                Validate-Costs-Links  (newNode  )
        RETURN  newNode
END
```

(a) PushUP Algorithm



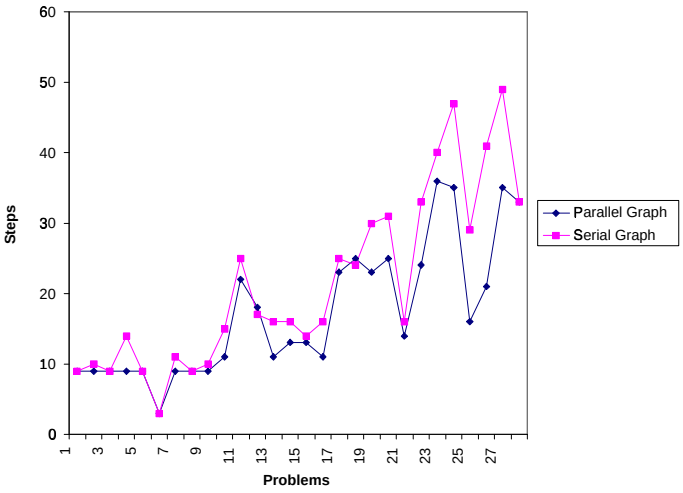

(b) Pushing an action to higher levels of the tree

Figure 4: PushUP Algorithm and Example

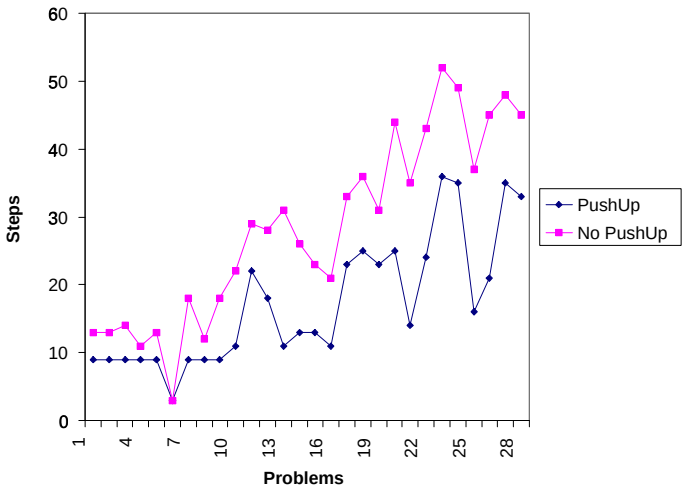
described in section 2.1 of this paper, and with a planning graph grown until the first level where the top goals are present without being mutex.

On Figure 5(a) we observe the behavior of *AltAlt-p* using a parallel planning graph and a serial one. We can see that our intuition from Section 2.1 is true, a serial planning graph overestimates the heuristic values in terms of steps, producing longer parallel solutions. We show also on Figure 5(b) another important characteristic of *AltAlt-p*, the effect of rearranging the partial plan. The $pushUP$ procedure provides higher quality in terms of the number of parallel steps. It is interesting to compare this approach with Graphplan, which tries to maximize parallelism by satisfying the most of actions at each time step, if the search fails then it backtracks and reduces the set of parallel actions being considered one level before. *AltAlt-p* does the opposite, trying to guess an initial parallel node given the heuristics, and iteratively adding more actions to this node as possible.

Recall from Section 1 that Post-processing is one of the three feasible approaches used to minimize the parallel execution of a plan. This approach is extensively discussed in [14], in which reorderings and deorderings of sequential plans are done offline to minimize the number of parallel steps. In deordering algorithms, ordering relations can only be removed, while in reordering techniques arbitrary modifications to the ordering are allowed. The main disadvantages of this approach is that the plan needs to be given, and there are not good heuristics to determine the best linearization. Since the plan is fixed, this approach is just concerned with modifying the order between the actions in the plan, but it does not consider to modify the current set of actions. In other words, this approach will try to find a minimal parallel plan with respect to the current plan, but that does not imply that it will be minimal also with respect to the initial problem. Moreover, if the plan given does not show any optimality even




(a) Parallel Planning Graph vs Serial Planning Graph




(b) Effects of rearranging the partial plan

Figure 5: Comparison of Planning Graphs and Effects of pushUp procedure in *AltAlt-p*

in the number of actions in the plan, then the final parallel plan could be very unoptimal. To prove this intuition, we have encoded the Minimal Deordering Algorithm from [14], and we have run it using the sequential plans produced by *AltAlt* using the default heuristic $h_{AdjSum2M}$. Figure 6 shows that *AltAlt-p* is superior to this approach.

Figure 7 compares the performance of *AltAlt-p* in the Logistics domain from *AIPS-00* [1] with respect to STAN and BlackBox in terms of the number of parallel steps plot 7(a) and total time plot 7(b) for each of the plans. STAN solves only half of the problems for this particular domain, and *AltAlt-p* produces optimal parallel plans for this subset. However, *AltAlt-p* tries to approximate the optimal parallel plans produced by Blackbox in the rest of the problems, but its approximation in this particular domain is not good, although in terms of efficiency outperforms clearly the other two competing approaches..

Results from the Scheduling domain in Figure 8 present a similar behavior. *AltAlt-p* can approximate optimal parallel plans for a large set of the problems, having a better performance than in the Logistics domain. On Figure 8(b) we observe another important criteria for evaluating plans: total number of actions. For this case, *AltAlt-p* is better than both of the other planners. Recall from Section 2.1 that our heuristic allow a good trade off between parallelism and length of the plan. While Blackbox and STAN are completely pushing for the best parallelism of the plan, adding the most of actions at each time step, even if they are not necessary, *AltAlt-p* heuristic and algorithm are trying to guess an approximation to the optimal parallel plan without fully committing to insert all independent new actions into the plan as Blackbox and STAN do. In other words, Blackbox and STAN will insert more actions at each time step to get better parallelism but increasing the plan length, and *AltAlt-p* will only insert the most promising actions at each time step reducing the length of the plan, but also increasing the number of parallel steps. We can observe that this intuition is true on Figure 8(b). In

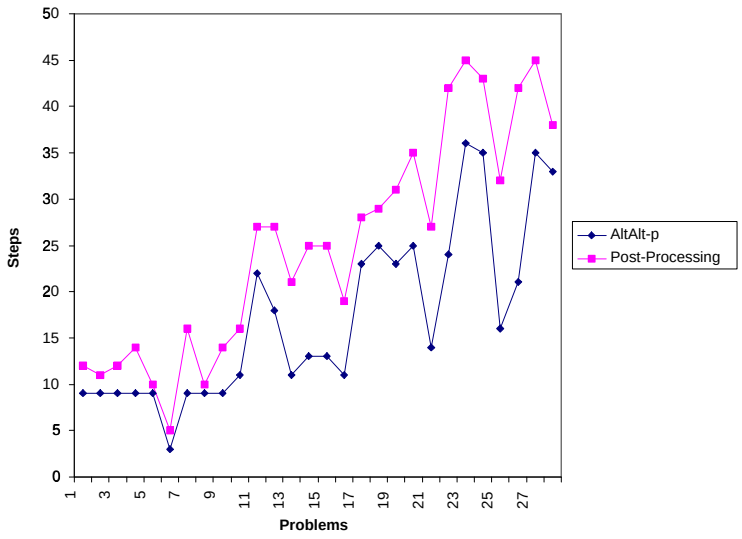



Figure 6: *AltAlt-p* vs Post-Processing algorithm

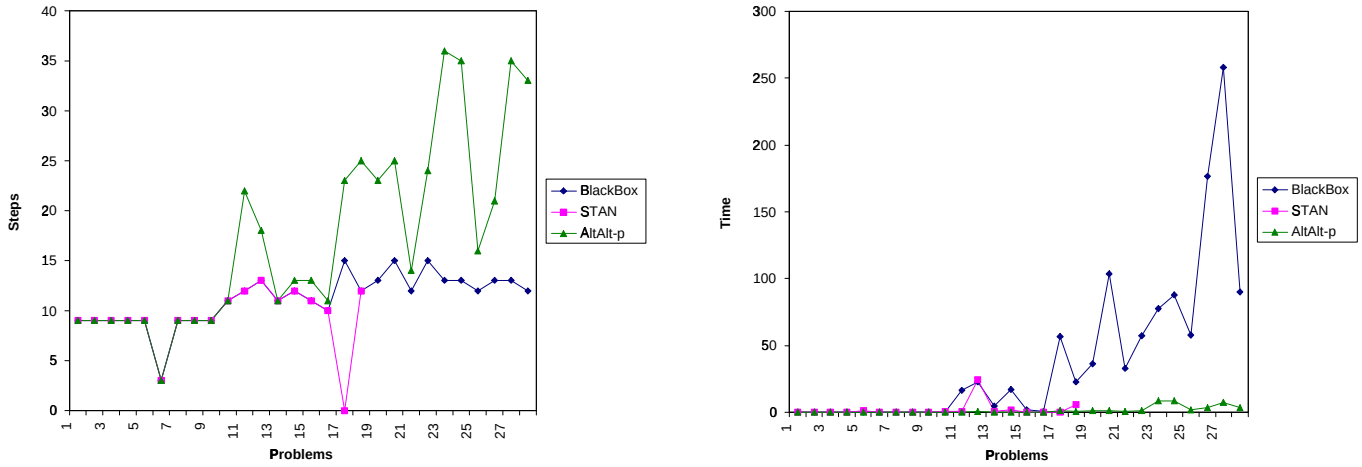
the first 50 problems tested *AltAlt-p* returns optimal parallel plans, and there is not a major difference on the length of these plans and those produced by the other two systems. In the rest of the problems *AltAlt-p* approximates the parallelism of STAN and Blackbox, getting longer parallel plans but shorter sequences.

Figure 9 compares the number of steps and time performance of *AltAlt-p*, STAN and Blackbox on the Gripper domain. We see that for many of these problems *AltAlt-p* is the only possible solution to generate parallel plans. Specifically, neither of the other two approaches is able to solve more than four problems. Furthermore, *AltAlt-p* is able to solve all and get optimal plans. In Figure 9(b) we have included the serial version of *AltAlt* and we can observe that *AltAlt-p* actually performs slightly better than the normal version. Our intuition here is that we are trying to push the parallelism of a problem in a limited way, then the difference among the two versions of *AltAlt* will be small but significant for the parallel version on parallel problems.

One concern would be how much of extra computation the parallel algorithm takes in serial domains. We expect it to be negligible because both the parallelization and $pushUP$ steps can fail very quickly in serial domains. To validate our expectation we ran *AltAlt* and *AltAlt-p* on a set of serial problems of the Blocks-world domain from *AIPS-00* [1]. We see from the plots in 10 that the length of the two solutions and time performances are basically equal supporting our intuition.

# 6 Related work

The idea of partial exploration of parallelizable sets of actions is not new [13, 12]. It has been studied in the area of concurrent and reactive planning, where one of the main goals is to approximate optimal parallelism. However, most of the research there has been focused on Forward-Chaining Planners [13], where the state of the world is completely known. It has been implied that backward-search methods are not suitable for this kind of analysis [12] because the representation of the states are partial. We




(a) Number of Steps

(b) Total Time

Figure 7: Logistics Domain (AIPS-00)

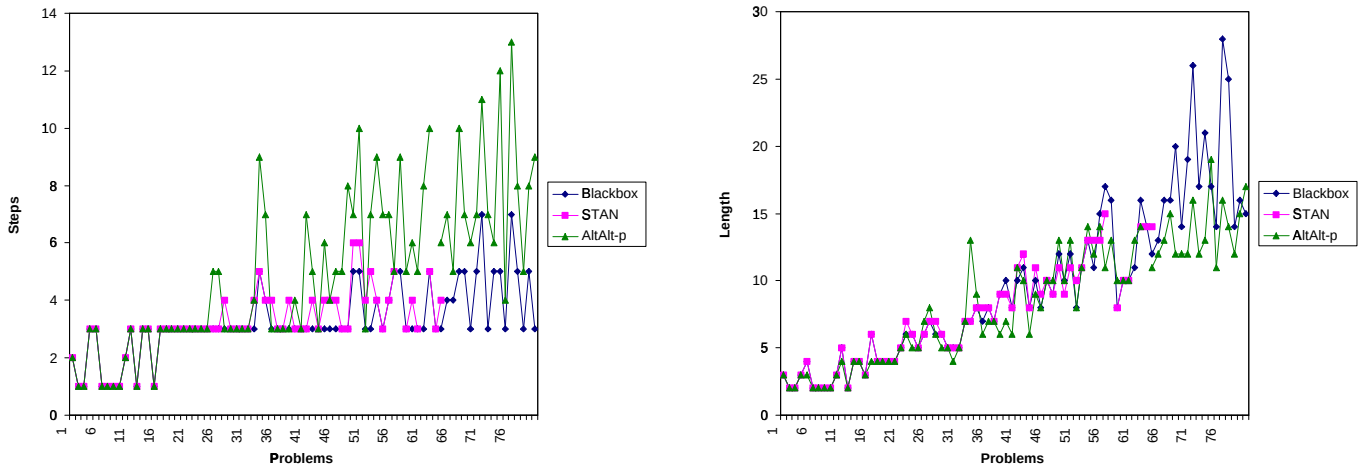
have shown that backward-search methods can be also used to approximate parallel plans in the context of classical planning.

Optimization of plans according to different criteria (e.g. execution time, quality, etc) has also been done offline. The post-processing computation of a given plan to minimize its parallelism has been discussed in [14]. Reordering and deordering techniques are used to minimize the parallelism of the plan. However, as discussed in Section 1 and 5, they are just concerned with modifying the order between the existing actions of a given plan. Our approach not only considers modifying such orderings but also inserting new actions which can minimize the possible number of parallel steps of the overall problem.

We have already discussed Graphplan based planners [7, 10], which return optimal plans based on the number of time steps. Graphplan uses IDA* to include the most possible number of parallel actions at each time step of the search. However, the iterative procedure is both memory intensive and reduces the efficiency of the planners. Finally, some work has been done to obtain admissible heuristics for optimal planning [5]. That work shows that optimal parallel plans for state space search are hard to achieve, and they do not approach the performance of Graphplan based planners. Our implementation, although an approximation, provides a tradeoff between quality in terms of the number of parallel steps and number of actions contained in the plan at a fraction of time of the competing approaches.

# 7 Concluding Remarks

We have presented an approach to find parallel plans in state space search. This is a challenging problem because of the exponential branching factor caused by naive methods. Our approach tries to limit the branching factor of the problem by exploring a subset of the possible parallel sets in state space search. Our implementation is based on a backward search planner, which makes use of heuristics extracted




(a) Number of Steps

(b) Total Number of Actions

Figure 8: Schedule Domain (AIPS-00)

from a planning graph data structure. The algorithm tries to minimize the number of parallel steps considering only a subset of all applicable actions at each time step, and updating dynamically the partial plan during the search if additional actions can be parallelized at higher levels of the tree.

We have demonstrated that the additional cost incurred by our algorithm in serial plans is negligible. We have compared our approach with Post-Processing techniques and Graphplan based planners, which produce optimal parallel plans. We have seen that our results are promising, producing a tradeoff between quality and efficiency.

## References

[1] F. Bacchus. Results of the AIPS 2000 Planning Competition. URL: http://www.cs.toronto.edu/aips-2000.

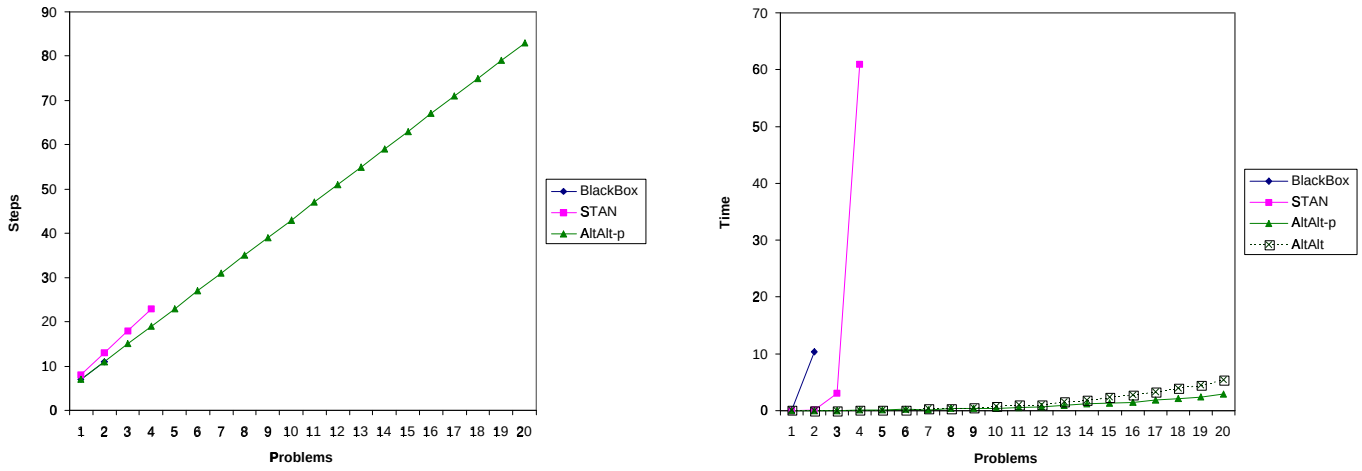
[2] A. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*. 90(1-2). 1997.

[3] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proc. ECP-99*, 1999.

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

[5] P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. In *Proc. AIPS-2000*, 2000.

[6] S. Kambhampati. EBL & DDB for Graphplan. *Proc. IJCAI-99*. 1999.




(a) Number of Steps

(b) Total Time

Figure 9: Gripper Domain (AIPS-98)


[7] H. Kautz and B. Selman. Blackbox: Unifying sat-based and graph-based planning. In *Proc. IJCAI-99*, 1999.

[8] X. Nguyen and S. Kambhampati. Extracting effective and admissible heuristics from the planning graph. In *Proc. AAAI-00*, 2000.

[9] X. Nguyen, S. Kambhampati, and R. Sanchez Nigenda. Planning Graph as the Basis for deriving Heuristics for Plan Synthesis by State Space and CSP Search. ASU Technical Report. *To appear in Artificial Intelligence.*

[10] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *JAIR*, 10(1-2) 1999.

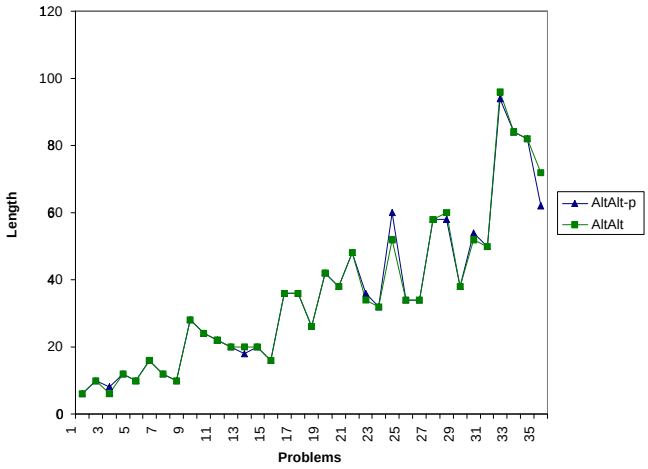
[11] D. McDermott. Using regression graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–160, 1999.

[12] P. Godefroid and F. Kabanza. An Efficient Reactive Planner for Synthesizing Reactive Plans. In *Proc. AAAI-91*, 1991.

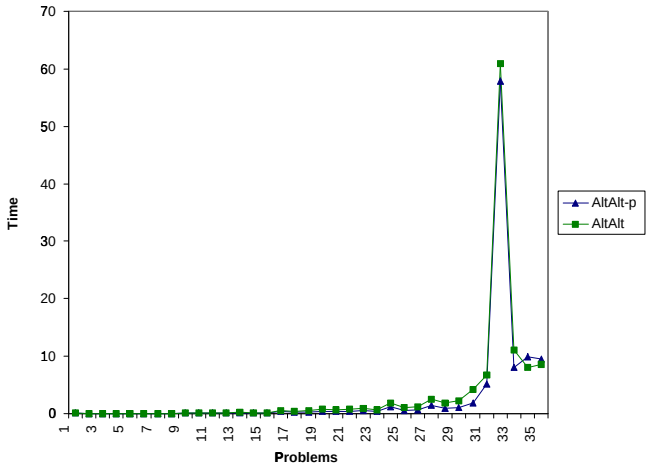
[13] F. Kabanza. Planning and Verifying Reactive Plans (Position Paper). In *Proc. AAAI-97*, 1997.

[14] C. Backstrom. Computational Aspects of Reordering Plans In *Journal of Artificial Intelligence Research*, 1998.

[15] D. Weld. Recent advances in AI planning. *AI magazine*, 1999.




(a) Total Number of Actions




(b) Total Time

Figure 10: Blocks World Domain (AIPS-00)