Searching for an Alternative Plan

Ariel Felner
Dept. of Computer Science
Bar-Ilan University
Ramat-Gan, 52900, Israel
felner@cs.biu.ac.il

Alex Pomeransky
Dept. of Computer Science
Bar-Ilan University
Ramat-Gan, 52900, Israel
pomeransky@cs.biu.ac.il

Jeffrey S. Rosenschein
School of Computer Science
and Engineering
Hebrew University
Jerusalem, 91904, Israel
jeff@cs.huji.ac.il

ABSTRACT

Suppose that an intelligent agent accepts as input a complete plan, i.e., a sequence of states (or operators) that should be followed in order to achieve a goal. For some reason, the given plan cannot be followed by the agent, and thus an alternative plan needs to be found — but we would like the alternative plan to be as close as possible to the original. To achieve this, we define a number of *distance metrics* between paths or plans, and characterize these functions and their respective attributes and advantages. We then develop a general algorithm based on best-first search that helps an agent find the most suitable alternative plan efficiently, and propose a number of heuristics for the cost function of this best-first search algorithm. We then experimentally show that our algorithm is efficient in finding an alternative plan.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed AI—*Intelligent Agents*

General Terms

Algorithms, Experimentation

Keywords

Search, Alternative Plan, Agents, A*

1. INTRODUCTION

Suppose that an intelligent agent accepts as input a complete plan from a supervisor agent, human or automated (this is related to the "supervisor - subordinate" model from [3]). Provision of the complete plan implies that for the supervisor, *how* the goal is achieved is of some concern.

The plan is intended to achieve a goal state s_g, and the agent is assumed to be starting in state s_1. The plan originally input contains a sequence of states, and will be denoted here by $S = \{s_1, s_1, ..., s_n = s_g\}$ (we use both *plan* and *path* to denote a sequence of states). When the agent examines the plan, it realizes that the supervisor that generated it did not have accurate data on the current domain. The agent realizes that this particular plan from s_1 to s_g cannot be followed because of one of the following reasons:

1. The supervisor that generated the original path did not know the agent's exact initial location, and generated a plan for the wrong initial state.

2. One or more states in the plan do no exist in the given domain. For example, in the *blocks world* the plan might have the intermediate state $\{ON(A, B), ON(B, C), ON(C, D)\}$, but a stack of more than 3 blocks is not allowed (in our world). The supervisor that generated the plan was not familiar with this restriction.

3. One or more operators cannot be applied. For example, suppose that one of the operators was *cross the bridge to the island*, but the bridge has collapsed.

When the original plan cannot be followed, we would like the agent to search for an alternative path from its current state to the goal state. However, we make the following assumptions: (1) The most important consideration for the agent is to get to the *goal* state; (2) Since the input plan contains a complete sequence of states and not just the goal state, we assume that each of the intermediate states has some importance of its own. Otherwise, the supervisor could have given the *goal* state to the agent (allowing the agent to do its own planning from scratch).

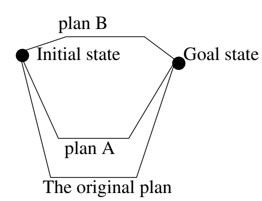

Figure 1: Plans

If there is more than one alternative plan that leads to the goal state, we would like the agent to pick the plan "closest" to the original plan among all potential candidates. This problem differs from classic search problems because there the agent has to find the shortest path to the goal, while in our problem it might pick a longer path which is, nevertheless, closer to the original path. Figure 1 shows an example of such a situation. Plan B is shorter than plan A, but we want our agent to pick plan A because it is closer to the original plan.

A real world example could be a police patrol that receives

AAMAS'03, July 14–18, 2003, Melbourne, Australia.

a command to go to a specific location. The command also instructs the patrol to go there via a specific route with special neighborhoods or locations along the way.

Two basic questions arise here, and answering these questions is the crux of this research. The first question is, given a number of alternative paths, which one should the agent follow? To answer this, a *distance metric* between plans or paths should be defined. The agent should pick the plan closest to the original according to such a distance function. Below, we will propose a number of distance functions with different characteristics. We study these functions and their respective attributes and advantages. We then try to answer the question of what distance function to use under what circumstances.

The second question the research addresses is, given any of the suggested distance functions, how does the agent efficiently find a path close to the original path? We have developed a search algorithm based on best-first heuristic search; this algorithm searches the domain (which is usually represented as a state graph) in order to find a path from the current initial location of the agent to the goal state. The most important attribute of our search algorithm is that it should prefer paths that are as close as possible to the original path. Thus, during the search, the cost function should try to predict whether the current path from the initial state to the current state is likely to be a prefix of a path that is close to the original path. States with such a likelihood will be favored to be expanded first. Below, we suggest a number of such heuristics and study their attributes.

We then present an implementation of our algorithm using both an instance of the blocks world domain, and Delaunay graphs (which simulate real-world road maps). Experimental data confirms the efficiency of our algorithm for finding closest paths.

2. RELATED WORK

The fact that planning agents work in dynamic environments, and therefore previously prepared plans cannot always be followed, has been understood by a variety of researchers. The field of *reactive planning* was developed to create an agent that can choose the best action for the state in which it finds itself, while avoiding the complexities of planning in dynamic, inaccessible environments. Important work in this field includes [5, 10].

Another related area is replanning and plan reuse. When an agent realizes that his intended plan cannot be followed, an alternative plan must be created. A search for a new plan consumes many resources, so the agent tries to use prepared plans. Replaning algorithms help the agent find the best way to get the original plan back on track, as in [9, 1]. Plan reuse and modification helps the agent choose one plan among several prepared plans, and to modify it to fit the current situation.

A focused example of plan reuse is presented in [11]. They create real maps in computer applications that automatically find good map routes. In their work, they accumulate and reuse previously traversed routes, thus demonstrating a route-planning method that retrieves and reuses multiple past routing cases that collectively form a good basis for generating a new routing plan. Other plan reuse work is presented in [7], where they present an integrated approach using knowledge about the road network, past cases, and an efficient search algorithm for route finding.

Other related work includes the Distributed Task Plan (DTP) [6]. A mobile agent plans a DTP according to the current user's objectives, and knowledge about current environments — not necessarily seeking the shortest path.

All the above work uses previously prepared plans in order to reduce the search effort and the complexity of creating a new plan from scratch. However, none of them assign importance to the intermediate states or actions of the original plan, as our work does.

Ephrati and Rosenschein [3] considered a situation consisting of two agents, one of them the supervisor and the other the subordinate. The supervisor created a plan for the subordinate, but the latter realizes that the supervisor's intended plan cannot be followed. Thus, he tries to find an alternative plan that will please the supervisor.

2.1 Distances Between Paths

Measuring distance between plans or paths is a non-obvious problem, and a great deal of work needs to be done so as to define a distance metric that will prefer plans that satisfy intuitive notions of closeness.

In their work, Ephrati and Rosenschein suggested a number of distance metrics between plans. One class of metrics they suggested compares the *cost* or the *utility* of the two plans. If, for example, $C(P)$ is the cost of plan P then the distance between the two plans will be $D_c(P_1, P_2) = |C(P_1) - C(P_2)|$. This class of metrics has a flaw from our point of view. We would like to define a function that measures the geometric distance between paths, because we assume that intermediate states have importance of their own. We can imagine two plans that lead us to the same goal, both having the same cost, but each going in an opposite direction. The *cost metric* described above will evaluate them as close to one another.

3. DISTANCE BETWEEN SETS OF STATES

Ephrati and Rosenschein also suggested another class of metrics that have more geometrical aspects. They suggested measuring distances between states of the two paths, and provided a number of ways for doing that.

For example, they suggested using the *Hausdorff metric*, which is used in the field of image processing to measure distance between two shapes. Taking the state in the first set that is closest to any state in the second set, one measures the distance between this state in the first set and the farthest state in the second set. They also suggested the *shifting distance metric*, in which measurement is done by summing the distance of each state in one plan from the set of states in the other (defining the distance between one state s' and a set of states, S, to be $D(s', S) = min_{s \in S} d(s', s)$).

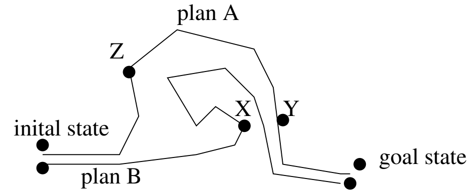

Figure 2: The Shifting Distance Metric

The Hausdorff metric loses a lot of information if we compare the distance of just one representative state. Moreover, both these metrics have a flaw because they treat a plan as a *set* of states while a plan should more correctly be considered a *sequence* of states (we would like the beginning of plan A to be close to the beginning of plan B, and so on). The *shifting distance* as defined in [3] might map a state at the beginning of one plan to a state at the end of the other plan, only because they are very close to one another. Figure 2 illustrates such a scenario. State X in plan B is very close to state Y in plan A, so the shifting distance will take their distance as the distance between X and plan A. However, in the time dimension state X of plan B should be mapped to state Z of plan A. In that sense, the best distance metric suggested by [3] was the *dynamic deviation metric*, which sums the sequential relative deviation produced by operators of each plan, and is defined as $D(P_1, P_2) = \sum_{i=1}^{m} d(p_i^1, p_i^2)$. This metric does treat a plan as a sequence of states, and thus the time dimension is taken into account. However, this metric is rather basic and can be enhanced further. Below, we introduce a number of distance functions that combine positive aspects from both the *dynamic deviation metric* and the *shifting metric*.

3.1 The Monotonic Mapping Function

In this section, we define the *monotonic mapping function* (MMF). We assume that there exists an efficient way to calculate distances between any two states in the search space (e.g., the straight line distance if the domain is a roadmap).

We denote the original plan by $S = \{s_1, s_2, ..., s_n\}$ and the agent's plan will be denoted by $A = \{a_1, a_2, ..., a_m\}$. Like the shifting distance above, we want to map a state of one path to the closest state in the other path, but also to take into consideration that a path is a sequence of states. This means that some kind of monotonicity must be maintained, i.e., if a node a_i was mapped to a node s_j, then nodes that appear later in A such as a_{i+1} cannot be mapped to nodes in S that appear before s_j. In other words, we cannot allow *crossings* in our mapping function. Such a crossing might occur in the shifting distance (as in Figure 2). We therefore introduce the MMF (Monotonic Mapping function) which calculates distances between paths as follows:

We first find all the different mappings of states of A into states of S, $F_{map} : A \rightarrow S$, that maintain the following *monotonic rule*: $F_{map}(a_i) \leq F_{map}(a_{i+1})$. This rule prevents crossings in the mapping function. For each of these different mappings, we define the distance between two paths according to this mapping to be $D(A, S) = \sum_{i=0}^{n} d(a_i, F_{map}(a_i))$.

There are many such mappings that obey this monotonic rule. A trivial example for such a mapping is that all the states of A are mapped into the same state of S. We want to choose the particular mapping that minimizes the summation of the distances between the states. Through simple mathematical analysis we can show that the number of potential mappings of m states of A into n states of S that obey the monotonic rule is $\binom{n+m-1}{n}$. However, the *best* such mapping can be found by a simple algorithm (based on dynamic planning), in time complexity of $O(n * m)$.

The above formulation, which takes the sum of the mapping of each of the states into a state in the target plan, has a flaw — two long paths might have a large distance between them, even though the states themselves are very close to one another. The solution would be to add these distances and then take their average. Therefore, the MMF is defined as follows:

$$MMF(A, S) = \frac{\sum_{i=1}^{m} d(a_i, Fmap_{min}(a_i))}{m}.$$

$Fmap_{min}$ is, of course, the best mapping among all mapping candidates $F : A \rightarrow S$ that obey the monotonic rule that $F_{map}(a_i) \leq F_{map}(a_{i+1})$.

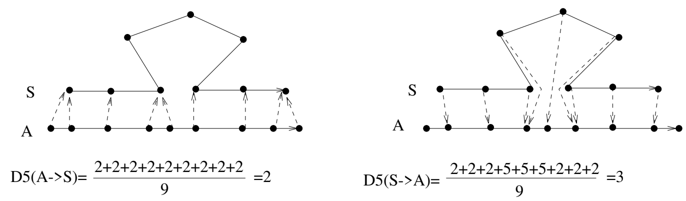

Figure 3: Mapping Both Ways

3.1.1 DMMF: Double Monotonic Mapping Function

We now have to answer the question of which path should be the source of the mapping, and which one should be the destination. In Figure 3, both plans have 9 states, but mapping $A \rightarrow S$ will yield $MMF(A, S) = 2$ while mapping $S \rightarrow A$ will yield $MMF(S, A) = 3$. Both mappings lose some information by having some states in the destination plan not have any state mapped into them. One solution is to take the sum or average of each of the mappings. This leads us to the *Double Monotonic Mapping function*: $DMMF(A, S) = MMF(A \rightarrow S) + MMF(S \rightarrow A)$.

3.1.2 IDMMF: Improved DMMF

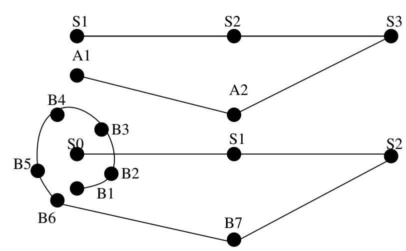

Figure 4: Mapping Both Ways, Again

Still, there are cases when the DMMF fails to provide a viable distance metric, as shown in Figure 4. There are two alternative paths in the figure: A is a simple path to the goal state, and B is another path that first circles the initial state of the original path. If we calculate the distance between the paths with DMMF (assuming that the distance between each of the states in the circle to s_1 is 5 and that the distance between a_2 and b_7 to s_2 is 20), then we get that $MMF(A, S) = \frac{5+30+0}{3} = 11.66$ while $MMF(B, S) = \frac{5+5+5+5+5+5+30+0}{8} = 7.5$. In this scenario, we will prefer a plan that circles around a state if it stays close to it, even though it is not moving forward. To solve this problem, we introduce the IMMF (Improved MMF) distance function. In IMMF, if we have a number of states mapped into the same state, we don't add them up but rather take their average. We divide all the states in A into different groups according to the state in S that they were mapped into.

For example, suppose that $a_{i_1}..a_{i_4}$ were all mapped into s_j. These nodes are all placed into a group that we call

G_j. We denote D_j as the average distance of nodes from G_j to s_j. If the number of different groups is N then we define $IMMF(A,S) = \frac{D_1+D_2+\ldots+D_N}{N}$. In Figure 4 the values of IMMF will be: $IMMF(A,S) = \frac{\frac{5}{1}+\frac{30}{1}+\frac{0}{1}}{3} = 11.66$ and $IMMF(B,S) = \frac{\frac{5+5+5+5+5+5}{6}+\frac{30}{1}+\frac{0}{1}}{3} = 11.66$

We then, once again, map the two paths both ways, and define a "double-version" of the mapping function:

$$IDMMF(A,S) = (IMMF(A,S) + IMMF(S,A))$$

3.2 TDMF: Time Dimension Mapping

In this section we suggest another distance metric which we call the *time dimension mapping function* (TDMF). In TDMF, the mapping is done according to the time dimension, i.e., each state in the original path is mapped to its relative state in the destination path according to the proportion of its relative location on the path. For each state a_i in A, we denote L_{a_i} as the length of a path from a_1 to a_i. We denote L_A as the length of path A, and $A[x]$ as the location on path A located at distance x from the beginning of the path. We map a state a_i from A to the corresponding state in S as follows: $F_{map}(a_i) = S[\frac{L_{a_i}}{L_A} * L_S]$. If there is no corresponding point at location $S[\frac{L_{a_i}}{L_A} * L_S]$), then we choose the neighboring state that minimizes the final TDMF mapping function. For the complete distance we define $TDMF(A,S) = \frac{\sum_{i=1}^{m} d(a_i, S[\frac{L_{a_i}}{L_A} * L_S])}{m}$.

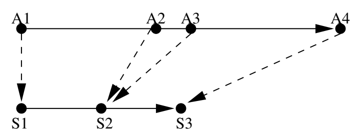

Figure 5: TDMF Mapping

Figure 5 shows a mapping according to the TDMF distance function. Node a_3 is closer to node s_3, but according to the time dimension it is mapped to node s_1, since both of them are located at the same relative location in their paths. Similar to the DMMF, we define $DTDMF(A,S) = TDMF(A,S) + TDMF(S,A)$. Here again we define ITDMF such that if more than one state is mapped into the same state, we take their average instead of their sum. The complexity of this mapping function is much smaller than IDMMF, as we only have to calculate the correct mapping for each of the states. This is unlike DMMF, where we had to look for the minimum among many relevant mappings.

3.3 Comparison of the Mapping Functions

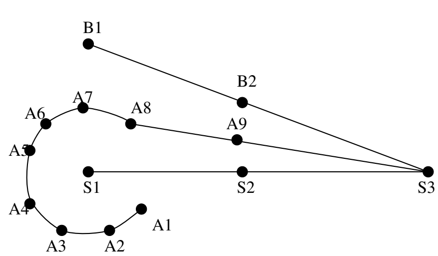

Figure 6: IDMMF versus ITDMF

We now compare the above two "improved" approaches, namely the IDMMF and ITDMF distance metrics. Figure 6 shows two alternative plans A and B. In A, there are many states that form a close circle around the initial state of the original plan s_1. In B, we have only one state b_1 near state s_1, but it is located quite far from s_1. With ITDMF, many of the states in the circle will have to be mapped to s_2. This will cause a large distance to be measured between the two paths. With IDMMF however, all the circle states will be mapped into s_1 and the distance between the two paths will be much smaller. Thus ITDMF will prefer plan B while the IDMMF function will prefer plan A.

The question of which metric is better is ultimately not a mathematical question, but might be situation dependent. If the desire is to always be located as close as possible to the original plan, then IDMMF will be preferred. If, however, the desire is to try to imitate the original plan in other aspects (like the number of states), and the time dimension is more important, then ITDMF is more suitable.

In fact, since "distance between paths" is not well-defined, other metrics might very well be considered, as long as these metrics do measure some kind of similarity between the two paths. For example, if we desire that the subordinate agent visit as many intermediate original states as possible, then we should give higher priority to plans that have as many such states as possible. This can actually be done rather easily by IDMMF, for example by taking the distance metric between two states to be $d(x,y) = 0$ if $x = y$ while $d(x,y) = K$ otherwise, for some constant K.

4. THE SEARCH PROCESS

We now discuss the search process that the agent should carry out so as to find the best alternative path as quickly as possible. We introduce a search algorithm which is based on best-first search. This search starts from the initial location of the agent and proceeds forward. A search tree is spanned until the goal node is found. The search can then continue until other paths to the goal are found. Each node in the search tree contains a partially developed path from the initial location of the agent. A cost function of a node in this search space will be an estimation of the likelihood of this partial path being a prefix of a plan which will be as close as possible to the original path.

The input for our problem is: (1) the graph that represents the domain; (2) the original path S; (3) the current location of the agent.

The output of our search algorithm is an alternative path which is as close as possible to the original path according to one of the distance metrics that were defined above. In order to find alternative paths to the goal state, we activate a search process from the initial state towards the goal state. At first glance, one might suggest the well-known A* [4] algorithm as a candidate for such a search. A* is a best-first search with a cost function of $f(n) = g(n) + h(n)$ where $g(n)$ is the elapsed distance from the initial state to the current node n and $h(n)$ is an estimation of the distance between n and the goal state. However, such a heuristic and the corresponding cost function is not suitable for our needs as we are not interested in the shortest path to the goal but rather in a path to the goal that will be as close as possible to the original path.

A heuristic function for our problem should be defined so that the search will lead towards the goal, but the search should prefer to expand those branches that are most promising — in the sense that those branches are likely to be prefixes of the paths that are very close to the original path.

In a classic heuristic function of A*, if the estimation is always a lower bound of the exact distance to the goal, then we say that the heuristic is *admissible*. With an admissible heuristic the complete cost function, $f = g + h$, is *monotonically increasing* and is always a lower bound on the shortest path. Thus, the first solution that is found by A* is the optimal solution. A hidden condition for the fact that the cost function will be monotonically increasing is that the g part is always increasing as we proceed with the search.

In our case, we do not divide the cost function into g and h, but rather want to try to estimate the distance of the partial path to the original path. Since our defined distance functions take the average of all the mappings between the two paths, then the distance between the paths might decrease during the course of the search. This happens if a path starts at a large distance from the original path but gets closer in later stages. Thus, a heuristic that will always be a lower bound is not possible here. Therefore, due to the nature of our distance metrics, we cannot guarantee that the first solution that will be found will be the best one available. However, our experimental results showed us that many times it is.

4.1 IDMMF as Heuristic Evaluation Function

We now turn to defining the heuristics. As a first step, we would like to take the IDMMF distance of a partial path from the *complete* original plan. However, such a heuristic function results in a problem which we call the *tail problem*.

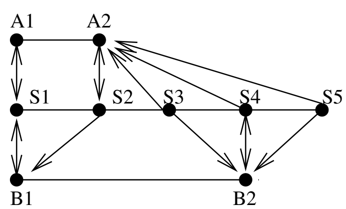

Figure 7: Tail Problem

In Figure 7 we have an original path S and two partial paths A and B. Suppose that both partial paths were developed and now we have to decide which one looks more promising. Suppose that for the heuristic we take the regular IDMMF function and measure the distances between each of these paths to S. A problem arises when we make the reverse mapping of the original path S into A. Since A is only a partial path, all the states in the tail of S are forced to be mapped into a_2 even though they are located very far from it. For the partial plan B, however, b_2 is closer to the middle of S and states from the tail of S are mapped better (closer) to B. Thus, using IDMMF the process will prefer plan B, while it is obvious that plan A, though partial, better imitates the behavior of the original plan.

4.2 The Prefix Heuristic (PH)

A possible solution to handling the tail problem is to consider for purposes of the heuristic value only the mapping of those states of the original path S that are close to the relevant prefix of the alternative path. We call this method the *prefix heuristic* (PH), which carries out the following steps:

1: First, a full mapping is done from A to S. Let s_k be the last state from S that any node from A was mapped into.

2: We now only map $s_1, s_2 \ldots s_k$ into A. Other nodes of S are not mapped, since they are in the tail of S and are far from A.

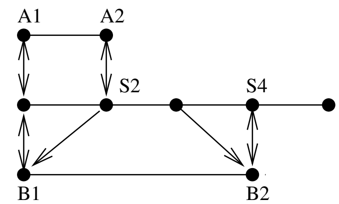

Figure 8: The PH Heuristic

For example, in Figure 8 the mapping of the original path S into the partial alternative path $A = \{a_1, a_2\}$ only considers nodes s_1 and s_2 of the original path S, since these are the nodes that are targets of mappings from A, while the tail nodes of S are not mapped. With this formulation, the mapping of the original path S into $B = \{b_1, b_2\}$ considers the nodes s_1, s_2, s_3 and s_4 of the original path S. With this calculation, plan A will be preferred because it is closer to the relevant part of S.

4.3 PH for the ITDMF distance function.

As explained above, the TDMF maps each state in A into its relative state in S according to the proportion of its relative location on the path. States in S are mapped into states of A in the same manner. In order to do that, the length of both paths should be known. However, during the course of the search the length of the full alternative path A is not known, since we only have a prefix of A. Therefore, we need to estimate the length of the entire alternative path.

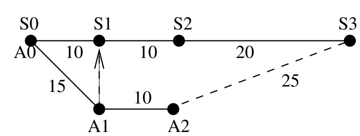

Figure 9: Estimating the Length of A

For any node v in A, we define $Parent(v)$ as a parent node of node v in A. For any two nodes x, y in A, we define $d(A, x, y)$ to be the total distance in A from x to y. We propose three approaches for estimating the length of the alternative path A when we want to develop node v in a prefix of path A.

1: We define $Ls_A(v) = d_A(start, parent(v)) + d_S(Fmap(parent(v)), Goal)$. In this case, since we do not know the length of the tail of A, we replace it by the relevant tail of S. For example, in Figure 9 we are now developing node a_2 in the search tree. Its parent is node a_1, and it was mapped to s_1. According to our definition, we get $Ls_A(a_2) = d_A(a_0, a_1) + d_S(s_1, Goal) = 15 + (10 + 20) = 45$.

2: We define $La_A(v) = d((A, Start, v) + SLDist(v, goal)$. In this approach, we just take the straight line distance (SLDist, which we assumed above is easily calculated) from v to the goal as an estimation of the tail of A. In Figure 9 we get $La(a_1) = (15 + 10) + 25 = 50$.

3: The third approach simply takes the maximum of the two approximations above: $Lmax_v = max(Ls_v, La_v)$.

5. A BLOCKS WORLD EXAMPLE

We presented a number of geometric metrics for measuring the difference between two plans. We now demonstrate how these metrics can be used for a domain which is not in itself geometric, namely a modified blocks world.

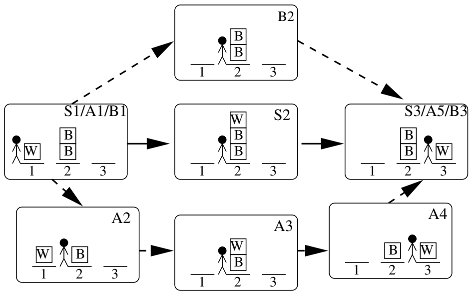

Figure 10: A Modified Blocks World Example

Consider the blocks world in Figure 10 which is described by the following predicates:
Blocks(n) - The total number of blocks in the world.
On(A, B, S) - Block A is on block B and located at slot S.
Color(A,C) - The color of block A is C.
At(S) - The robot is at slot S.
The domain operators are:
Go(S, T) - The robot goes from slot S to slot T.
Carry(A, S, T) - The robot carries block A from the top of slot S to the top of slot T.
Destroy(S) - The robot destroys the block at the top of slot S.
Create(A,S,C) - The robot creates a new block A at the top of slot S with color C.

Note that there are three different plans in Figure 10 that all move the white block from slot 1 to slot 3. In the original plan S, the robot takes this block and puts it on top of the black blocks in slot 2 and then moves it again to slot 3. Suppose that there is a restriction on the number of blocks, and a restriction that a stack of three blocks is not allowed. Both alternative plans A and B obey these restrictions. In plan A, the robot first destroys the black block at slot 2, moves the white block to its destination, and then creates the black block again. In plan B the agent destroys the white block in slot 1 and creates it again in slot 3.

We would like to use the DMMF function and measure the distance between these two plans. For that purpose we should define a distance metric between the states. There are several sensible ways to define such a distance metric in this domain. We introduce two metrics that were also proposed in [3].

Since we consider each state to be described by a set of predicates, a straightforward metric can be generated by the summation of the differences between the conflicting predicates of the two states in question. In our example the difference between different predicates can be defined in the following way:
$Diff(Blocks(N), Blocks(M)) = 3 * |N - M|$
$Diff(At(S), At(T) = |S - T|$
$Diff(On(A, B, S), On(A, C, T)) = 2 * |S - T|$

Another approach would be that the distance between any two states is defined by the plan needed to bring about the differing predicates of one state starting from the other. Therefore, we define $d(x, y)$ to be the minimal plan from x to y. In this example, we set the following values for the different operators: $Destroy = 2$, $Create = 3$, $Go(S, T) = |S - T|$ and finally $Carry(A, S, T) = 2 * |S - T|$.

By defining a distance function between any two states in the blocks world, we make it possible to use any of the geometric distance functions defined in the previous section in the context of the blocks world.

5.1 Using the IDMMF in the Blocks World

Here we describe the steps of our algorithm on the example of Figure 10. For the distance metric between the states we use a plan that unifies the two states. For the distance between paths and for the PH heuristic we use the IDMMF distance function.

The purpose of the search is to find an alternative plan that is "as close as possible" to the original plan S. During the search, a tree is developed and at every stage of the search each leaf defines a unique branch. Each branch is given a heuristic value. Based on this heuristic, we activate a best-first search that prefers those branches that are most likely to be the prefixes of the plans that are very close to the original plan.

The search starts at state s_1. According to the best-first search, state s_1 is expanded and its two neighbors a_2 and b_2 are generated. In order to decide which state to expand first, a_2 or b_2, we use our PH heuristic function. The branches are $\{a_1, a_2\}$ and $\{b_1, b_2\}$. The closest branch to the original plan according to the heuristic will be expanded first.

We now calculate the heuristic of the branch $\{A = a_1, a_2\}$. For this we first need to map the states of A into the states of S. a_1 is mapped into itself (s_1) . For a_2 we need to find the closest state in S. We get the following:
$d(a_2, s_1) = Destroy(2) + Go(2, 1) = 3$
$d(a_2, s_2) = Create(A, 2, Bl) + Go(2, 1) + Carry(A, 1, 2) = 6$
$d(a_2, s_3) = 13$

Thus, a_2 is also mapped into s_1. Since s_1 was the last state in S that was a destination, then for the reverse mapping we only need to map this node, s_1, to the closest node from a_0 and s_1, a_2. Of course, since a_1 and s_1 are identical, it is mapped to itself. Thus we get that
$ph(\{a_1, a_2\}, S) = \frac{F_map(\{a_1,a_2\} \to S_1)}{2} + \frac{F_map(S_1 \to \{a_1,a_2\})}{1} = 2.$

In the same manner, we need to find the distance between $\{b_1, b_2\}$ and the original plan S. Similar calculations show that $ph(\{b_1, b_2\}, S) = 4.5$. Thus, the node in the search tree that includes $A = \{a_1, a_2\}$ is selected and expanded. However, in the end we get that plan B is closer, and this is the plan that is chosen.

6. EXPERIMENTAL RESULTS

We have experimented with Delaunay graphs [8], which simulate roadmaps. Delaunay graphs comprise Delaunay triangulations of planar point patterns that are generated by a *Poisson point process* [8] that distributes points at random over a unit square using a uniform probability density function. Delaunay triangulation of a planar point pattern is constructed by creating a line segment between each pair of points *(u,v)*, such that there exists a circle passing through u and v that encloses no other point. This characteristic simulates roadmaps. To construct Delaunay graphs for our experiments, we used the Qhull software package [2], which generates Delaunay triangulations on a square frame with unit size of 1.

In Delaunay graphs, nodes are connected to the nodes that are near them. In roadmaps, however, nearby locations might not have roads between them, perhaps due to some sort of an obstacle like a mountain or a river. To imitate this characteristic in our graphs, we deleted some of the edges in some of our experiments. Another characteristic of roadmaps is the existence of highways that connect distant locations. To add this effect to our graphs as well, we added additional random edges to the Delaunay graphs in some of the experiments. The size of the graph, as well as the number of edges that we add or delete, are variables.

6.1 Creating the Environment

Once the domain graph was created, we used two methods for creating the original path:

(1) Shortest path: The start and goal nodes are randomly selected. Then, the A* algorithm is carried out and the shortest path between the start and goal nodes is used for the original path.

(2) Random path: Here, we first determine the number of states in the path. Then, we choose the start node and make a number of random moves to determine the intermediate states as well as the goal state.

At this stage, we have to make some changes to the graph (which simulates environmental changes in the domain) to prevent the original path from being followed (and force the search for an alternative). Thus we randomly delete vertices and edges from both the original plan and the whole graph.

The size of the graph, the length of the original path, and how many changes will be made in the graph are all parameters of the system. We experimented with many values for these parameters. Below, we present experiments where the values were determined as follows. The number of edges in the Delaunay graph was varied from 200 to 6000. Then, 50% of the edges were deleted and 20% of random edges were added, by randomly choosing two nodes in the graph and creating an edge between them. After the original path was determined, 40% of the vertices of the original path were marked and all the edges that connected them were deleted. Then, given an instance of a problem, a best-first search with the different heuristic functions and the different metrics was activated.

While we tested both the IDMMF and the ITDMF, as well as many different heuristics, due to space constraints we present only the data on ITDMF and our PH heuristic function in this paper. However, similar results were obtained for the IDMMF and for other heuristics.

6.2 Original Path Created with A* Algorithm

Figure 11 presents a graph with 200 nodes. The dashed line is the original path created by A*. The black line is the alternative path that was found by running our algorithm. The black square nodes are the nodes that were developed during the search. Our alternative path was found quickly, as evidenced in the figure by the fact that only a small number of nodes were generated by our algorithm which are all located close to the original and alternative paths.

As mentioned above, we have three approaches for a heuristic function, based on three different methods to estimate the length of the alternative path. Figure 12 presents the number of nodes generated by each of the heuristics until the first alternative path was found. The distance function is ITDMF. All datapoints are the average of 300 random instances with the same characteristics. The figure

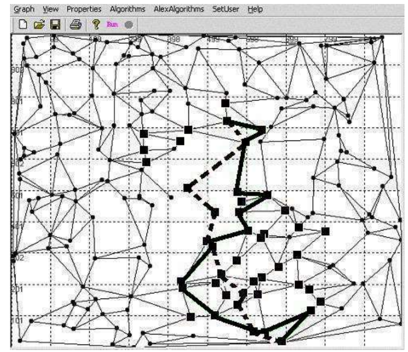

Figure 11: Alternative and Original Paths

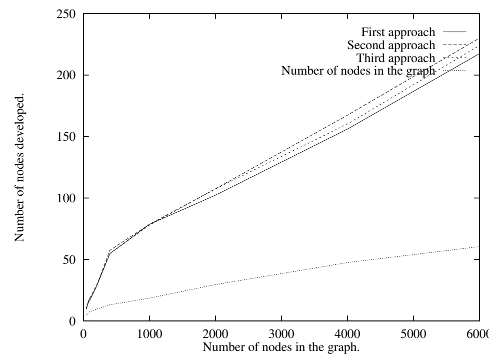

Figure 12: Number of Generated Nodes, ITDMF

also presents a curve with the average number of nodes of the original path. It shows that all three heuristics behave about the same, and they all generate only four times as many nodes as the original path. For example, for a graph of size 6000 the length of the original path was 50. Our algorithm needed to generate only 200 nodes in order to find an alternative path. This shows that our algorithm is quite efficient and finds the alternative path rather quickly.

As discussed above, our heuristics cannot guarantee the best alternative path since they are not admissible. However, when we continued to run our algorithm, more alternative paths were created, and experiments confirmed that the optimal alternative path was found very quickly. In most runs, it was either the first or the second alternative path that was found. We found that the distance between the first solution that was found by our algorithm to the original path is larger than the distance between the optimal solution to the original path by an average of only 2.2%.

Since the original path was found by A* and was a shortest path between the two nodes in the original graph, one might consider running A* on the current graph (i.e., after all the changes were made) in order to determine whether the current shortest path is also close to the shortest original path. Figure 13 presents two curves which measure the distance between an alternative plan and the original plan. The first curve presents solutions that were found by our algorithm. The second curve presents shortest paths that were found by A* after the graph was changed. The distance is measured according to the ITDMF distance function. Both algorithms found the goal node rather fast and both generated approximately 4 times as many nodes as the size of

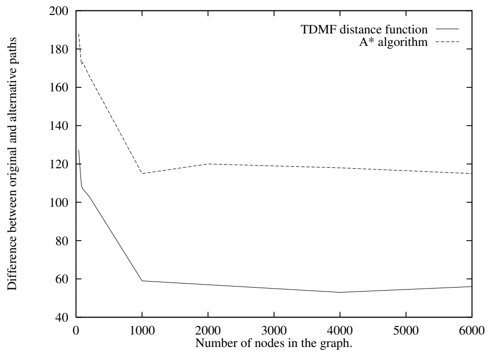

Figure 13: ITDMF versus A*

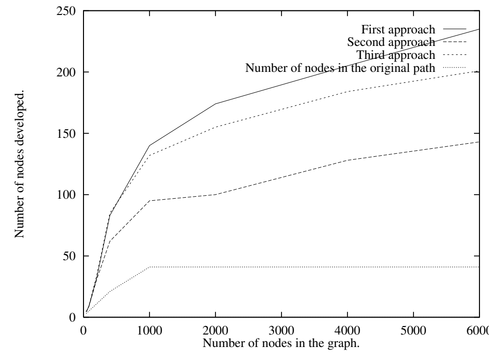

Figure 15: Number of Generated Nodes, ITDMF

the original path. However, Figure 13 clearly shows that the alternative path found using the ITDMF distance function is always closer to the original path than the alternative path found by running the A* algorithm, even though the original path was created with the A* algorithm.

6.3 Randomly created original path

We now consider our experiments that used original paths that were generated by a random walk on the graph (and are not the shortest paths, as above). The number of random walks was 5% of the number of nodes in the graph for graphs with less than 1000 nodes, and a constant of 40 nodes for larger graphs.

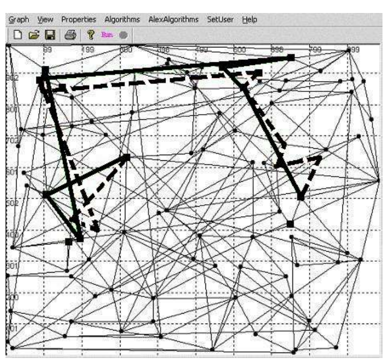

Figure 14: Randomly Created Original Path

Figure 14 presents an original path (the dashed line) and an alternative path found by our algorithm (the black line). Again, the similarity of the two paths is clear.

Figure 15 presents the number of nodes generated by each of our approaches (for measuring the estimated length of the alternative path) compared to the number of nodes in the original path. The quality of the first solution found by all these approaches was the same, and again, was larger than the optimal solution by no more than 2.2%. However, the difference in the speed of reaching the goal between the three approaches is much larger than the previous case (where the shortest path was used for the original path) and it seems that the combined version, which takes the maximum, produced the best results and was the fastest. Here again the number of generated nodes was never larger than the number of nodes in the original by more than a factor of 5, which confirms that the alternative path was found rather quickly by all three approaches.

7. CONCLUSIONS

We have presented an algorithm that efficiently finds an alternative path which is close to an original path. We suggested two metrics for the purpose of assessing path "closeness", and presented a heuristic function to guide the search. Results confirm that such an alternative path is reached relatively efficiently.

8. REFERENCES

[1] J. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *Proceedings AAAI-88*, pages 735–740, St. Paul, Minnesota, 1988.

[2] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hull. *Geometry Center Technical Report GCG53*, 1993.

[3] Eithan Ephrati and Jeffrey S. Rosenschein. Planning to please: Following another agent's intended plan. *Group Decision and Negotiation*, 2(3):219–235, 1993.

[4] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SCC-4(2):100–107, 1968.

[5] R. M. Jensen and M. M. Veloso. OBDD-based universal planning: Specifying and solving planning problems for synchronized agents in non-deterministic domains. *Artificial Intelligence Today, Recent Trends and Developments*, pages 212–248, 1999.

[6] W. Li and M. Zhang. Distributed task plan: A model for designing autonomous mobile agents. In *Proceedings of International Conference on Artificial Intelligence*, pages 336–342, Las-Vegas, 1999.

[7] B. Liu. Intelligent route finding: Combining knowledge, cases and an efficient search algorithm. In *Proceedings of ECAI-96*, pages 380–384, Budapest, Hungary, 1996.

[8] A. Okabe, B. Boots, and K. Sugihara. Spatial tessellations, concepts, and applications of Voronoi diagrams. *Wiley, Chichester*, 1992.

[9] S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 1995.

[10] A. Stentz. Optimal and efficient path planning for partially-known environments. In *Proc of ICRA*, pages 3310–3317, 1994.

[11] K. Zita and M. Veloso. Route planning by analogy. *In the Proceedings of the International Conference on Case-Based Reasoning*, 1995.