Searching for Close Alternative Plans

Ariel Felner[1], Roni Stern[1], Jeffrey S. Rosenschein[3], and Alex Pomeransky[2]

[1] Department of Information Systems Engineering, Ben-Gurion University, Beer-Sheva, Israel
[2] Department of Computer Science, Bar Ilan University, Ramat Gan, Israel
[3] School of Engineering and Computer Science, Hebrew University, Jerusalem, Israel

Abstract. Consider the situation where 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 implemented by the agent, who then goes about trying to find an alternative plan that is as close as possible to the original. To achieve this, a search algorithm that will find similar alternative plans is required, as well as some sort of comparison function that will determine which alternative plan is closest to the original.
In this paper, we define a number of *distance metrics* between 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 efficiently find the most suitable alternative plan. We also propose a number of heuristics for the cost function of this best-first search algorithm. To explore the generality of our idea, we provide three different problem domains where our approach is applicable: physical roadmap path finding, the blocks world, and task scheduling. Experimental results on these various domains support the efficiency of our algorithm for finding close alternative plans.[4]

Keywords: Best-first Search; Shortest path; Scheduling; Replanning; Rescheduling

[4] A preliminary version of this paper appeared in the International Joint Conference on Autonomous Agents and Multiagent Systems [8].

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 Ephrati and Rosenschein's work [6]). Provision of the complete plan implies that for the supervisor, *how* the goal is achieved and the exact intermediate states are of some concern.

A real world analogy might be a police patrol that receives a command to go to a specific location. The command also instructs the patrol to get there via a specific route, with special neighborhoods or locations along the way. While getting to the final location is important, the path that is taken is also specified, and of some independent value.

The plan is intended to achieve a goal state s_g, and the agent is assumed to be starting in state s_1. The original plan that was given contained a sequence of states, and will be denoted here by $S = \{s_1, s_2, ..., s_n = s_g\}$ (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 about 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:

- 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.
- One or more states in the plan do not 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 might not be allowed (in our world). The supervisor that generated the plan was not familiar with this restriction.
- 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, the agent tries 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 might have given the *goal* state to the agent (allowing the agent to do its own planning from scratch).

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 that 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.

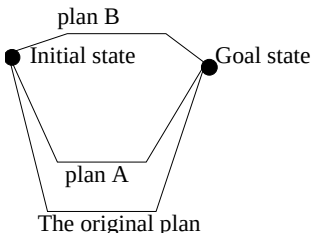

Fig. 1. Plan B is shorter, but plan A is closer to the original plan

In a way, we can view the states of the original plan as subgoals, or as goals with lower priority than that of reaching the goal state. For the police patrol example mentioned above, the goal is to reach the desired location, but there are subgoals, namely to get there through specific neighborhoods or locations. Thus, the search should be guided towards the goal state but should be biased in favor of paths or prefixes of paths that are close to the original path. Note that neither the original plan nor its alternative are necessarily the shortest path to the goal.

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. We show in this paper that while there is a general theme among distance functions, there are many domain dependent issues that need to be considered. In Section 3, we propose a number of distance functions with different characteristics that are applicable in different problem domains and circumstances. We study these functions and their respective attributes and advantages, and try to give general guidance as to 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 general search algorithm based on best-first heuristic search; this algorithm searches the domain (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.

To show the generality of our approach, we then present implementations of our algorithm in three different domains: 1) physical roadmaps modeled by random Delaunay graphs (which simulate real-world road maps), 2) the blocks world domain, and 3) task scheduling. Experimental results support the efficiency of our algorithm for finding closest paths in all three domains.

The paper is organized as follows. Section 2 describes related work. Section 3 introduces the various distance functions between plans. Our search algorithm is introduced

in Section 4. Implementation and experimental results on the three different testbed domains are provided in Sections 5, 6, and 7. Finally, our conclusions, and possibilities for future work, are discussed in Section 8.

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* considers agents that must choose the best action for the state in which they find themselves, while avoiding the complexities of classical planning in dynamic, inaccessible environments. Important work in this field includes the work of Jensen and Veloso [13], Stentz [31], and Koenig and Likhachev [16].

Another related area is replanning and plan reuse. When an agent realizes that its intended plan cannot be followed, an alternative plan must be created. The search for a new plan consumes many resources, so an agent may try to use prepared plans. Replanning algorithms help the agent find the best way to get the original plan back on track, as in the work of Russell and Norvig [27], Ambros-Ingerson and Steel [1], Koenig and Likhachev [15], and Brock and Oussama [5]. Plan reuse and modification help an agent choose one plan among several prepared plans, and to modify it to fit the current situation [25].

A focused example of plan reuse is presented in the work of Haigh and Veloso [9]. 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 has been done by Liu [19], where an integrated approach is explored, using knowledge about the road network, past cases, and an efficient search algorithm for route finding.

Lifelong-planning A* [16] and its dynamic version D*-lite [15] also assume that a path to the goal has already been found, but that the environment has changed. Instead of starting a new search from scratch, these algorithms try to use old parts of the open-list (of A*) for the new search on the changed graph whenever possible.

Other related work includes the Distributed Task Plan (DTP) [18]. 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.

2.1 Plan Distance Metrics

Ephrati and Rosenschein [6] 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, the subordinate tries to find an alternative plan that will please the

supervisor, which *inter alia* involves assessing the distance between plans. 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 $D_c(P_1, P_2) = |C(P_1) - C(P_2)|$. Since we assume that intermediate states have importance of their own, this class of metrics is obviously unsuitable. Imagine two plans, both leading to the same goal and having the same cost, but each plan reaches the goal via a completely different path (via different intermediate states). The *cost metric* described above will evaluate them as equal, disregarding the difference in the intermediate states.

Related research was carried out by Mayers and Lee [24]. In their work, the intention is to generate a set of qualitatively different plans, from which the user can pick one. This was achieved by defining metrics that estimate a value for a set of plans, and using biases created from metadata over the domain to direct a planner towards different sections of the search space. The metric used to estimate a value for a set of plans was based on the distances between plans in the set. In their work, the distance metric between two individual plans is similar to the *dynamic deviation metric* [6], discussed below. An interesting follow-on to Mayers and Lee's research would be to employ their framework to generate sets of plans, but to use the newer distance metrics discussed in this paper.

Line generalization is yet another subject that is related to comparing a set of points. In line generalization, the objective is to take a line with n points and create an approximation of that line with m points. As with a set of states in a plan, there are many ways to measure how close one set of points is to another set of points. A large variety of line generalization methods and measures of line approximations have been discussed by McMaster [21–23].

3 Distance between Sequences of States

Ephrati and Rosenschein [6], in addition to the cost metric mentioned above, also explored 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.

As one example, they considered using the *Hausdorff metric* [11, 2, 3], 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.

More formally, if P_1 and P_2 are two plans then $shiftingDistanceMetric(P_1, P_2) = \sum_{i=1}^{n} D(p_i^1, P_2)$ where p_i^1 is the i-state in the plan P_1, $D(x, P) = min_{p' \in P} d(x, p')$, and $d(x, y)$ is the distance between two nodes.

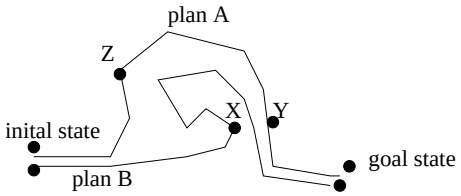

Fig. 2. The shifting distance metric

The Hausdorff metric loses a lot of information, as we calculate the distance of just one representative state. Moreover, both these metrics have an inherent 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* [6] 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, considering the internal order of the plans, state X of plan B should be mapped to state Z of plan A.

In that sense, the best distance metric suggested by Ephrati et al. [6] was the *dynamic deviation metric*, which sums the sequential relative deviation produced by operators of each plan, and is defined as $D_{ddv}(P_1, P_2) = \sum_{i=1}^{m} d(p_i^1, p_i^2)$, where p_i^1 is the i-state in P_1, p_i^2 is the i-state in P_2, and m is the minimum between the number of states in the two plans.

This metric does treat a plan as a sequence of states, and thus the internal order is taken into account. However, this metric is rather basic and can be further enhanced. 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 assume that the original plan has n states and denote it by $S = \{s_1, s_2, ..., s_n\}$. We also assume the agent's plan has m states and denote it 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. We therefore use the following restriction on the allowed mapping:

The monotonic restriction: 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. Mapping functions between the paths A and S, $F_{map} : A \rightarrow S$, that keep this restriction must maintain the following **monotonic rule**: $F_{map}(a_i) \leq F_{map}(a_{i+1})$. 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 *Monotonic mapping function* (MMF), 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 monotonic rule (i.e., that $F_{map}(a_i) \leq F_{map}(a_{i+1})$. For each of these different mappings, we define the distance between two paths according to this mapping to be

$$D_{map}(A, S) = \sum_{i=0}^{m} d(a_i, F_{map}(a_i)).$$

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 programming), in time complexity of $O(n * m)$. The main idea of the algorithm is to have a two-dimensional array of size $m \times n$. Each entry x, y in this array corresponds to the best monotonic mappings of the first x state in A into the first y states in S. An entry x, y is easily calculated from its predecessors. The exact details and complexity analysis of the algorithm are provided by Felner [7].

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:

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

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

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. However, as shown in the figure, the states at the beginning and end of S are two moves away from states in A, and the states in the middle of S are three moves away.

Note that mapping $A \rightarrow S$ will yield $\textit{MMF}(A, S) = 2$ while mapping $S \rightarrow A$ will yield $\textit{MMF}(S, A) = 3$. Both mappings lose some information by having some states in the destination plan with no 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*:

$$\textit{DMMF}(A, S) = \textit{MMF}(A \rightarrow S) + \textit{MMF}(S \rightarrow A).$$

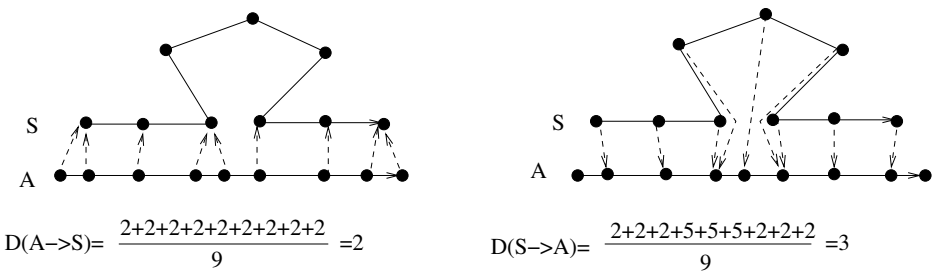

Fig. 3. Mapping both ways

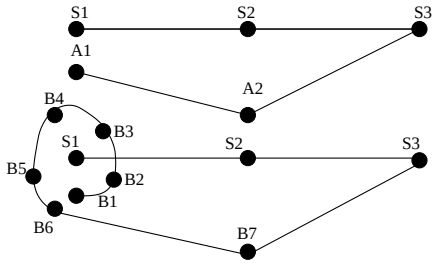

Fig. 4. Mapping both ways, again

IDMMF: Improved DMMF 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 30), then we get that $\mathit{MMF}(A,S) = \frac{5+30+0}{3} = 11.66$ while $\mathit{MMF}(B,S) = \frac{5+5+5+5+5+5+30+0}{8} = 7.5$. 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 do not sum them, 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.

We define G_j as follows: given an original plan S, an alternative plan A, and a mapping F_{map} between A and S, then for every $s_j \in S$ we define the group G_j=$\cup a_i \in A | F_{map}(a_i) = s_j$. For example, suppose that $a_{j_1}..a_{j_4}$ 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

$$\mathit{IMMF}(A,S) = \frac{D_1 + D_2 + \ldots + D_N}{N}$$

In Figure 4 the values of IMMF will be: $\mathit{IMMF}(A,S) = \frac{\frac{5}{1}+\frac{30}{1}+\frac{0}{1}}{3} = 11.66$ and $\mathit{IMMF}(B,S) = \frac{\frac{5+5+5+5+5+5}{6}+\frac{30}{1}+\frac{0}{1}}{3} = 11.66$. All the states in the circle of plan B are now combined into one group, and therefore both plans have the same distance to the original plan S. We then, once again, map the two paths both ways, and define a "double-version" of the mapping function:

$$\mathit{IDMMF}(A,S) = (\mathit{IMMF}(A,S) + \mathit{IMMF}(S,A))$$

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 state 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

$$\mathit{TDMF}(A,S) = \frac{\sum_{i=1}^{m} d(a_i, S[\frac{L_{a_i}}{L_A} * L_S])}{m}$$

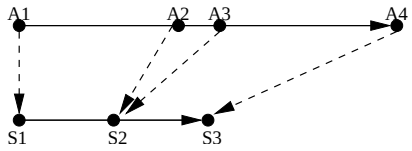

Fig. 5. TDMF mapping

Figure 5 shows a mapping according to the TDMF distance function. State a_3 is closer to state s_3, but according to the time dimension it is mapped to state s_2, since both of them are located at the same relative location in their paths. Again, similar to DMMF, we define

$$\mathit{DTDMF}(A,S) = \mathit{TDMF}(A,S) + \mathit{TDMF}(S,A)$$

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 between the Distance Metrics

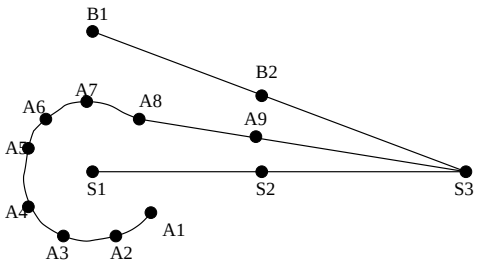

Fig. 6. IDMMF versus ITDMF

We now provide an example for comparing 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 "circle" close to the initial state of the original plan S_1 (i.e., A_1 through A_8). 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 of A (e.g., A_4, A_5 and A_6) 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 is 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 describe 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 that is based on best-first search, which starts from the initial state and proceeds until the goal state has been reached.

The input for our problem is:

1. The graph that represents the domain;
2. The original path S;
3. The selected distance metric.

The output of our search algorithm is an alternative path from the initial state to the goal state that is as close as possible to the original path, according to the selected distance metric (chosen from the metrics that were defined above).

4.1 The Search Algorithm

To find such an alternative path to the goal state, we activate a best-first search process from the initial state towards the goal state. This search starts from the initial location of the agent and proceeds forward; thus, a search tree is spanned. Each node in the search tree contains a partially developed path from the initial location of the agent. A best-first search chooses to expand a node that minimizes some cost (or evaluation) function. 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 that will be as close as possible to the original path. A number of relevant cost functions that estimate this are presented below. At each expansion cycle, we expand next the state with the lowest such estimation, until the goal state has been reached for the first time. When that happens, we return the branch of the search tree with the goal node as the alternative path to the goal. Note that the search need not stop at this point, and that it could continue in the same manner until other alternative paths to the goal have been found.

4.2 The Cost Function for the Search

How do we define a cost function to estimate the likelihood of a partial path being a prefix of a path that will be as close as possible to the original path? At first glance, one might suggest using the same cost function as that of the well-known A* algorithm [10] for our best-first 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 estimate of the distance between n and the goal state.

To return the optimal path, A* has some conditions that do not apply in our domain. In a classic heuristic function of A*, if the estimate 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 component 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 state mappings between the two paths, 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. As a consequence,

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 that it often is.

4.3 IDMMF as Heuristic Evaluation Function

We now turn to defining the heuristics used in our best-first search. 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 that we call the *tail problem*.

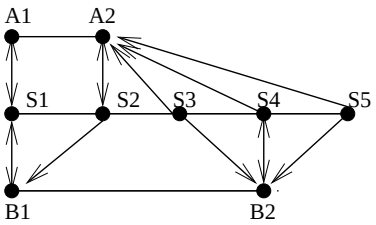

Fig. 7. Tail problem

In Figure 7 we have an original path S and two partial paths A and B. Suppose that both partial paths were developed (after generating two states for each), 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.

4.4 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.

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

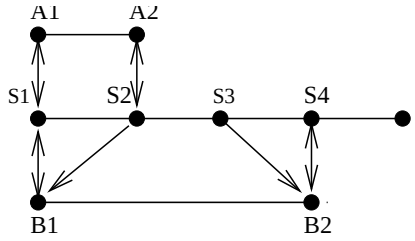

Fig. 8. The PH heuristic

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.

4.5 PH for the ITDMF Distance Function

As explained above, ITDMF 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, for properly using the prefix heuristic with the ITDMF distance metric we need to estimate the length of the entire alternative path while only given its prefix.

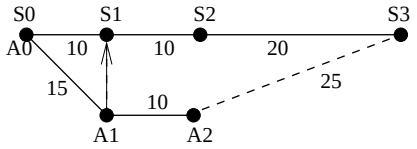

Fig. 9. Estimating the length of A

For any node v in A, we define *predecessor*(v) as the node that appears in A just before v. 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 possible complete alternative path A when we want to develop node v in the current prefix of path A that ends at node v.

Using the Exact Tail in S In this case, since we do not know the length of the tail of A, we replace it by the relevant tail of S. Since v was not mapped yet, we consider the tail of A from its predecessor. We therefore define the length of the estimated complete alternative path $L1_A(v) = d_A(start, \textit{predecessor}(v)) + d_S(F_{map}(\textit{predecessor}(v)), \textit{Goal})$. 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 $L1_A(A_2) = d_A(A_0, A_1) + d_S(S_1, S_3) = 15 + (10 + 20) = 45$. Note that the dashed line in the figure is not an edge, but only signifies that the actual tail would be 25 if we knew it — but we estimate it here as 20.

The Approximated Tail in A In this approach, since v is the last state in the prefix, we just take the straight line distance (which we will denote SLDist, and which we assumed above is easily calculated) from v to the goal as an estimation of the tail of A. That is, we define $L2_A(v) = d_A(\mathit{Start}, v) + \mathit{SLDist}(v, \mathit{Goal})$. In Figure 9, we get $L2_A(A_2) = (15 + 10) + 25 = 50$.

Maximum of Both The third approach simply takes the maximum of the two approximations above: $\mathit{Lmax12}_A(v)$=$\max(L1_A(v), L2_A(v))$.

These estimated lengths of A ($L1$, $L2$, or *Lmax12*) are then used as the length of A when calculating the PH heuristic (as described above) for ITDMF.

5 Experimental Results for Simple Path Finding

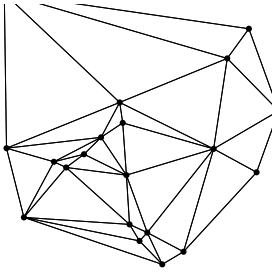

Fig. 10. Delaunay graph with 20 nodes

We have experimented with Delaunay graphs [26], which simulate roadmaps. Delaunay graphs comprise Delaunay triangulations of planar point patterns that are generated by a *Poisson point process* [26] 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 [4], which is a well known package that generates Delaunay graphs 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. Figure 10 illustrates a Delaunay graph with 20 nodes.

5.1 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 the graph (some of these edges and vertices were used by the original plan).

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 graphs were built 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. On this graph we determine the original path according to the two methods presented above. After the original path was determined we change the graph as follows. 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 on this modified graph.

We tested both the IDMMF and the ITDMF, as well as other distance metrics that we do not mention here (we present only the data on ITDMF and our PH heuristic function in this paper). However, similar results and conclusions were obtained for the IDMMF distance metric and for the other distance metrics. Detailed results can be found in Felner's thesis [7].

5.2 Original Path Created with the 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, and that they 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 when using the

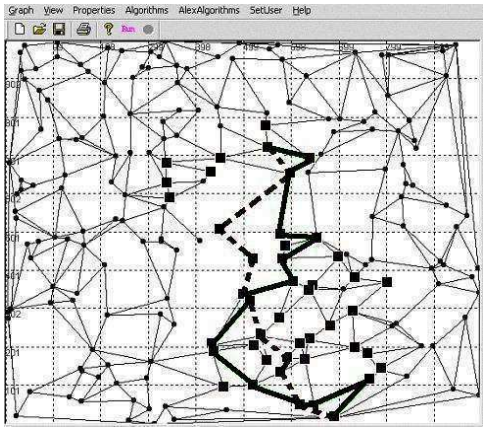

Fig. 11. Alternative and original paths

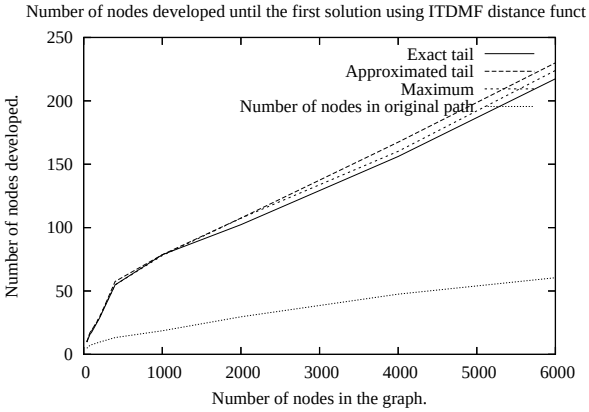

Fig. 12. Number of generated nodes, ITDMF

Best path	Number of runs
1	119
2	107
3	46
≥ 4	28

Table 1. When the best path was found, in 300 runs; each row corresponds to a different case (that is, row 1 signifies that in 119 cases the best path was the first path reached).

prefix heuristic with the ITDMF distance metric.[5] Figure 12 presents the number of nodes generated by each of the heuristics until the first alternative path was found. All datapoints are the average of 300 random instances with the same characteristics. The figure also presents a curve with the average number of nodes of the original path. It shows that all three heuristics behave approximately 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 supports the contention that our algorithm is quite efficient and finds the alternative path rather quickly. The results of the three approaches are not significantly different in this set of experiments because the original path was generated as the shortest path between the initial and goal node. Therefore, it tends to be a path that is relatively straight, and the heuristics measure similar values. These approaches performed differently in the set of experiments below where the original path was created randomly.

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, as illustrated in Table 1. Furthermore, 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 that 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 quickly, and both generated approximately four times as many nodes as the size of 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

[5] We used straight line distances for measuring distance between two states.

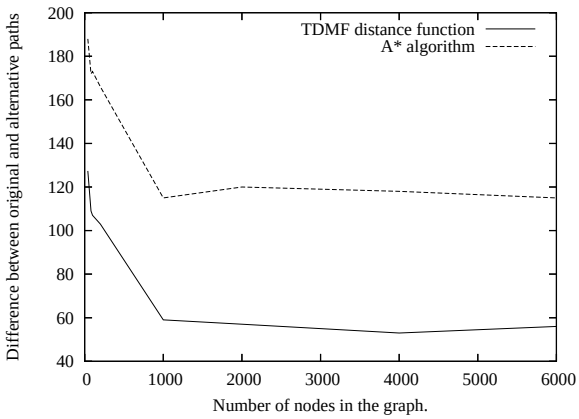

Fig. 13. ITDMF versus A*

by running the A* algorithm, even though the original path was created with the A* algorithm.

5.3 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.

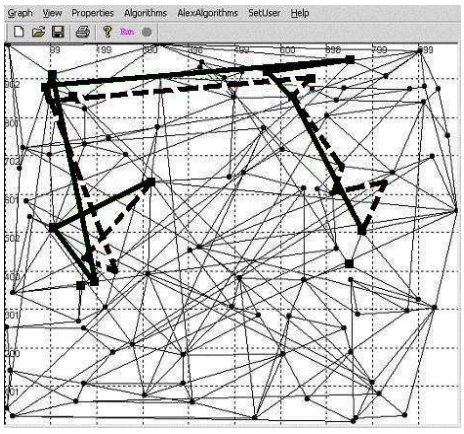

Fig. 14. Randomly created original path

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 intuitively clear.

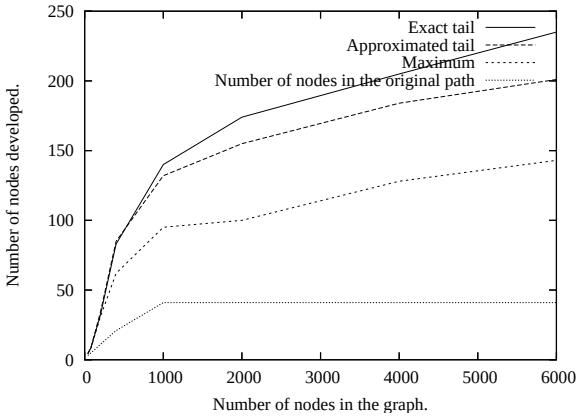

Fig. 15. Number of generated nodes, ITDMF

Figure 15 presents the number of nodes generated by each of our approaches (for measuring the estimated length of the alternative path for ITDMF) 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 speed between the three approaches of reaching the goal is much larger than in the previous case (where the shortest path was used for the original path); it seems that the combined version, which takes the maximum, produced the best results and was the fastest. This is due to the nature of the path which was randomly created and could go in different directions. 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 quickly by all three approaches.

The experimental results support the applicability and effectiveness of our search algorithm and the various heuristics. It also illustrates the differences between the three heuristics. They perform similarly for the set of experiments where the shortest path was used as the original path, but perform differently when the path was randomly generated. The combined heuristic — where the maximum between the previous two heuristics is used — performed best because it incorporates more information.

6 Example in the Blocks World

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

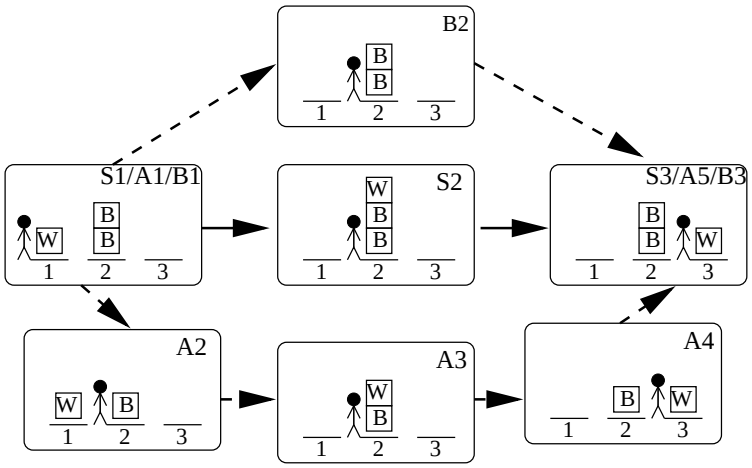

Fig. 16. A modified Blocks World example

Consider the blocks world in Figure 16 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.

The initial state is at the left-most frame in the middle row, and the goal is to move the white box (marked by W) to the third slot, as can be seen in right-most square of that row (which models a goal state). There are three different plans in Figure 16 that all reach a goal state. The original plan S is illustrated in the middle of the figure while two alternative plans A and B are illustrated below and above S respectively. In the original plan S, the robot takes the white block and puts it on top of the black blocks in slot 2, and then moves it again to slot 3; this is done by performing two $carry$ operators. Suppose that there is a restriction on the number of blocks in a given slot and that a stack of three blocks is not allowed. Both alternative plans A and B must 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 IDMMF function and measure the distance between these two plans to the original plan. The IDMMF function assumes that there exists a distance function between any two states in the domain. While this is trivial for a physical graph, we should first define a distance metric between the states of the blocks world. There

are several sensible ways to define such a distance metric in this domain. We introduce two metrics that were also proposed in the work of Ephrati et al. [6].

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)) = $3 \times |N - M|$
Diff(At(S),At(T)) = $|S - T|$
Diff(On(A,B,S),On(A,C,T)) = $2 \times |S - T|$

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 $Carry(A, S, T) = 2 \times |S - T|$. Of course, in some circumstances finding this minimal plan might be costly and another distance function should be used.

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. This is mainly for purposes of illustration, and in more realistic domains other distance functions could be explored.

6.1 Using IDMMF in the Blocks World

Here we describe the steps of our algorithm on the example of Figure 16. 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 partial plan that corresponds to its branch in the tree. 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 next, 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 next.

We now calculate the heuristic value 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
$ph(\{a_1, a_2\}, S) = \frac{F_{map}(\{a_1,a_2\} \rightarrow S_1)}{2} + \frac{F_{map}(S_1 \rightarrow \{a_1,a_2\})}{1} = 2.$

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.

7 Searching for an Alternative Schedule

The basic idea considered above, of applying best-first search to find alternative plans, is also relevant to problem domains that are not purely state-space domains. To further explore the generality of our approach, we expand it to the problem of rescheduling. In this domain, a given schedule of tasks (i.e., the internal order in which they should be executed) cannot be followed as planned, and thus an alternative schedule is required. There are many real world examples, such as flight or delivery scheduling.

Consider the flight scheduling problem. Flights are often delayed or canceled due to mechanical failures or weather. When this happens, the original flight schedule cannot be executed as planned and a new schedule is required. One of the important properties of the new schedule might be that it should require "a small number" of changes from the original schedule, so as to minimize the discomfort of passengers. Therefore, we might want to find a plan — a schedule — that is "as close as possible" to the original plan.

Scheduling algorithms attempt to create good schedules under a set of constraints. The field of scheduling algorithms is both broad and complex; there exist many surveys, reviews, and handbooks of scheduling problems and algorithms (e.g., [12, 28, 14, 17]). In this section, we do not present a general scheduling algorithm, but rather an algorithm for finding a schedule that is as close as possible to a given original schedule. We show how our approach to finding an alternative solution (discussed above), is also applicable to scheduling. We propose distance metrics, and employ a best-first search algorithm similar to the search described in previous sections.

7.1 The Scheduling Model

There are many classes of scheduling problems, and many associated ways to model them. However, since our focus is not on scheduling per se, but rather on the search for an alternative plan (or schedule), we choose a simple and well-known domain-independent scheduling problem and its associated model. Let $j_1, j_2, \ldots, j_n$ be a set of homogeneous jobs (i.e., jobs requiring exactly the same amount of time to execute), that can be executed sequentially. A schedule S defines the order of job execution, such that $S(j_1)$ designates the position in the sequence of execution. For example, if the schedule defines performing j_1 first, then j_2 followed by j_3, $S(j_1)$ would be 1, with $S(j_2) = 2$ and $S(j_3) = 3$. Therefore, a schedule can be viewed as a permutation of the job indices. For example for $n = 3$, a schedule $S =< 3, 1, 2 >$ states that job j_3 is executed first, then j_1, and then j_2.

7.2 Distance between Schedules

Our first need is to define distance metrics between two schedules. There are many considerations that apply to schedule comparison, depending on the specific problem.[6] We have used here two simple and intuitive approaches, and formalize them into distance metrics that will later be used in the search process.

There are many research fields in which distances between permutations are considered. For example, in the sorting community distance metrics between permutations are used to measure the amount of "presortedness" of an unsorted sequence, as can be seen in the work done by Mannila [20]. A comprehensive discussion of various distance metrics between permutation-type representations (such as the scheduling model mentioned here) is provided in the work of Sorensen [29, 30].

We now turn to defining metrics between schedules for the purposes of our research. These metrics can be used along with other metrics.

Mismatch and Index-based Metrics In *mismatch metrics*, two schedules are considered closer to one another if they contain many jobs that are performed, in both schedules, in the same time slot. A simple distance metric counts the number of jobs that are not executed in the same time slot in both schedules. More formally, we define $eq(a, b) = 0$ if $a = b$, and $eq(a, b) = 1$, otherwise. We then define the mismatch metric as:

$$\text{mismatch}(S_1, S_2) = \sum_{i=1}^{n} eq(S_1(i), S_2(i))$$

where n is the number of jobs.

However, this approach to counting the number of misplaced jobs only measures how many jobs were given different time slots in the compared schedules. The mismatch metric does not consider how many time slots each job was moved (i.e., the distance between a job's position in the two schedules). For example, consider schedules $S_{org} =< 1, 2, 3, 4, 5, 6 >$, $S_1 =< 6, 2, 3, 4, 5, 1 >$, and $S_2 =< 1, 2, 3, 4, 6, 5 >$. According to the mismatch distance metric, schedules S_1 and S_2 have the same distance to S_{org} — that is, mismatch(S_1, S_{org}) = mismatch$(S_2, S_{org}) = 2$. However, the two misplaced jobs in S_1 (jobs 1 and 6) were moved by five time slots from the original schedule, while in S_2 the change was of only a single time slot (for each misplaced job). To address this issue, we generalize the above metric to the *index-based* metric:

$$\text{indexBased}(S_1, S_2) = \sum_{i=1}^{n} |S_1(i) - S_2(i)|.$$

Thus, for the example above, indexBased$(S_1, S_{org}) = 5 + 0 + 0 + 0 + 0 + 5 = 10$, while indexBased$(S_2, S_{org}) = 0 + 0 + 0 + 0 + 1 + 1 = 2$.

[6] Of course, there may be many domain-specific considerations that make one schedule "closer" to another. We are exploring domain-independent aspects of the problem.

Sequence metric Another distance metric of schedules might take as its main criterion the comparison of internal sequences in the schedule. According to this notion, a schedule is considered close to another schedule if they contain a large identical sequence of jobs. Formally, we define the *longest sequence* function as:

$$\text{longestSequence}(S_1, S_2) = n - \textit{MaxSequence},$$

where

$$\textit{MaxSequence} = \max(Q) \text{ s.t. } \exists t\ 1 \leq t \leq n,\ \forall i\ t \leq i < (t+Q) \mid S_1(i) = S_2(i).$$

MaxSequence is therefore the maximal internal sequence of jobs that appears in both schedules.

Consider the distance between the following schedules, $S_{org} =< 1, 2, 3, 4, 5, 6, 7 >$, and $S_1 =< 2, 3, 4, 5, 6, 7, 1 >$, computed by the different distance metrics described above. Then:

- mismatch$(S_1, S_{org}) = 7$;
- indexBased$(S_1, S_{org}) = 6 + 1 + 1 + 1 + 1 + 1 + 1 = 12$;
- longestSequence$(S_1, S_{org}) = 7 - 6 = 1$.

Of course, one can suggest hybrids incorporating features of these metrics (and others), but the overall principles remain the same.

7.3 The Search Process

After having defined distance metrics, we now turn our attention to a search algorithm for finding a schedule close to the original. We can easily adopt the same search mechanism that was used above for state spaces, with only a few necessary modifications; here too, each node in the search tree will be a partial plan. In this case, of course, a partial plan is a schedule where not all the jobs have been assigned a time slot. Here we also require a heuristic that will evaluate the probability that the partial schedule will lead to a full schedule that will be close to the original schedule (according to the distance metric).

In domains we previously considered, the cost functions used by the search algorithm to evaluate nodes on the open list were not admissible. That is, we were not guaranteed that the current distance between the two partial paths is a lower bound on the final distance between the two complete paths. Therefore, when expanding the first possible goal node (the first node representing a full alternative plan), we did not necessarily have the alternative plan with the smallest possible distance to the original plan. In the scheduling domain, however, it is possible to have an admissible cost function, i.e., distances between a partial schedule and a full schedule will always be a lower bound on the distance between the two full schedules. Therefore, when using a best-first search mechanism to expand nodes, when the first goal node is chosen for expansion we are guaranteed to have the solution with the lowest cost — in our case, the closest schedule.

Creating admissible cost functions for the distance metrics described above can be done via problem abstraction by deleting constraints. This is achieved by computing

the distance metric between the partial schedule and the original schedule, while not considering any jobs that have not yet been scheduled (in the partial schedule). This is easy to calculate in all distance metrics we presented for the scheduling problem. For example, suppose that S_{org}=$< 1, 2, 3, 4, 5, 6, 7 >$ is the original schedule, and S_{par}=$< 3, 4, 2, 1, *, *, * >$ is the partial schedule for which the heuristic is needed. For the mismatch metric the heuristic distance is $h(S_{par}, S_{org}) = 4$ because all 4 jobs are mismatched. For the index-based metric it is $h(S_{par}, S_{org}) = 2+2+1+3+0+0+0 = 8$. Finally, for the sequence metric since the maximal sequence is of size 1 we get $h(S_{par}, S_{org}) = 3$.

7.4 Experimental Results for the Scheduling Environment

Since in the scheduling domain we use admissible heuristics, we have the best alternative schedule as soon as the first full alternative schedule is chosen for expansion. In this subsection we assess the performance of our search process empirically, by experimenting on a number of randomly created problem instances.

We experimented on what we refer to as *scheduling search trees*. Each node in this tree corresponds to a partial or full schedule (i.e., a partial or full permutation of the tasks). The root of this tree contains the empty schedule. Expanding a node means adding the next task to the end of the partial schedule. The leaves of this tree are complete schedules.

In real life, there are usually scheduling constraints among the various tasks assignments that define which partial and full schedules are allowed. Therefore, when expanding a node in the scheduling search tree we only add tasks that will not violate scheduling constraints.

To better simulate a wide variety of scheduling rules, we ran our experiments on randomly created *scheduling search trees* that were created as follows. We first determined two parameters, the *schedule size* (which is the number of tasks to be scheduled), and the *tree size* (which is the number of nodes in the tree).[7] We built the tree as follows: we randomly generated a schedule, then added it to the search tree. We continued this process as long as the number of nodes in the tree did not exceed the *tree size* parameter. Schedules that were not added to the tree are assumed to be schedules that violate the constraints.

We performed several experiments on random scheduling search trees with different schedule sizes and tree sizes. For each of these trees, we ran our best-first search algorithm with the various heuristics and distance metrics defined above.

Figure 17 shows the average number of nodes generated during the search with the various heuristics, as a function of the number of nodes in the tree. Figure 18 provides the ratio of generated nodes to the number of nodes in the graph, on the same set of experiments. Each data point in these figures is the average of 100 random scheduling search trees with the same characteristics.

The curves denoted “Indices 20” and “Sequence 20” refer to using the index-based and longest sequence metric, respectively, for schedules of size 20. “Indices 30” and “Sequence 30” correspond to the same metrics, but for schedules of size 30. The size of

[7] Note that schedule size is actually the depth of the tree.

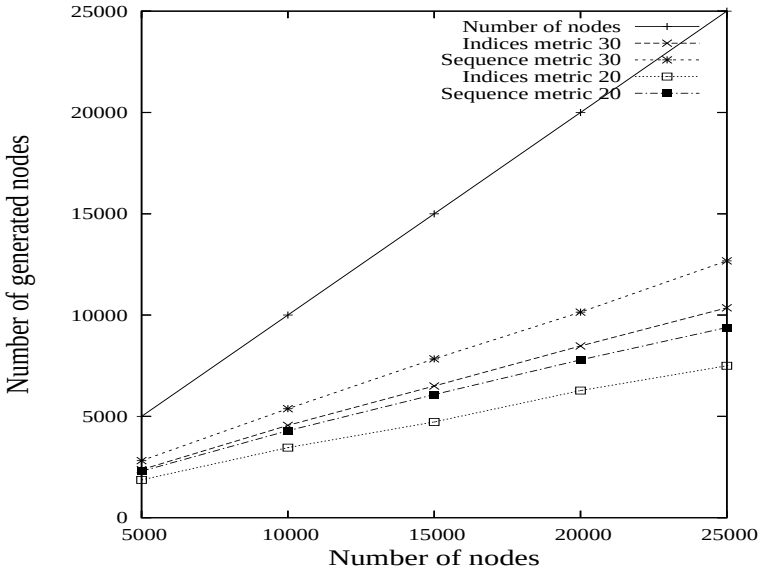

Fig. 17. Number of nodes generated during the search

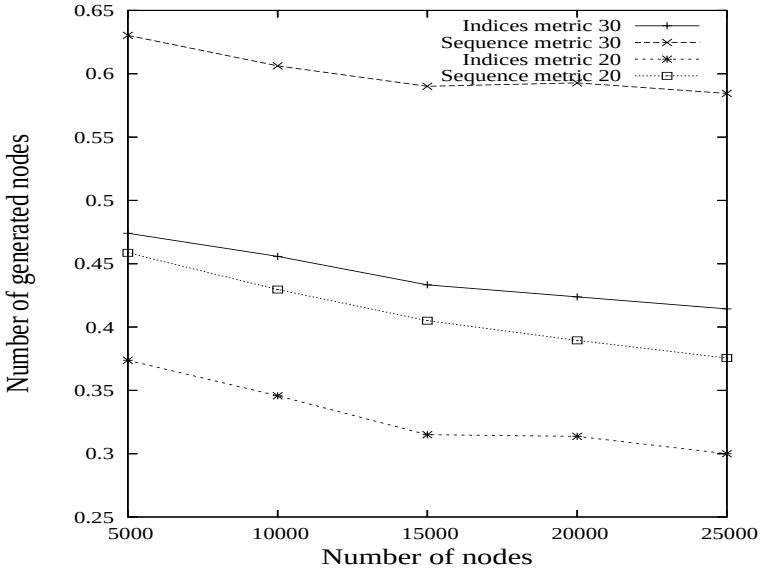

Fig. 18. Percentage of nodes that were generated during the search

the search tree was varied between 5,000 and 25,000. To better emphasize our results, we also added to Figure 17 another line, denoted "nodes," that displays the number of nodes in the tree (graph).

As would be expected, when the tree size is larger, we need to expand more nodes in order to find the best alternative schedule. However, for all metrics and graph sizes, the number of generated nodes is always significantly smaller than the actual number of nodes in the tree. This shows the benefit of our approach — we need to generate only a part of search tree to find the optimal alternative schedule. For example, search using the index-based metric in a scheduling search tree of size 15,000 required generating 6,499 nodes for a schedule of size 30, and only 4,726 nodes for schedule size 20.

Furthermore, as can be seen in Figure 18, the ratio of generated nodes decreases when the size of the graph increases. This is because as the number of nodes increases, the graph becomes denser, and thus enables the heuristic to prune more nodes during the search. This also suggests that for even larger graphs, the benefit of our algorithm would increase.

Note that even though the graphs used here were extremely sparse (a full tree for schedule size 20 will contain $20!$ nodes, which is much larger than the 25,000 nodes used here), the search was able to prune large portions of the graph — thus suggesting the efficiency of our method.

Another observation is the following. Figure 17 shows that when the search is performed to find the best alternative schedule according to the index-based metric, it generates a smaller number of nodes than when it searches for the longest sequence metric. This was valid for all sizes of schedules and schedule trees. The reason for this is that the index-based metric allowed a wider range of values than the largest sequence metric. In the largest sequence metric, the maximum value that a schedule can have is the schedule size minus 1 (which happens when the compared schedules have no identical sequences), while in the index-based metric, values can get much higher. A larger range of values suggests a larger variety of node costs during the search. This enables the heuristic to prune more nodes, thus explaining the improved performance of the index-based metric.

Notice also from Figure 18 that when comparing searches for different schedule sizes on a fixed tree size, we see that it is harder to find an alternative schedule for longer schedules. This is because, for a given tree size, a larger schedule size will produce a tree with a smaller branching factor. A smaller branching factor enables a smaller amount of pruning with admissible heuristics, and thus more nodes will be generated during the search.

8 Conclusions and Future Work

We have presented an algorithm that efficiently helps an agent find an alternative path that is close to an original path it has been given. We have demonstrated the use of this algorithm in three different environments: physical roadmaps, the blocks world, and scheduling. For the physical environment, we presented two complex metrics for the purpose of assessing path "closeness", and presented a heuristic function to guide the search. Experimental results confirm that such an alternative path is reached relatively

efficiently. For the scheduling environment, we have discussed two distance metrics that enable the use of admissible heuristics during the search process. With admissible heuristics, the "closest" schedule can be found efficiently, by pruning parts of the search space.

Future work can proceed in four different directions. The first direction is to further explore the heuristics and distance metrics that were used. New "closeness" distance metrics can be devised for various environments; for example, a hybrid metric for physical environments that is a combination of IDMMF and ITDMF may be a good metric for considering both time and distance considerations. All examples presented in this paper show a simple model with simple distance metrics. Creating more sophisticated distance metrics for more interesting models, and eventually creating a method for creating distance metrics, is in our view a subject worthy of future study. A general automated method for providing distances could also be a subject of future research.

The second direction is to focus on the search algorithm itself. In this paper, we presented a general (domain independent) search method for our problem. Devising domain-specific search algorithms may prove useful.

The third direction is to explore more intricate models of domains. In this work, we assumed that the domain can be modeled as a state graph, and that every node in the original plan is equally important. Future work can address different types of domains where these assumptions do not hold. For example, the supervisor that defined the original plan may be able to define which states in the original plan are not important, and thus the distance from them in the alternative plan may be ignored.

Finally, in this paper we dealt only with discrete domains. Future work could address the implementation of our various techniques in continuous domains.

9 Acknowledgments

The research was supported by the Israeli Ministry of Science, Infrastructure grant No. 3-942.

References

1. J. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *Proceedings of AAAI-88*, pages 735–740, St. Paul, Minnesota, 1988.
2. E. Arkin, L. P. Chew, D. P. Huttenlovher, K. Kedem, and J. S. B. Mitcjell. An efficiently computable metric for comparing polygonal shapes. In *Proceedings of the first ACM-SIAM Symposium on Discrete Algorithms*, pages 209–216, 1990.
3. M. J. Atallah. A linear time algorithm for the Hausdorff distance between convex polygons. *Information Processing Letters*, 17:207–209, 1983.
4. C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hull. Geometry Center Technical Report GCG53, University of Minnesota, 1993.
5. O. Brock and K. Oussama. Real-time replanning in high-dimensional configuration spaces using sets of homotopic paths. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 550–555, San Francisco, USA, 2000.
6. E. Ephrati and J. S. Rosenschein. Planning to please: Following another agent's intended plan. *Group Decision and Negotiation*, 2(3):219–235, 1993.

7. A. Felner. Searching for an alternative plan. Master's thesis, Department of Computer Science, The Hebrew University, Jerusalem, Israel, 1995.
8. A. Felner, A. Pomeransky, and J. S. Rosenschein. Searching for an alternative plan. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 33–40, Melbourne, Australia, 2003.
9. K. Z. Haigh and M. Veloso. Route planning by analogy. In *Proceedings of the International Conference on Case-Based Reasoning*, 1995.
10. 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*, SSC-4(2):100–107, 1968.
11. F. Hausdorff. *Grundzuege der Mengenlehre*. Viet, Leipzig, 1914.
12. A. Jain and S. Meeran. A state-of-the-art review of job-shop scheduling techniques, 1998.
13. 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.
14. D. Karger, C. Stein, and J. Wein. Scheduling algorithms. In M. J. Atallah, editor, *Handbook of Algorithms and Theory of Computation*. CRC Press, 1997.
15. S. Koenig and M. Likhachev. D* lite. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 476–483, Edmonton, Canada, July–August 2002.
16. S. Koenig and M. Likhachev. Incremental A*. In *Advances in Neural Information Processing Systems 14 (NIPS)*. MIT Press, Cambridge, MA, 2002.
17. J. Leung and J. H. Anderson. *Handbook of Scheduling*. CRC Press, May 1, 2004.
18. W. Li and M. Zhang. Distributed task plan: A model for designing autonomous mobile agents. In *Proceedings of the International Conference on Artificial Intelligence*, pages 336–342, Las-Vegas, 1999.
19. B. Liu. Intelligent route finding: Combining knowledge, cases and an efficient search algorithm. In *Proceedings of ECAI-96*, pages 380–384, Budapest, Hungary, 1996.
20. H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, 34(4):318–325, 1985.
21. R. B. McMaster. Measurement in generalization. In *Proceedings of the 20th International Cartography Conference*, pages 20–82, August 2001.
22. R.B. McMaster. A statistical analysis of mathematical measures for linear simplification. *The American Cartographer*, 13(2):103–117, 1986.
23. R.B. McMaster. The integration of simplification and smoothing algorithms in line generalization. *Cartographica*, 26:101–121, 1989.
24. K. L. Myers and Thomas J. Lee. Generating qualitatively different plans through metatheoretic biases. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 570–576, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.
25. B. Nebel and J. Koehler. Plan reuse versus plan generation. *Artificial Intelligence*, 76:427–454, 1995.
26. A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations, Concepts, and Applications of Voronoi Diagrams*. Wiley, Chichester, UK, 1992.
27. S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach, Second Edition*. Prentice Hall, 2005.
28. J. Sgall. *Line scheduling — a survey, On-Line Algorithms*. Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1997.
29. K. Sorensen. Distance measures based on the edit distance for permutation type representations. In *Proceedings of the Workshop on Analysis and Design of Representations and Operators*, pages 29–35, Chicago, USA, 2003.

30. K. Sorensen, M. Reimann, and C. Prins. Permutation distance measures for memetic algorithms with population management. In *Proceedings of MIC 6th Metaheuristics International Conference*, Vienna, Austria, 2005.
31. A. Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of ICRA*, pages 3310–3317, 1994.