RC25463 (WAT1405-005) May 2, 2014
Computer Science

IBM Research Report

New Algorithms for the Top-*K* Planning Problem

Anton V. Riabov, Shirin Sohrabi, Octavian Udrea
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Research Division
Almaden – Austin – Beijing – Cambridge – Dublin - Haifa – India – Melbourne - T.J. Watson – Tokyo - Zurich

New Algorithms for The Top-K Planning Problem

Anton V. Riabov Shirin Sohrabi Octavian Udrea
IBM T.J. Watson Research Center
PO Box 704, Yorktown Heights, NY 10598, USA
{riabov, ssohrab, oudrea}@us.ibm.com

Abstract

Cost-optimal planning is a variant of a general planning problem, where all actions have non-negative costs, and the solution is a valid plan that minimizes the sum of the costs of all actions included in the plan. In this paper, we propose a new planning problem formulation, top-k planning, which is a generalization of cost-optimal planning with applications in plan recognition, diagnosis, explanation generation, and other domains. No existing planners can solve this problem out of the box. We have implemented and compared a total of four new planning algorithms for top-k planning. Two of the algorithms are based on the k shortest paths algorithm by Eppstein and a recently proposed variant of that algorithm for dynamic graphs called K^*, by Aljazzar and Leue. We also implemented a branch and bound algorithm, and an iterative replanning algorithm based on LAMA. Our experiments show that the top-k planning problem can be solved efficiently, in time comparable to cost-optimal planning. We also show that our implementation of top-k planning based on the K^* algorithm outperforms other algorithms.

Introduction

The shortest path problem is a problem of finding a path connecting a given source-destination pair in a graph with minimum total cost (or length). The all-pair shortest path requires computation of the shortest path between all pairs in the graph. Several researches have examined modeling the planning problem as a shortest path computation in a graph or more specifically applying the single-source or all-pair shortest paths algorithm to precompute all shortest paths (e.g., (Botea and Harabor 2013; Edelkamp and Kissmann 2009)). This eliminates the need to search and can lead to fast computation of cost-optimal plans. In turn, cost-optimal planning can also be used for solving a class of preference-based planning problems (Baier and McIlraith 2008) following Keyder and Geffner 2009.

Given an arbitrary number k, the k shortest paths problem is a problem of finding the k shortest paths from a source node to a destination node in a graph. This problem has many applications including path planning (e.g., (Zhu et al. 2013)), video games (Botea 2011), and networking. There are a number of reasons why a k shortest paths algorithm could be needed instead of a single shortest path. Computing k shortest paths can be beneficial, for example, if there are other types of constraints beyond path costs, but those constraints are not fully defined or are missing. In addition, analyzing the k shortest paths can help gain better understanding of the properties of the problem and its optimal solutions. See (Eppstein 1998) for a more comprehensive discussion of these applications.

In this paper, similar to how the k shortest paths extends the shortest path problem, we propose the formulation of the top-k planning problem for cost-optimal planning. We also propose four planning algorithms for top-k planning based on existing methods, including k shortest paths computation for plan cost minimization over the state graph. Unlike optimal planning, where the objective is to find one optimal plan with minimum cost, we define the top-k planning problem as the problem of finding a set of k distinct plans with lowest cost. This set can contain both optimal plans and near-optimal plans, depending on k, and, by definition, for each plan in this set all valid plans of lower cost must also be in the set. To the best of our knowledge we are the first to formulate the top-k planning problem and propose a solution to finding top-k plans, at least for cost-optimal planning.

We argue that the top-k planning problem has important applications, some of which intersect with those of the k shortest paths problem. In particular, we are interested in this problem because of its applications in plan recognition, diagnosis of discrete event systems, and explanation generation, all of which can be modeled as planning problems (e.g., (Ramírez and Geffner 2009; Sohrabi, Baier, and McIlraith 2011)). In these applications it may be important to not only generate one optimal solution, but a set of "good" alternatives. In recent work, the need for top-k planning has been highlighted in the malware detection problem, where the objective is to explain the sequence of observations given the system description (Sohrabi, Udrea, and Riabov 2013). There, top-k plans correspond to alternative plausible hypotheses explaining unreliable observations. Generally, computing top-k plans can help deal with incompleteness in the domain, imperfect quality measures, and unreliable knowledge, such as missing or noisy observations. Hence, we believe top-k planning formulations and algorithms can provide some of the tools needed for fulfilling the model-lite planning vision (Kambhampati 2007).

The k shortest paths problem was introduced in (Hoffman and Pavley 1959) and several efficient algorithms were developed for it. In particular Yen's algorithm (Yen 1971) and

several later implementation improvements of it are used to find ranked loopless paths. Another known algorithm is by Eppstein 1998, which allows loops and has better performance. However, one drawback of Eppstein's algorithm is that it requires the graph to be fully defined and available in memory. The recent extension of the Eppstein's algorithm called the K^* algorithm (Aljazzar and Leue 2011) overcomes this problem by supporting on-the-fly construction of the graph and thus allows use of heuristic search, making it a very strong candidate match for planning problems.

In this paper, we introduce and compare four planning algorithms for top-k planning: iterative replanning using LAMA or other existing high-performance planners, branch and bound, a planning algorithm based on the Eppstein's k shortest paths algorithm, and an algorithm based on the K^* algorithm. We call the top-k planner based on Eppstein's algorithm, *TK*, and the planner based on the K^* algorithm, *TK**. Note, that our algorithms are based on known existing methods, but are employed to address the top-k planning problem. Our experiments show that planning time required for top-k planning is comparable to cost-optimal planning that finds a single cost-optimal plan. We also find that *TK** outperforms all other approaches by a large margin.

Problem Formulation

Top-k planning problem is defined as $R' = (F, A, I, \mathcal{G}, k)$, where F is a finite set of fluent symbols, A is a set of actions with non-negative costs, I is a clause over F defining the initial state, $\mathcal{G}$ is a clause over F defining the goal state, and k is the number of plans to find. The set of plans $\pi = \{\alpha_1, ..., \alpha_k\}$ is the solution to the top-k planning problem R' if an only if each plan $\alpha_i \in \pi$ is a solution to the cost-optimal planning problem $(F, A, I, \mathcal{G})$ and there does not exists a plan α' for $(F, A, I, \mathcal{G})$, $\alpha' \notin \pi$ such that $cost(\alpha') < cost(\alpha_i)$ for all $\alpha_i \in \pi$. It follows that at least one optimal plan is in the set of plans π if $k > 0$.

Note, while we indicated that the goal state, $\mathcal{G}$, is in a form of a final-state goal in the definition of R', we consider temporally extended goals as well. Temporally extended goals such as sequence of observations from a system description either totally ordered or partially ordered can be compiled away to final-state goal following a compilation technique discussed in several papers (e.g., (Sohrabi, Baier, and McIlraith 2010; Haslum and Grastien 2011)); the temporally extended goals can be compiled away by an action which enforces the temporal sequence of the goal.

Top-k Planning via Iterative Replanning

The first of the four approaches we describe builds upon existing Planning Domain Definition Language (PDDL) (McDermott 1998) planners, extending the applicability of those planners to top-k planning problems. We have introduced this approach in prior work as a simple practical solution for top-k problems (Sohrabi, Udrea, and Riabov 2013). In our experiments we used LAMA (Richter and Westphal 2010), as one of the fastest planners available, but the approach does not depend on the choice of the planner. LAMA can be used to find cost-optimal plans by modeling costs as domain variables, and the last returned plan is optimal if LAMA is given sufficient time to complete the search and exit (this was confirmed in our experiments).

The main idea is to use a PDDL planner iteratively, slightly modifying the problem each time, until top-k plans are found. To solve the top-k planning problem $R' = (F, A, I, \mathcal{G}, k)$, we solve a sequence of cost-optimal planning problems $R_i = (F_i, A_i, I_i, \mathcal{G}_i)$, starting with finding the optimal plan of length n for $R_1 = (F, A, I, \mathcal{G})$. Then, given a cost-optimal plan p for R_i, and assuming m problems were created at a previous iteration, we create a new set of problems $\{R_j | j = m+1, ..., m+n)\}$ by modifying R_i by adding, for each action that occurs in p, a new precondition that prevents that action from appearing at the same position in the new plan. The best solution to the new problems R_j will be used to find the next-best plan p'. The generated problems are modified again to generate new problems and find the next best plan, until k such plans are found. Note that this algorithm does not find plans that have top-k plans as their prefixes (i.e., plans that reach the goal more than once). Overall, solving the top-k planning problem requires at most $O(N^k)$ replanning iterations, where N is the length of the longest plan among the top-k plans.

The actions are modified by introducing new predicates *(at-pos ?i)* and *(next ?i ?j)* to keep track of the position of each action in the plan. For example, the initial state will include predicates *(at-pos p1) (next p1 p2), (next p2 p3)*, etc., and for each action the precondition will include *(at-pos ?i) (next ?i ?j)*, while the effect of the same action will include *(at-pos ?j) (not (at-pos ?i))*. This modification does not change the set of valid plans. However, it allows disabling application of an action by adding a negated precondition: for example, adding *(not (at-pos p3))* to an action will prevent that action from appearing at the third position in any valid plan.

The outline of this algorithm is presented below.

```
0. Find plan α for the original problem R.
1. Set π = {α}.
2. Add each action a of α and its position i
    S = {(a, i)} to future exploration list L.
3. For each S in L
4.    For each (a, i) ∈ S
5.         Add negated predicate associated
           with a, i to action a.
6.    Generate a plan α' for the new problem
      where all actions in S are disallowed.
7.    For each action a at position i in α'
8.         Add the set S ∪ {(a, i)} to L'.
9. Add one of the plans α' with minimum cost to π.
10. Replace L with L'.
11. If |π| < k and L ≠ ∅ go to step 2.
12. Return π as the solution to the problem.
```

Top-k Planning via Branch and Bound

The second approach we propose for top-k planning problem is based on branch and bound. Unlike iterative replanning, it does not require solving many similar planning problems. Branch and bound is a general framework used for finding optimal solutions in a variety of settings, and it can be modified for solving top-k problems. The additional advantage of the general framework is the flexibility it allows in defining optimization objectives and constraints.

Branch and bound begins by selecting a variable for branching (e.g., the variable can represent the action applied at the first step of the plan), creates a search node for every feasible value of the variable, and computes lower and upper bounds on the possible solutions. For example, the lower bound can be the cost of the selected action and the upper bound can be infinity. Assuming that a minimization problem is being solved, search nodes can be pruned if their lower bound value exceeds already known upper bound on the solution, or the value of the current best solution. In the next iteration, which can be done separately for each search node, the next variable is selected, and the next set of search nodes is created following the same procedure.

This standard framework can be modified to find top-k solutions instead of a single optimal solution, by pruning the search tree based on the k-th best solution found instead of using the value of the best solution. When search terminates, the remaining k best solutions will be the top k solutions. In most scenarios, however, this modification increases the search time, especially for large values of k, because the search tree cannot be efficiently pruned until k complete solutions are found.

In our implementation we apply forward branching on operators of the planning problem after grounding. Grounding, implemented during preprocessing, assigns fixed values to variables of actions and thus creates multiple operators from a single action. During this preprocessing, an index of operators is also created, based on the matching between preconditions and effects. To control the potential exponential explosion of the number operators, we implement grounding based on forward reachability within a relaxed planning problem without deletes. We note that despite the reduction in the number operators thanks to reachability analysis, a significant fraction of operators created during grounding may still never be explored during search, since action costs are not accounted for in this process. The fourth approach we describe further in this paper will improve on this by implementing dynamic grounding.

The final algorithm can be summarized as follows.

```
0. Read planning problem R' = (F, A, I, G, k);
   Set U = {}, π = {}; Set UB = ∞;
   Insert a partial plan α = {I} into U.
1. Apply forward grounding to A
   to create operator set O.
2. If U is empty, return the set of plans π.
3. Remove a partial plan α from U.
4. If for last state s of α, s ∈ G Then
5.   Set π = π ∪ {s}.
6.   If |π| > k Then
7.      Set π = π \ arg max{cost(α') | α' ∈ π}.
8.      If |π| = k Then
9.         Set UB = max{cost(α') | α' ∈ π}.
10. Find operators {o} ⊂ O applicable in s;
    Set U = U ∪ {α' = (α, o) | cost(α') < UB}.
11. Repeat from step 2.
```

Note the algorithm assumes that all actions have nonnegative costs, and therefore costs of partial plans can be used as lower bounds for complete plans in Step 10. In addition, a variety of heuristics can be used to order the set of partial plans U in Step 3, and the choice of the heuristic will affect performance. In our implementation we sort partial plans by distance to goal computed based on the relaxed formulation used during grounding. Finally, while we have not done this ourselves, branch and bound and heuristic search share multiple features, and modifications for finding top-k plans can similarly be made to heuristic search algorithms.

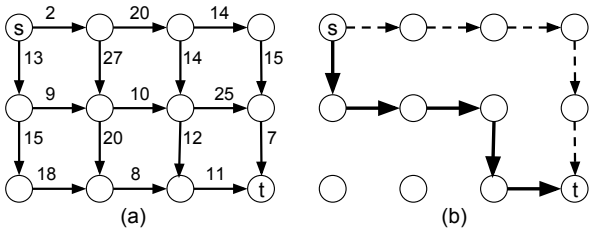

Figure 1: (a) shows the nodes and edges of a graph with source node s and terminal node t with edge lengths specified on the edges; (b) shows the shortest path in bold arrows and the second shortest path in dashed arrows.

Top-k Planning via K Shortest Paths

The cost-optimal planning problem can be modeled as the problem of finding the shortest path in state space from initial state to the goal. In this section we build on this idea by applying Eppstein's k shortest paths algorithm (Eppstein 1998) in state space to solve the top-k planning problem. The resulting algorithm is very efficient, but requires the complete graph of states and actions to be available in memory. Constructing this graph is expensive in large problems, and this shortcoming will be addressed using an improved variant of the algorithm in the approach described in the next section. In this section we first introduce notation for the k shortest paths problem, and then describe the planning algorithm based on Eppstein's k shortest paths.

Background: K Shortest Paths Problem

K shortest paths problem is an extension of the shortest path problem where in addition of finding one shortest path, we need to find a set of paths that represent the k shortest paths (Hoffman and Pavley 1959). Following Eppstein 1998, k shortest path problem is defined as 4-tuple $R = (G, s, t, k)$, where $G = (V, E)$ is a graph with a finite set of n nodes (or vertices) V and a finite set of m edges E, s is the source node, t is the destination node, and k is the number of shortest paths to find. Each edge $e \in E$ has a *length* (or *weight* or *cost*), which we denote by $l(e)$. The length of a path p, $l(p)$, is consequently defined by the sum of its edge lengths. The *distance* $d(u, v)$ for any pair of nodes u and $v \in V$ is the length of the shortest path between the two nodes. Hence, $d(s, t)$ is the length of the shortest path for the problem R. Figure 1 shows an example from (Eppstein 1998) to illustrate the terminology. The distance $d(s, t) = 55$, is the length of the shortest path shown in bold; the length of the second shortest path is 58.

The set of paths $P = \{p_1, p_2, ..., p_k\}$ is the solution to the k shortest paths problem R if and only if it is a set of shortest paths from node s to node t. That is each $p_i \in P$, $1 \leq i \leq k$, is a path in graph G and there does not exists a path p' in graph G, $p' \notin P$ such that $l(p') < l(p_i)$ for all $p_i \in P$. That is, there is no path, except amongst the k

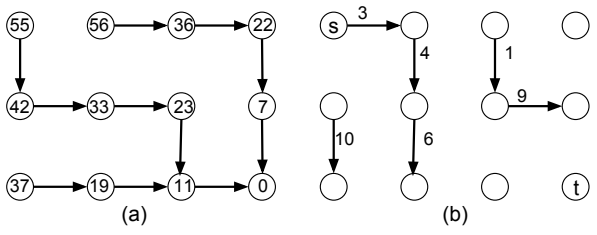

Figure 2: (a) shows the shortest path tree T and distance to destination t; (b) shows the side edges with their associated detour cost.

shortest paths, with better length than any of the paths in the set P. It follows that at least one shortest path with length $d(s,t)$ is in the set P if $k > 0$.

Background: Eppstein's Algorithm (EA)

Given a k shortest paths problem $R = (G, s, t, k)$, the EA algorithm first computes a single-destination shortest path tree with t as the destination (or the reversed single-source shortest path tree) by applying Dijkstra's algorithm on G. The edges in the resulting *shortest path* tree, T are called the *tree* edges while all the missing edges (i.e., the edges in $G - T$) are called the *sidetrack* edges. Each edge in G is assigned a number that measure the detour cost of taking that edge. Consequently, the detour cost of the tree edges is 0, while the detour cost of the sidetrack edges is greater than 0. Figure 2 shows the shortest path tree T and the side edges along with their detour cost of our earlier example.

The EA algorithm then constructs a complex data structure called *path graph* $P(G)$ that stores the all paths in G, where each node in represents a sidetrack edge. This is followed by the use of Dijkstra search to $P(G)$ to extract the k shortest paths. An important property is that given a sequence of sidetrack edges representing a path in $P(G)$ and the shortest path tree T, it is possible to uniquely construct a s-t path in G. This can be done by using sub-paths from T to connect the endpoints of sidetrack edges. Given this property and the special structure of $P(G)$, it is ensured that the i-th shortest path in $P(G)$ results in a sidetrack sequence which can be mapped to the i-th shortest path in G. By construction, $P(G)$ provides a heap-ordered enumeration of all paths in G, and since every node of $P(G)$ has limited outdegree (at most 4), the complexity of enumerating paths in increasing cost order is bounded. The worst-case runtime complexity of the EA algorithm is $O(m + n \log n + kn)$. This complexity bound depends on a compact representation of the resulting k paths, and can be exceeded if the paths are written explicitly, by enumerating all nodes and links, as we have done in our planner implementation. For more details see (Eppstein 1998).

Top-k Planning Algorithm Based on EA

Our planning algorithm can be summarized as follows. We call the top-k planner based on this algorithm, *TK*.

```
0. Read planning problem R' = (F, A, I, G, k).
1. Apply forward grounding to A
   to create operator set O.
2. Initialize G=(V,E): let V={I}, E=∅.
3. Let U={I}.
4. For each state s ∈ U
5.   U = U − {s}
6.   For each operator o ∈ O
     such that s satisfies precondition of o
7.       Let s' = o(s).
8.       If edge o(s,s') ∉ E Then
9.           If s' ∉ V Then
10.              Let V = V ∪ {s'}, U = U ∪ {s'}.
11.          Add o(s,s') to E.
12.          Let cost(o(s,s')) = cost(o).
13. If U ≠ ∅ goto step 4.
14. Apply EA to G to find k shortest paths.
```

This algorithm consists of three main stages. Step 1 implements action grounding. Steps 2-12 implement forward search to construct the complete state transition graph G. Finally, step 13 applies Eppstein's algorithm to the resulting graph. Since nodes in G represent states and edges in G correspond to operators, all paths in G correspond to plans in R', and paths have the same cost as corresponding plans. Therefore, the solution produced by Eppstein's algorithm can be directly used as a solution to the top-k planning problem. We note that in our experiments the first two stages, grounding and creating the state graph, taken together, took approximately the same amount of time as the last stage.

Top-k Planning via K^* Search

The major bottleneck of the previous approach is the construction of the complete state transition graph, which may include a huge number of states that are very far away from the goal, and would not appear in top-k plans. Planners commonly deal with this challenge by relying on heuristic search algorithms like A* to dynamically expand only the necessary portion of the state graph during search, while being guided by a heuristic toward the goal (e.g., FF (Hoffmann and Nebel 2001) and Fast Downward (Helmert 2006)), and the effectiveness of this approach has been proven (Bonet and Geffner 2001). The $K*$ algorithm proposed by Aljazzar and Leue combines the best of both worlds: it allows constructing the graph G dynamically using heuristic-guided A* search, while updating its equivalent of $P(G)$ to find k shortest paths. In addition to eliminating the complete state graph construction, with K^* we can ground actions dynamically, eliminating the expensive grounding stage.

Background: K^* Algorithm

The K^* algorithm (Aljazzar and Leue 2011) uses many of the same concepts as in the EA algorithm including sidetrack edges, detour costs, and the path graph $P(G)$ (although with a few differences in its construction) and has the same worst-case complexity as the EA algorithm. However the K^* algorithm has better performance in practice because unlike the EA algorithm it does not require the graph G to be completely defined or available when the search starts. It also does not perform the all-nodes shortest path computation on G to compute the shortest path tree T. In short, the K^* algorithm works as follows. The first step is to apply a forward A* search to construct a portion of graph G. The second step is suspending A* search, updating $P(G)$ to include nodes and sidetracks discovered by A*, and applying Dijkstra to $P(G)$ to extract solution paths and resuming the A* search. The use of A* search to dynamically expand G enables the

use of heuristic search and also allows extraction of the solution paths before G is fully explored.

Top-k Planning Algorithm Based on K^*

In the implementation of the planning algorithm we follow the algorithm structure imposed by K^*, as follows. Note that we call our top-k planner that is based on K^*, ***TK******.

```
0. Read planning problem R' = (F, A, I, G, k).
1. Expand the state graph G by using A*
   and applying actions to compatible  states
   starting from I, and until G is reached.
2. Continue applying A* to expand G
   until 20% increase in links or nodes.
3. Update P(G) based on new links in G.
4. Apply Dijkstra step
   to extract the next path from P(G).
5. If k paths are found
6.      Exit.
7. If K* scheduling condition is reached
8.      Goto step 2.
9. Goto step 4.
```

The K^* scheduling condition is evaluated by comparing the state of A* and Dijkstra searches, as defined in K^* algorithm. It determines whether new links must be added to G before resuming Dijkstra search on updated $P(G)$. There is no separate grounding stage, since actions are ground at the same time when they are applied during A* search. The amount of A* expansion required before resuming Dijkstra (in our implementation, 20%), is an efficiency tradeoff, and 20% is the same value that was used in experiments in the original K^* paper (Aljazzar and Leue 2011). Of course, step 2 may also be completed if no new links can be added.

Overall, due to multiple improvements in efficiency made possible by this algorithm, ***TK****** was the best performing in our experiments. We also expect that with some work this approach can be integrated into planners that use A* search, enabling those planners to solve top-k problems.

In our experiments, ***TK****** with constant 0 heuristic performs very well, and we have not experimented with other, potentially better performing heuristics. This is an interesting direction for improvement that could be explored in future work. Even though this is not a requirement for K^* in general, our implementation requires a consistent heuristic, which did not allow us to experiment with, for example, lookahead heuristics. Further, the dynamic grounding prevented the use of heuristics used in the Branch and Bound approach, since those heuristics require static grounding.

Experimental Evaluation

In this paper we argue that practical solutions can be developed for the top-k planning problem. To that end, we have 3 main objectives in our experiments. First, we measure the change in performance that results from the requirement to find top-k plans instead of a single cost-optimal plan. Second, we compare the performance of four different approaches we propose. Third, we measure the effect that increasing the value of k will have on planning time.

Generated Random Problem Instances

The approach we introduced in this paper is general and can be applied in a variety of applications that require cost-optimal planning. As a benchmark for performance evaluation, we have generated random instances of varying size based on the hypothesis exploration problem with unreliable observations (Sohrabi, Udrea, and Riabov 2013). This application provides a good example of a challenging top-k planning problem, and generated problems typically have a very large number of possible plans with different costs. The domain and the generated problems were represented in a STRIPS-like planning language recognized by our planner, as well as in PDDL for LAMA.

All generated problems share a planning domain description containing 6 actions and 8 predicates. In this domain, low costs were assigned to actions used in perfect explanations of observations, and high costs to actions representing exceptions, such as unexplained observations or state transitions without observations. To generate a random problem instance, we generated a random state transition system with a given number of states.

Malware Detection Instances

In addition to randomly generated state transition systems, we used the malware detection problem (18 states), as described in (Sohrabi, Udrea, and Riabov 2013). In short, the malware detection problem involves generating hypotheses about the network hosts by analyzing the network traffic data. To make this possible, the domain description includes the states of the host (e.g., infected with malware due to downloading an executable file or the *Command & Control Rendezvous* state via Internet Relay Chat (IRC)) and transitions between these states and many-to-many correspondence between states and observations. The results for this domain is shown under the "Malware Domain" rows.

Planning Time for The Top-k Problem

We have varied the size of the problem by changing the number of the states of the system being modeled (not to be confused with planning states) and the number of observations received from the system. For all time measurements in this paper we used the same Quad-core 2.93 GHz Intel Xeon X5570 processor with 32 GB RAM and 64-bit Red-Hat Linux OS.

Table 1 presents the results of comparison between approaches described in this paper. For all algorithms except iterative replanning, we measured time it took to find top $k = 50$ plans. For iterative replanning ("LAMA top-1" column in the table), we measured the duration of a single iteration, i.e., solving one cost-optimal planning problem using LAMA, while at least $k = 50$ iterations will be required for the top-k problem (and in the worst case, exponentially more). Conveniently, this also helps compare planning time of top-k and regular cost-optimal planning. During measurement we enforced a limit on planning time of 300 seconds, and the instances where this limit was exceeded are indicated by "-" in the table. For LAMA the time we report is the time that it took for LAMA to terminate (i.e., exhaust the search space) and hence its last returned plan is cost-optimal or top-1. The "-" entries indicate that LAMA was not able to find the cost-optimal plan or terminate its search before the time limit is reached.

	LAMA, top-1			Branch & Bound, top-50			*TK*, top-50			*TK* *, top-50		
Problem size	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Malware Domain, 5 obs.	0.52	0.64	1.16	0.11	0.22	0.39	0.06	0.07	0.09	0.04	0.06	0.08
10 states, 5 obs.	0.35	0.52	1.68	0.08	0.13	0.22	0.05	0.05	0.06	0.03	0.04	0.06
50 states, 5 obs.	0.94	1.08	1.17	0.36	0.56	0.97	0.18	0.21	0.24	0.06	0.08	0.10
100 states, 5 obs.	2.64	2.95	3.20	1.45	2.49	4.19	1.03	1.23	1.47	0.18	0.22	0.25
Malware Domain, 10 obs.	0.69	0.75	0.86	0.28	3.67	10.33	0.09	0.12	0.15	0.06	0.07	0.11
10 states, 10 obs.	0.40	0.47	0.54	0.64	0.97	1.59	0.06	0.07	0.08	0.04	0.06	0.07
50 states, 10 obs.	1.50	1.89	2.37	1.46	7.86	37.04	0.42	0.51	0.60	0.10	0.12	0.13
100 states, 10 obs.	5.20	6.27	9.14	5.88	21.08	51.93	2.62	3.43	3.97	0.21	0.35	0.55
Malware Domain, 20 obs.	1.06	1.37	2.10	1.49	-	-	0.19	0.27	0.33	0.08	0.12	0.38
10 states, 20 obs.	0.50	0.71	0.94	7.24	31.13	132.34	0.10	0.13	0.15	0.06	0.09	0.37
50 states, 20 obs.	3.50	4.48	6.65	11.20	77.11	300.08	1.36	1.72	2.02	0.17	0.21	0.30
100 states, 20 obs.	12.08	20.11	28.01	51.14	-	-	8.52	10.55	12.10	0.46	0.66	0.82
Malware Domain, 60 obs.	2.93	4.30	7.04	25.05	-	-	1.06	1.45	2.15	0.08	0.15	0.23
10 states, 60 obs.	1.98	2.65	3.22	-	-	-	0.44	0.54	0.70	0.14	0.17	0.20
50 states, 60 obs.	18.93	61.96	134.84	-	-	-	9.49	12.52	15.48	0.35	0.60	0.80
100 states, 60 obs.	107.39	-	-	-	-	-	56.52	75.17	102.63	1.12	2.07	2.81
Malware Domain, 120 obs.	6.63	10.23	16.93	-	-	-	4.61	5.77	8.94	0.15	0.26	0.47
10 states, 120 obs.	5.83	9.28	22.23	-	-	-	1.81	2.40	3.27	0.27	0.33	0.41
50 states, 120 obs.	69.98	-	-	-	-	-	40.42	51.91	70.16	0.90	1.51	2.23
100 states, 120 obs.	-	-	-	-	-	-	229.22	294.80	-	2.81	5.35	7.68

Table 1: Relative performance: Minimum, maximum and average planning time, in seconds, for 15 instances of each size.

The results were obtained on the same problem instances, and help illustrate the advantages and disadvantages of the algorithms we evaluated. While iterative replanning is the easiest to implement and may perform well in practice on small instances, it is by far the worst performing, as expected. Iterative replanning results were omitted from Table 1 to save space, but they can be easily estimated based on "LAMA top-1", by multiplying the time of one iteration by a very optimistic estimate of the minimum number of iterations (in this case, 50). Branch and bound generalizes for a variety constraints and objective functions, but as expected, it is not as fast as specialized shortest paths algorithms.

The unexpected result is how well *TK* and *TK** perform, in comparison with the time it takes LAMA to find a single plan. While our implementation of the Eppstein's algorithm, very fast on small problems, is limited on large problems by the requirement to create the complete planning state graph, *TK** does not have that limitation, and performs much better. For example, for the largest problem size of 100 system states and 120 observations, *TK** is on average 55 times faster by the next fastest approach, *TK*.

We note that *TK** is the only approach that implements dynamic grounding. In our experiments action grounding is responsible for roughly half of the planning time of k shortest paths algorithm, and since the same grounding implementation is used for branch and bound, it suffers the same performance penalty.

The Impact of the Value of K

Above, we showed that *TK* and *TK** scale well with increasing problem size. Notably, these planners also scale well with increasing k. To measure this, we rerun the same experiments with k=1000, and these results, along with the results for k=50 from Table 1, for the two best planning approaches, are presented in Table 2.

As previously shown for *TK* and *TK**, while *TK** is faster overall, it is more sensitive to the value of k, and larger values lead to somewhat longer planning times. For example, for the largest problem of 100 states and 120 observations, for k=1000 the average planning time increases by 70% compared to k=50. *TK* spends significant time upfront computing a shortest path tree covering the entire state graph, but finding individual plans after that is very fast, and difference in planning time between k=50 and k=1000 for *TK* is negligible. For the same problem size, average planning time increases only by approximately 1%.

Conclusions

Our work on top-k planning is motivated by a specific application where finding multiple high-quality plans is required, namely hypothesis exploration for malware detection (Sohrabi, Udrea, and Riabov 2013). We proposed a new top-k plans formulation for cost-optimal planning, which can be used in this and other applications.

Generating diverse plans is a notable related work (e.g., (Myers and Lee 1999; Srivastava et al. 2007; Nguyen et al. 2012)). However, rather finding a representative set of plans, our approach focuses on computing top-k plans. Also, generating Pareto frontiers or a Pareto set (e.g., (Sroka and Long 2012; Khouadjia et al. 2013)) is related. Like diverse plans, this work does not focus on finding all top-k plans in any of the dimensions of the objective function, instead multiple diverse plans from a Pareto optimal curve are found. Furthermore, we do not rely on additional objectives to find near-optimal plans with a single objective.

We have implemented and evaluated four top-k planning algorithms, two based on the k shortest paths algorithm by Eppstein and its successor K^*. We also compared the result of these algorithms with the result of computing top-k plans from our iterative replanning and branch and bound algorithms. The results show that computation of top-k plans is comparable to the computation of a single cost-optimal plan. Additionally, we found that our top-k planning system based on the K^* algorithm, *TK**, is as expected, the most promis-

	TK, top-50			*TK*, top-1000			*TK* *, top-50			*TK* *, top-1000		
Problem size	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
10 states, 5 obs.	0.05	0.05	0.06	0.27	0.29	0.35	0.03	0.04	0.06	0.21	0.24	0.28
50 states, 5 obs.	0.18	0.21	0.24	0.42	0.45	0.48	0.06	0.08	0.10	0.24	0.28	0.41
100 states, 5 obs.	1.03	1.23	1.47	1.30	1.47	1.74	0.18	0.22	0.25	0.41	0.44	0.47
10 states, 10 obs.	0.06	0.07	0.08	0.42	0.49	0.53	0.04	0.06	0.07	0.32	0.38	0.42
50 states, 10 obs.	0.42	0.51	0.60	0.83	0.94	1.10	0.10	0.12	0.13	0.43	0.48	0.53
100 states, 10 obs.	2.62	3.43	3.97	3.09	3.85	4.44	0.21	0.35	0.55	0.67	0.75	0.81
10 states, 20 obs.	0.10	0.13	0.15	0.90	0.96	1.03	0.06	0.09	0.37	0.72	0.74	0.78
50 states, 20 obs.	1.36	1.72	2.02	2.23	2.56	2.85	0.17	0.21	0.30	0.81	0.89	0.93
100 states, 20 obs.	8.52	10.55	12.10	9.33	11.44	12.96	0.46	0.66	0.82	1.10	1.36	1.49
10 states, 60 obs.	0.44	0.54	0.70	2.69	2.95	3.34	0.14	0.17	0.20	1.98	2.10	2.24
50 states, 60 obs.	9.49	12.52	15.48	11.82	14.95	18.33	0.35	0.60	0.80	2.24	2.52	2.75
100 states, 60 obs.	56.52	75.17	102.63	57.93	77.72	106.43	1.12	2.07	2.81	2.96	4.07	4.73
10 states, 120 obs.	1.81	2.40	3.27	6.58	7.18	7.84	0.27	0.33	0.41	4.07	4.24	4.44
50 states, 120 obs.	40.42	51.91	70.16	45.92	57.22	75.92	0.90	1.51	2.23	4.67	5.40	6.01
100 states, 120 obs.	229.22	294.80	412.89	234.58	300.41	419.97	2.81	5.35	7.68	6.55	9.13	11.37

Table 2: The impact of the value of k: Minimum, maximum and average planning time, in seconds, for 15 instances of each size.

ing direction for top-k planners, and in our implementation it performed more than 100 times faster than all other algorithms (in part due to faster grounding). To conclude, the contribution of this paper is: 1) the formulation of the top-k planning problem and its reduction to a k shortest paths computation in a graph, 2) comparison of four implementations of the top-k planner, 3) experimental evaluation of performance of our implementations on synthetic benchmarks derived from a real-world scenario.

References

Aljazzar, H., and Leue, S. 2011. K*: A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence* 175(18):2129–2154.

Baier, J., and McIlraith, S. 2008. Planning with preferences. *AI Magazine* 29(4):25–36.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Botea, A., and Harabor, D. 2013. Path planning with compressed all-pairs shortest paths data. In *Proc. of the 23rd Int. Conference on Automated Planning and Scheduling (ICAPS)*, 293–297.

Botea, A. 2011. Ultra-fast optimal pathfinding without runtime search. In *Proc. of the 7th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 112–127.

Edelkamp, S., and Kissmann, P. 2009. Optimal symbolic planning with action costs and preferences. In *Proc. of the 21st Int. Joint Conference on Artificial Intelligence (IJCAI)*, 1690–1695.

Eppstein, D. 1998. Finding the k shortest paths. *SIAM Journal on Computing* 28(2):652–673.

Haslum, P., and Grastien, A. 2011. Diagnosis as planning: Two case studies. In *Int. Scheduling and Planning Applications woRKshop (SPARK)*, 27–44.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffman, W., and Pavley, R. 1959. A method for the solution of the *n*th best path problem. *Journal of the ACM* 6(4):506–514.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Kambhampati, S. 2007. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *Proc. of the 22nd National Conference on Artificial Intelligence (AAAI)*, 1601–1604.

Keyder, E., and Geffner, H. 2009. Soft Goals Can Be Compiled Away. *Journal of Artificial Intelligence Research* 36:547–556.

Khouadjia, M. R.; Schoenauer, M.; Vidal, V.; Dréo, J.; and Savéant, P. 2013. Pareto-based multiobjective AI planning. In *Proc. of the 23rd Int. Joint Conference on Artificial Intelligence (IJCAI)*, 2321–2327.

McDermott, D. V. 1998. PDDL — The Planning Domain Definition Language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Myers, K. L., and Lee, T. J. 1999. Generating qualitatively different plans through metatheoretic biases biases. In *Proc. of the 16th National Conference on Artificial Intelligence (AAAI)*, 570–576.

Nguyen, T. A.; Do, M. B.; Gerevini, A.; Serina, I.; Srivastava, B.; and Kambhampati, S. 2012. Generating diverse plans to handle unknown and partially known user preferences. *Artificial Intelligence* 190:1–31.

Ramírez, M., and Geffner, H. 2009. Plan recognition as planning. In *Proc. of the 21st Int. Joint Conference on Artificial Intelligence (IJCAI)*, 1778–1783.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Sohrabi, S.; Baier, J.; and McIlraith, S. 2010. Diagnosis as planning revisited. In *Proc. of the 12th Int. Conference on the Principles of Knowledge Representation and Reasoning (KR)*, 26–36.

Sohrabi, S.; Baier, J. A.; and McIlraith, S. A. 2011. Preferred explanations: Theory and generation via planning. In *Proc. of the 25th National Conference on Artificial Intelligence (AAAI)*, 261–267. Accepted as both oral and poster presentation.

Sohrabi, S.; Udrea, O.; and Riabov, A. 2013. Hypothesis exploration for malware detection using planning. In *Proc. of the 27th National Conference on Artificial Intelligence (AAAI)*, 883–889.

Srivastava, B.; Nguyen, T. A.; Gerevini, A.; Kambhampati, S.; Do, M. B.; and Serina, I. 2007. Domain independent approaches for finding diverse plans. In *Proc. of the 20th Int. Joint Conference on Artificial Intelligence (IJCAI)*, 2016–2022.

Sroka, M., and Long, D. 2012. Exploring metric sensitivity of planners for generation of pareto frontiers. In *Proc. of the 6th Starting AI Researchers' Symposium (STAIRS)*, 306–317.

Yen, J. 1971. Finding the k shortest loopless paths in a network. *Management Science* 17:712–716.

Zhu, A. D.; Ma, H.; Xiao, X.; Luo, S.; Tang, Y.; and Zhou, S. 2013. Shortest path and distance queries on road networks: towards bridging theory and practice. In *ACM SIGMOD Int. Conference on Management of Data (SIGMOD)*, 857–868.