UNIVERSITÄT BASEL

Iterative Tunneling A* in Planning

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Computer Science
Artificial Intelligence
ai.cs.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Martin Wehrle

Stefano Branco
stefano.branco@stud.unibas.ch
09-065-863

July 10th, 2013

Acknowledgments

I would like to express my thanks to Dr. Martin Wehrle for his support and supervision of my work, as well as Dr. Malte Helmert for allowing me to write this thesis in the field of artificial intelligence. I would also like to thank my friends and colleagues, and especially my family, for their support during this time.

Abstract

In planning, we address the problem of automatically finding a sequence of actions that leads from a given initial state to a state that satisfies some goal condition. In *satisficing* planning, our objective is to find plans with preferably low, but not necessarily the lowest possible costs while keeping in mind our limited resources like time or memory. A prominent approach for satisficing planning is based on heuristic search with inadmissible heuristics. However, depending on the applied heuristic, plans found with heuristic search might be of low quality, and hence, improving the quality of such plans is often desirable.
In this thesis, we adapt and apply *iterative tunneling search with A* (ITSA*)* to planning. *ITSA** is an algorithm for plan improvement which has been originally proposed by Furcy et al. for search problems. *ITSA** intends to search the local space of a given solution path in order to find "short cuts" which allow us to improve our solution. In this thesis, we provide an implementation and systematic evaluation of this algorithm on the standard IPC benchmarks. Our results show that *ITSA** also successfully works in the planning area.

Table of Contents

Acknowledgments **i**

Abstract **ii**

1 Introduction **1**

2 Background **3**
2.1 Planning 3
2.2 Heuristic Search and A* 3

3 Iterative Tunneling Search with A* **6**
3.1 Idea 6
3.2 The Algorithm 7
3.3 Implementation 9

4 Evaluation **12**
4.1 Experimental Setup 12
4.2 Results 13
4.2.1 Landmark Cut 13
4.2.2 Fast Forward 14
4.3 Discussion 15

5 Conclusion **17**

Bibliography **18**

Declaration of Authorship **19**

1 Introduction

Planning is the attempt to automatically find a sequence of actions leading from an initial state to one of many goal states. This sequence of actions is called a plan. There are two main disciplines in planning - *satisficing* and *optimal* planning. While optimal planning is only interested in finding a plan where there provably does not exist any plan with lower cost, satisficing planning also accepts suboptimal plans, and instead focuses on managing limited resources such as time or memory.

One widespread method to find plans is called heuristic search. A heuristic is a mathematical function that assigns every state a numerical value which represents an approximation of the distance of each state to a goal. A heuristic based search then uses this value to keep a sorted list of which states to expand next. It is immediately apparent that the quality of found plans is highly dependant on the choice of heuristics.

While satisficing planning does not require a plan to be optimal, we are still interested in obtaining the best possible solution within our limited resources. One possibility to achieve this goal is plan improvement. Depending on the heuristic used, plans found by common heuristic search algorithms are often far from the optimal solution, but may still give us information that we can use to craft better solutions. As the name suggests, plan improvement intends to take an existing plan, e.g. one found by a heuristic search, and improve that plan. Within the search community, Furcy et al. [1] have proposed and evaluated such an algorithm called *iterative tunneling search with A* (ITSA*)* in order to improve their search results. The basic idea behind *ITSA** is to search the area around a found plan for improvements on our original play, hoping to eventually lead us to a plan as close as possible to the optimal plan with little additional effort. The contribution of this work is to adapt the algorithm proposed by Furcy at al. to planning, implement that algorithm in the existing Fast Downward planning system [2] and evaluate that implementation based on standard IPC benchmarks.

The rest of this work is organised as follows. The second chapter will provide the theoretical background on planning and heuristics. The third chapter will focus on the motivation of *ITSA** and introduce the algorithm in it's original form as well as the adaptions we made. The fourth chapter will contain the results of our evaluation, and the last chapter will provide the conclusions of this work.

2
Background

This chapter will lay the background foundation required to understand the algorithm introduced in chapter three. It will formally define planning tasks and heuristic functions, and introduce the concepts behind heuristic search and A*.

2.1 Planning

Each planning task is given by a tuple $(\mathcal{V}, \mathcal{O}, s_0, s_*)$. $\mathcal{V}$ is a finite set of state variables. Each variable $v \in \mathcal{V}$ has it's own finite Domain D_v. A partial state over $\mathcal{V}$ is a function s over a subset $\mathcal{V}_s \subset \mathcal{V}$. $\mathcal{O}$ is a finite set of operators over $\mathcal{V}$. Each operator $\mathcal{O} = \langle pre, eff \rangle$ has a precondition pre and an effect eff. A precondition is a partial state over $\mathcal{V}$, and an operator can only be applied to a state s if it's preconditions are fulfilled, whereas the effect represents the effects caused by applying such an operator to a state s. Applying any operator additionally has a cost linked to it. s_0 is the initial state of our problem, and s_* is a partial state over $\mathcal{V}$ representing our goal.
Planning is the attempt of automatically finding a sequence of operators which, when applied to our initial state s_0 end up arriving at any goal state. *Optimal* planning is interested only in plans whose total costs are provably the lowest possible, while *statisficing* planning, the discipline we are interested in, accepts suboptimal solutions as well.

2.2 Heuristic Search and A*

A heuristic is a function $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$ which maps every state to a numeric value representing an approximation of it's distance to a goal state. Heuristic functions have different characteristics which can influence when it is sensible to use a certain heuristic and when it is not. A heuristic is called admissible if it never overestimates the cost to the goal, and consistent if for every pair of connected states s and s', $h(s) \leq c_{s \to s'} + h(s')$, where $c_{s \to s'}$ represents the cost of the operator that leads s to s'. A consistent heuristic is automatically admissible (but not vice-versa).
A heuristic based search can be used to systematically search the state space of a planning task for a plan. Since a heuristic is supposed to estimate the distance from any state to a

goal state, heuristic search prioritizes exploring states with a low heuristic value. Algorithm 1 shows a pseudo code example of how such a heuristic search might look like. In this case, function $priority(s) = h(s)$ returns the heuristic value of a state s. This is called greedy search.

There is another piece of information that we can use to improve the above algorithm.

Algorithm 1: Heuristic Search Algorithm

```
1  open := newpriorityqueue
2  open.insert(s_0, priority(s_0))
3  closed := ∅
4  while not open.empty() do
5      s = open.pop_min()
6      if s ∉ closed then
7          if is_goal(s.) then
8              return extract_solution(s)
9          end
10         closed.push_back(s.state)
11         foreach successor s' of s do
12             if priority(s') < ∞ then
13                 open.insert(s', priority(s'))
14             end
15         end
16     end
17 end
18 return nosolution
```

A* uses both the heuristic value of a state as well as the path cost from the starting state to a state to evaluate which way to go next. The idea behind this is while we want to eventually reach a goal, we also want to make sure that the path we have taken to that goal is as short as possible. This way, provided we have a consistent heuristic, we can guarantee not only to find a plan for a given planning task, but find an optimal plan without having to revisit any states. The above code can be used for A* as well, with the slight modification that in this case, $priority(s) = g(s) + h(s)$, where $g(s)$ is the path cost from the initial state up to the state s, and $h(s)$ is once again the heuristic value of s.

It becomes immediately apparent that both A* and a basic heuristic search are highly dependant on the quality of the used heuristic. Assigning every state a random number, while being a valid heuristic, will not be of much help. On the other hand, a heuristic that could calculate the exact distance of every state to the goal would make the search process trivial.

However, calculating the heuristic values of a state is not always a simple task. Complex heuristics, such as Landmark Cut [3], one of the heuristics we are using to evaluate *ITSA** for improving the quality of a suboptimal plan, rely in computationally intensive operations. This means that by using less complex heuristics, we can save time while at the same time accept that our solution might not be as close to optimal compared to when using a more advanced heuristic. This is significant for us in two ways. First, it means that in our satisficing domain we often deal with suboptimal solution plans, plans that we intend to improve using *ITSA**, which we will introduce in chapter three. And second, since *ITSA**

is based on A* itself, we can alter the performance of our algorithm by using different heuristics, which will become important when benchmarking our implementation.

3

Iterative Tunneling Search with A*

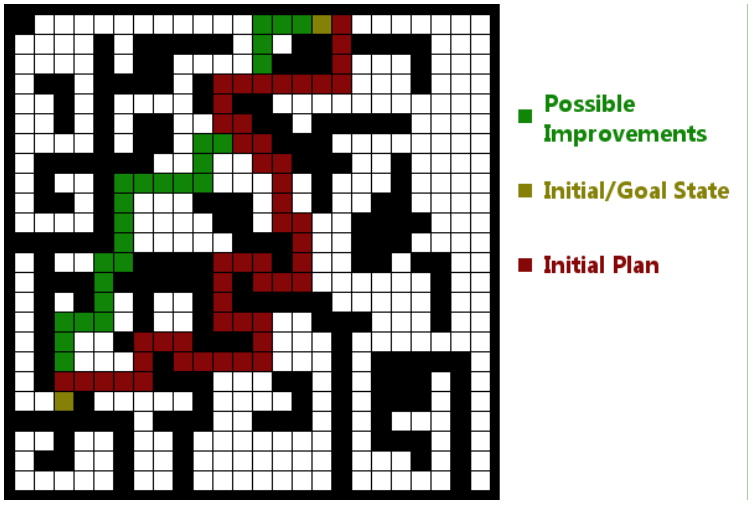
In this chapter we will introduce the *ITSA**[1] algorithm. We are working under the assumption that we already found a solution plan P, e.g. with a heuristic search algorithm as proposed in chapter two. *ITSA** systematically tries to improve this plan by searching the area around the plan for short cuts. In chapter 3.1 we will provide the general concept behind *ITSA**, and chapter 3.2 will introduce the actual algorithm.

3.1 Idea

As proposed by Furcy at al., the idea behind *iterative tunneling search with A** is to search the neighbourhood of a solution plan P for plans P' with lower total cost than P. In more figurative way, we are looking for shortcuts (tunnels) between the initial state s_0 and our goal state s_* which decrease the total cost of our original plan P.

The image 3.1 shows a scenario where *ITSA** is applicable and would result in a decrease in

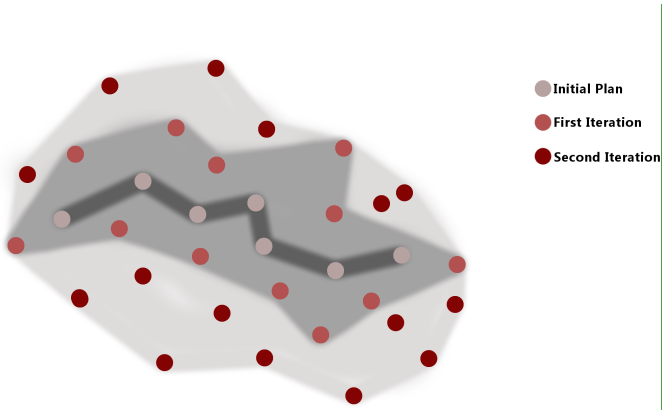
Figure 3.1: Visualisation of a possible scenario where *ITSA** is applicable



overall plan cost. The yellow fields mark our initial and goal states. The cost of transitioning

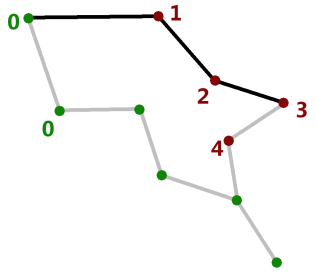
from any square to another is uniform, meaning it is the same for all squares. By applying any kind of algorithm, we have found our original plan P, marked in red. Green marks two possible improvements to our initial plan P which would decrease our total cost. The intention of *ITSA** is to find these short cuts. To do this, Furcy et al. propose to search a larger and larger area around the original plan P with A* for shorter solution plans until we run out of memory. Intuitively, one might define the neighbourhoods of our plan as shown in figure 3.2. Bright Circles in the dark grey area correspond to points on our initial Plan

Figure 3.2: Solution plan and it's neighbourhood



P. Each new state accessible via application of one operator on at least one of the states part of P is part of the immediate neighbourhood, which is represented by the brighter area. New states accessible via states of iteration level one are assigned iteration level two (brightest area), and so forth. However, to assign each state it's correct distance from the initial plan, we would have to pre-process our search by expanding the state space around our plan P using a breadth-first search. This is highly inefficient, since it basically requires us to perform two searches. To prevent this, Furcy et al. suggest an alternative method of assigning the distance values that can be performed during the actual search. Whenever a state is newly generated by A*, it is assigned the iteration value of it's parent plus one, or zero if it's part of the original plan. Only states with iteration levels smaller than the current iteration limit get inserted into the open list. While this might initially seem like the same thing, it is actually a slightly more strict requirement. This is because once a state is assigned a distance, it is never changed, however it might very well be closer (in terms of states) to another part of the original plan, as opposed to the one the algorithm arrived from. The search depth of a state therefore depends on which way the algorithm took to expand that state, which means a state may have different search depth values depending on the heuristic. Figure 3.3 illustrates that issue. Every circle represents a state. Green states are our initial plan, while red ones are it's neighbourhood. Our algorithm travels along the black path. The numbers represent the distance from the initial plan as calculated with this method. When the successors of the state with distance *2* and *3* are created, they are assigned the value of their parents increased by one, which is *3* and *4* respectively, even though their actual distance from the solution plan is lower (*1* and *2*. In every other way, the algorithm performs exactly like a basic A* algorithm.

Figure 3.3: Illustration of Distance Assignement



3.2 The Algorithm

Algorithm 2 shows pseudo code of the algorithm as it was proposed by Furcy et al. Since this is basically a slightly modified A*, *open* is a min-heap ordered by $f(n) = g(n) + h(n)$. We start with our initial state in the open list and an empty closed list, as well as a iteration level of one (1-4). *get_Plan*() returns our original plan we found with whatever algorithm we used beforehand (5). While there are elements in our open list, we remove our minimum (6). We ignore it if it is already in our closed list(8). If it's a goal state, we extract our solution, save it and continue with our next iteration (9-12), otherwise create our successor states for each successor who's heuristic value is smaller than infinity, and assign them their distance values (14-21). We insert those with distances smaller than our current iteration limit (22-24), and repeat this until we found a solution. Once we found one, we increase our iteration level (29) and repeat the process. Eventually we will run out of memory, at which point our last solution saved will be our resulting plan.

Algorithm 2: Iterative Tunneling Search with A*

```
1  iteration = 1;
2  while true do
3      open := new priority queue;
4      open.insert(s_0, g(s_0) + h(s_0));
5      closed := ∅;
6      P := get_plan();
7      while not open.empty() do
8          s = open.pop_min();
9          if s ∉ closed then
10             if is_goal(s) then
11                 solution = extract_solution(s);
12                 break;
13             end
14             closed.push_back(s);
15             foreach successor s' of s do
16                 if h(s') < ∞ then
17                     if s' ∈ P then
18                         s'.distance = 0;
19                     end
20                     else
21                         s'.distance = s.distance + 1;
22                     end
23                     if s'.distance < iteration then
24                         open.insert(s', g(s') + h(s'));
25                     end
26                 end
27             end
28         end
29     end
30     iteration = iteration + 1;
31 end
```

3.3 Implementation

*ITSA** has been implemented for the Fast Downward[2] planner, or to be more exact, as an improvement for the eager-greedy search within the planner. The original idea of *ITSA** is to iteratively search further and further away from the initial plan, until one eventually runs out of memory. As we are interested in getting the best possible result while keeping in mind our limited resources, we have slightly modified the original *ITSA**. We have removed the iterative nature of *ITSA**, and instead made the maximum search depth a parameter of the actual search. That way, we only need to perform one search up to the maximal depth, rather than performing multiple searches with increasing iterations. This saves us a significant amount of time, and also allows us to observe the difference in performance of our implementation with different maximum search depths. In pseudo code, this changes the original version to something like Algorithm 3, where $depth_limi$ is the parameter that marks our maximum search depth.

Algorithm 3: Modified Tunneling Search with A*

```
1  open := new priority queue;
2  open.insert(s_0, g(s_0) + h(s_0));
3  closed := ∅;
4  P := get_plan();
5  while not open.empty() do
6      s = open.pop_min();
7      if s ∉ closed then
8          if is_goal(s) then
9              return extract_solution(s);
10         end
11         closed.push_back(s);
12         foreach successor s' of s do
13             if h(s') < ∞ then
14                 if s' ∈ P then
15                     s'.distance = 0;
16                 end
17                 else
18                     s'.distance = s.distance + 1;
19                 end
20                 if s'.distance < depth_limit then
21                     open.insert(s', g(s') + h(s'));
22                 end
23             end
24         end
25     end
26 end
```

4 Evaluation

This chapter contains information about the experimental setup, the results of our evaluation as well as a discussion about these results.

4.1 Experimental Setup

All experiments were performed on a Intel Xeon E5-2660 (2.2GHz) with a time limit of 30 minutes and a memory limit of 2GB respectively. As discussed in the previous chapter, we have chosen to not iteratively search deeper and deeper, but instead directly search at our iteration limit. This was a bit optimistic, since our algorithm runs out of memory more often than expected. In these cases, our resulting plan is simply the initial plan, rather than the improved one of smaller depth limits.
To find our initial plans, we have used two different heuristics with a greedy search - Landmark Cut [3], which is consistent, admissible and comparatively precise, but also rather costly in terms of computation time, as well as Fast Forward (ff) [4], the less precise but computationally more efficient heuristic of the two.
Our tunneling algorithm itself was run with three different heuristics. Blind search is very fast, but, as the name suggest, uninformed and therefore rather inaccurate. Landmark Cut once again marks the top end of our benchmark spectrum, with it's high accuracy and low computation speed. The last of the three is H Max [5], which places itself somewhere in between those two. Not all domains are compatible with all heuristics - some domains have no solutions, even no initial solutions. These are obviously not interesting in the evaluation of our algorithm.

4.2 Results

This chapter will show the results of our evaluation and specifically will show how our implementation of *ITSA** performed with our different setups.

4.2.1 Landmark Cut

All three heuristics perform very similar in terms of plan cost when using Landmark Cut to find our initial plan. That is, they improve the result, but only slightly. The difference between the different heuristics is negligible at best in terms of resulting plan cost for most domains. Figure 4.1 show a selection of domains and problems and their average cost for Landmark Cut, while table 4.2 shows the average search time required for those results. It is noteworthy that in table 4.3 and 4.4, which show the same total averages as table 4.1 but for all setup combinations, Landmark Cut actually performs worse than blind. The reason for that is most likely the difference in search depth values assigned to some states, as mentioned in chapter three. The differences in search time required however vary greatly more between heuristics. It seems apparent that when starting with a good initial plan, finding short cuts does not require a sophisticated heuristic. While H Max and blind perform better in terms of search time required, it is still questionable whether the small decreases in plan cost is worth the time.

Table 4.1: Average Cost of Landmark Cut over Selected Problems and Domains

lmcut - lmcut	**depth 1**	**depth 2**	**depth 3**	**depth 5**	**depth 7**	**depth 10**
airport	79	79	79	79	79	78.9
blocks	56.4	55.2	55.2	51.1	49.4	44.3
driverlog	18	18	18	17.7	17.5	16.6
freecell	19.11	18.8	18.8	18.7	18.6	18
gripper	53	53	53	53	52.3	51.9
logistics00	27.1	27.1	27.1	27	27	26.5
miconic	31.3	31.3	31.1	31.1	30.9	30.7
movie	7	7	7	7	7	7
mprime	5.4	5.4	5.4	5.4	5.4	5.4
mystery	6.9	6.9	6.9	6.5	6.2	5.9
pegsol-08-strips	13.3	13.3	13.3	13.3	13.3	13.3
pegsol-sat11-strips	8.9	8.9	8.9	9	9	8.9
pipesworld-notankage	11.3	11.3	11.3	11.4	11.4	11.2
pipesworld-tankage	23.8	23.2	23.1	22.1	21.4	20.5
psr-small	25	24.8	24.8	24.4	23.7	21.8
schedule	24.6	24.6	24.1	22.6	21.4	21.4
sokoban-sat08-strips	3.8	3.8	3.8	3.8	3.8	3.8
sokoban-sat11-strips	49.3	493	49.3	49.3	49.1	49
storage	58.3	58.3	58.3	58.3	58.1	58
trucks-strips	652.7	620.7	610.2	581.8	540	541.7
zenotravel	55	55	50	50	50	50
average	**26.9**	**26.8**	**26.7**	**26.3**	**26**	**25.5**

Table 4.2: Average Total Time of Landmark Cut over Selected Problems and Domains

lmcut - lmcut	**depth 1**	**depth 2**	**depth 3**	**depth 5**	**depth 7**	**depth 10**
airport	33	36.5	38.7	46.2	53.8	71.9
blocks	0.7	0.7	0.9	2.3	9.8	121.6
driverlog	0.1	0.2	0.4	1.6	4	7.4
freecell	3.7	5.5	10.4	45.1	112.9	176.8
gripper	0.2	0.5	1.1	5.7	24.3	150
logistics00	0.1	0.2	0.4	1	2.3	2.9
miconic	0.2	0.8	1.2	5.9	22.7	114.4
movie	0.1	0.1	0.1	0.1	0.1	0.1
mprime	24.4	27.9	44.2	43.6	45.5	45.9
mystery	51.6	55.1	61.9	77.3	69.2	55.9
pegsol-08-strips	1.4	1.4	1.4	1.6	2.3	7.6
pegsol-sat11-strips	2.1	2.1	2.1	2.4	3.6	11.6
pipesworld-notankage	27.3	27.5	28.4	33.8	56.4	146.5
pipesworld-tankage	35.8	37.7	40	52.3	67.1	121.2
psr-small	0.4	0.5	0.6	2	5.4	17.6
schedule	0.2	0.7	2.2	11.3	39.9	162.8
sokoban-sat08-strips	0.1	0.1	0.1	0.1	0.1	0.1
sokoban-sat11-strips	85.1	85.7	85	85.1	85.8	86.6
storage	130.9	131.7	131.7	132.4	132.6	133.8
trucks-strips	0.3	0.4	0.6	2	7.7	38.8
zenotravel	0.2	0.4	1.1	5.3	20.1	85.7
average	**19**	**19.8**	**21.5**	**26.5**	**36.5**	**74.2**

4.2.2 Fast Forward

Starting with a ff provides a higher initial starting cost, though not as much as initially expected. This setup is where our tunneling starts to show it's strength. The average costs as shown in table 4.2 are about the same as with the better initial plan, while the total time required is significantly lower. Blind and H Max achieve comparable results while needing not nearly as much time as Landmark Cut. A direct comparison between blind and H Max is shown in figure 4.1 below, which shows that blind is clearly the best choice for performing tunneling, both in terms of plan quality as well as time cost.

Figure 4.1: Comparison Between Blind and H Max

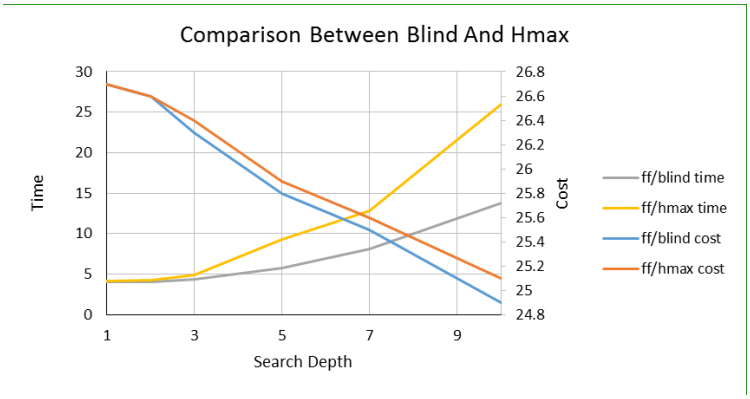


Table 4.3: Comparison of Average Cost For All Setups

lmcut - lmcut	26.9	26.8	26.7	26.3	26	25.5
lmcut - hmax	26.9	26.7	26.7	26.2	26	25.5
lmcut - blind	26.9	26.7	26.7	26.1	25.8	25.3
ff - lmcut	26.7	26.6	26.6	25.9	25.7	25.1
ff - hmax	26.7	26.6	26.4	25.9	25.6	25.1
ff - blind	26.7	26.6	26.3	25.8	25.5	**24.9**

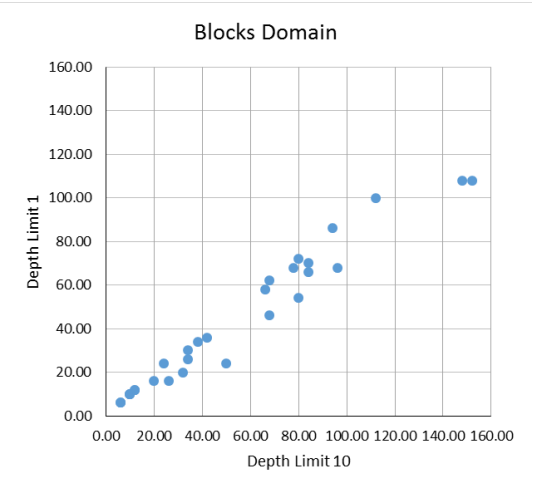
Table 4.4: Comparison of Average Time For All Setups

lmcut - lmcut	19	19.8	21.5	26.5	36.5	74.2
lmcut - hmax	18.8	19	20	24.9	28.9	40.8
lmcut - blind	18.8	18.9	19.2	20.5	23	29.1
ff - lmcut	4.3	4.9	6.4	10.3	20.2	57.4
ff - hmax	4.1	4.3	4.9	9.3	12.8	26
ff - blind	4	4	4.4	5.7	8.1	**13.8**

4.3 Discussion

There is a problem that plagues both benchmarks - memory shortage. For quite a few domains that were not represented in the averages above, there was not enough memory to calculate up to a depth where significant improvements, or even any improvements, can be achieved. The time costs are also a fact to consider. It is therefore questionable whether tunneling can be used as a general method to improve suboptimal plans. Regardless of that, as can be seen in our averages of domains not plagued by that issue, Fast Forward in combination with a Blind heuristic can provide results comparable with the sophisticated Landmark Cut heuristic in lower time. There are even domains where tunneling can be applied to greatly improve our results with very little effort, such as the blocks domain. The results of performing tunneling with a depth limit of one and ten respectively can be seen in figure 4.2 below. All in all, the results did not meet our expectations. In some circumstances,

Figure 4.2: Blocks Domain Costs With Search Depth 1 and 10



especially with high initial plan costs, tunneling is certainly worth the additional time it needs. More often than not however, massive memory cost and little benefit compared to the time required cripple the effects of tunneling. Going back to the original iterative method might soften the burden of the memory cost, but will only decrease the ratio of information gain and computation time.

5 Conclusion

In this Thesis we have adapted *ITSA** for the Fast Downward planner and evaluated it's performance. It's iterative nature has been dropped for a more direct approach in hope of reducing search time while keeping information gain.
The planned evaluations were performed with different initial paths as well as different heuristics used to perform the actual tunneling. Our evaluation results showed that the expected improvement exists, however that on average it is rather small compared to the time it costs. This is especially true if the initial plan is already close to the optimal solution. For many domains, a search distance of around five from the initial plan is required to provide satisfying results. For many domains however, that search depth was already reaching the limit of available memory.
This results in a sobering overall result, with many domains performing worse than expected. A return to the iterative method might provide more consistent improvements, but will further reduce time effectiveness of our algorithm.
While it was expected that Landmark Cut would perform badly in terms of time required, it was a surprise that after ten iterations, for many domains the quality was not better than of our tests with Blind or H Max. Regardless, for some domains, like the blocks domain, tunneling performs exceptionally well, producing decent results in reasonable time. As a general improvement for suboptimal solution plan, it is therefore not practical. As a domain-specific improvement for plans which are expected to be rather far from the optimal, it is certainly an option. It is also interesting to note that out of our three heuristics, the Blind heuristic actually performed best.

Bibliography

[1] Furcy, D. A. ITSA*: Iterative Tunneling Search with A*. *AAAI Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, pages 21–26 (2006).

[2] Helmert, M. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246 (2006).

[3] Helmert, M. and Domshlak, C. LM-Cut: Optimal Planning with the Landmark-Cut Heuristic (2009).

[4] Hoffmann, J. FF: The Fast-Forward Planning System. *AI magazine*, 22:57–62 (2001).

[5] Bonet, B. and Geffner, H. Planning as Heuristic Search. *Artificial Intelligence*, 129:5–33 (2001).

Declaration of Authorship

I hereby declare that this thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the bibliography and specified in the text.
This thesis is not substantially the same as any that I have submitted or will be submitting for a degree or diploma or other qualification at this or any other University.

Basel, date

Author