K*: Heuristics-Guided, On-the-Fly k Shortest Paths Search

Husain Aljazzar and Stefan Leue

Department of Computer and Information Science, University of Konstanz
D-78457 Konstanz, Germany

Abstract. We present a search algorithm, called K*, for finding the k shortest paths (KSP) between a designated pair of vertices in a given directed weighted graph. As a directed algorithm, K* has two advantages compared to current KSP algorithms. First, K* performs on-the-fly, which means that it does not require the graph to be explicitly available and stored in main memory. Portions of the graph will be generated as needed. Second, K* can be guided using heuristic functions. We discuss the properties of K*, including its correctness, and its asymptotic worst-case complexity, which has been shown to be of $\mathcal{O}(m + n \log n + k)$ with respect to both runtime and space, where n is the number of vertices and m is the number of edges of the graph. We report on experimental results which illustrate the favorable performance of K* compared to the most efficient k-shortest-paths algorithms known so far. In other work it has been shown that K* can be used to efficiently compute counterexamples for stochastic model checking.

1 Introduction

In this paper we consider the *k-Shortest-Paths* problem (KSP) which is about finding the k shortest paths from a start vertex $\mathbf{s}$ to a target vertex $\mathbf{t}$ in a directed weighted graph G for an arbitrary natural number k. Application domain examples for KSP problems include logistics, finance analysis, scheduling, sequence alignment, networking and many other optimisation applications. The initial motivation for our work stems from work on the generation of counterexamples for stochastic model checking, which can be cast as a KSP problem [10, 1].

Based on demands imposed by this problem domain we are interested in a variant of the KSP problem in which solution paths containing loops are allowed. We also assume that the number k is unknown at the beginning of the search. In other words, we aim at enumerating up to k paths from $\mathbf{s}$ to $\mathbf{t}$, including loops, in a non-increasing order with respect to their length. The most advantageous algorithm for solving this problem with respect to the worst-case runtime complexity is the one presented by Eppstein in [7], and the optimized lazy version of it presented in [11]. Whenever we refer to Eppstein's algorithm in the remainder of this paper, we mean to denote the lazy variant of it.

A salient feature of Eppstein's algorithm is that it requires the complete problem graph G to be available when the search starts. It also requires that an exhaustive search is performed on G in order to return any result at all. These are major drawbacks from a practical point of view, in particular if G is large. In order to address this problem we developed an algorithm called K*. For a graph with n vertices and m edges, K*

has an asymptotic worst-case runtime complexity of $\mathcal{O}(m + n \log n + k \log k)$ which can even be optimised to $\mathcal{O}(m + n \log n + k)$ [8, 9]. The space complexity of K* is $\mathcal{O}(m + n \log n + k)$. In other words, K* maintains the same asymptotic worst-case complexity as Eppstein's algorithm in terms of both runtime and space. On the other hand, the major two advantages of K* over existing KSP algorithms are the following:

- K* performs on-the-fly in the sense that it does not require the graph to be explicitly available and to be stored in the main memory. It partially generates and processes the graph as the need arises. Solution paths are computed earlier and made available as soon as they are computed.
- K* takes advantage of the heuristics-guided search, which often leads to significant improvements in terms of memory and runtime effort.

As our experimental evaluation shall illustrate, K* performs very favorably compared to Eppstein's algorithm when applied to route planning problems.

Related Work. A discussion of the counterexample generation for stochastic model checking problem and how it is represented as a variant of the KSP problem can be found in [10, 4, 3, 1]. The use of K* in the computation of counterexamples in stochastic model checking has been discussed in [1, 4, 2]. This paper hence focuses on the description of the algorithmic structure of K* and on a discussion of its properties.

2 Preliminaries

Let $G = (V, E)$ be a directed graph and $c : E \to \mathbb{R}_{\geq 0}$ be a length function mapping edges to non-negative real values. The length of a path $\pi = v_0 \to v_1 \to \ldots \to v_n$ is defined as the sum of the edge lengths, formally, $C(\pi) = \sum_{i=0}^{n-1} c(v_i, v_{i+1})$. For an arbitrary pair of vertices u and v, $\mathit{\Pi}(u, v)$ refers to the set of all paths from u to v. $C^*(u, v)$ denotes the length of the shortest path from u to v. If there is no path from u to v, then $C^*(u, v)$ is equal to $+\infty$. Let $\mathbf{s}, \mathbf{t} \in V$ denote vertices which we consider as a source and a target, respectively.

The Shortest-Path Problem (SP) is the problem of finding a path $\pi^* \in \mathit{\Pi}(\mathbf{s}, \mathbf{t})$ with $C(\pi^*) = C^*(\mathbf{s}, \mathbf{t})$. Dijkstra's algorithm is the most prominent algorithm for solving the SP problem [6]. Dijkstra's algorithm stores vertices on the search front in a priority queue open, which is ordered according a distance function d. Initially, open contains only the start vertex $\mathbf{s}$ with $d(\mathbf{s}) = 0$. In each search iteration, the head of the queue open is removed from the queue and *expanded*. We distinguish between two sets of *visited* vertices, namely *closed* and *open* vertices. Closed vertices are those which have been visited and expanded, where as open vertices are those which have been visited but have not yet been expanded, i.e., vertices in the search queue. For each visited vertex v, $d(v)$ is always equal to the length of some path from $\mathbf{s}$ to v which we call the *solution base* of v. The set of these solution bases forms a *search tree* T.

Directed graph search algorithms are on-the-fly algorithms that work on an implicit description of the search graph G. They are commonly guided by a heuristic evaluation function that aids in finding target nodes faster. The most prominent directed algorithm is A* [12] which is designed for solving the SP problem. It uses a *heuristic evaluation function* f to sort the search queue open. f is defined as the sum of two functions g and h, i.e., $f = g + h$. The function g is given by the solution base of some vertex v. h is the heuristic estimate of the length of an $\mathbf{s}$-$\mathbf{t}$ path through v. h is called *admissible* if $h(v) \leq C^*(v, \mathbf{t})$ for any vertex v. An admissible heuristic guarantees the solution optimality of A*. h is called *monotone* or *consistent* if for each edge (u, v) in G it holds that $h(u) \geq c(u, v) + h(v)$. Most directed search algorithms including A* have, in general, an exponential worst-case complexity but a good average-case performance. In the case of a monontone heuristic, A* has a worst-case complexity of $\mathcal{O}(m + n \log n)$ which is the same complexity of Dijkstra's algorithm.

The k-Shortest-Paths Problem (KSP) is a generalized form of the SP problem in which one determines the k shortest paths from the start vertex $\mathbf{s}$ to the target $\mathbf{t}$ for an arbitrary natural number k. In this paper we consider a variant of the KSP problem in which k does not need to be specified in advance, and where loops are allowed in the solution paths. We aim at enumerating the paths from $\mathbf{s}$ to $\mathbf{t}$, including loops, in a non-increasing order with respect to their length.

Eppstein's algorithm first applies Dijkstra's algorithm to the given graph G in reverse direction. The search starts at the target $\mathbf{t}$ and traces the edges lying on shortest paths back to their origin vertices. The result is a "reversed" search tree T rooted at $\mathbf{t}$ containing the shortest path from any vertex in G to $\mathbf{t}$. An edge (u, v) either belongs to the search tree T, in which case we call it a *tree edge*; otherwise we call it a *sidetrack edge*. The notion of a sidetrack edge is interesting because choosing any such edge $(u, v) \in G - T$ entails a certain detour compared to the shortest path. For any $\mathbf{s}$-$\mathbf{t}$ path π, we denote as $\xi(\pi)$ the subsequence of sidetrack edges which are taken in π. As Eppstein shows, π can be unambiguously described by the sequence $\xi(\pi)$. In other words, each $\mathbf{s}$-$\mathbf{t}$ path can be represented using its sidetrack sequence. Eppstein's algorithm uses a special data structure called *path graph* $\mathcal{P}(G)$ to save all sidetrack edges. The path graph $\mathcal{P}(G)$ is a directed weighted graph with a designated root. Its nodes represent sidetrack edges of G. The nodes are organized in $\mathcal{P}(G)$ using a heap landscape. The structure of $\mathcal{P}(G)$ ensures that any path in $\mathcal{P}(G)$, from the root to an arbitrary node, corresponds to a sidetrack representation of a valid $\mathbf{s}$-$\mathbf{t}$ path in G. Moreover, the shorter the $\mathcal{P}(G)$ path, the shorter is the corresponding $\mathbf{s}$-$\mathbf{t}$ path in G. Consequently, applying Dijktra's algorithm to $\mathcal{P}(G)$ results in finding the k shortest $\mathbf{s}$-$\mathbf{t}$ paths in G. Space limitation do not allow us to present a more detailed description of Eppstein's algorithm and we refer the interested reader to [7] for a more elaborate discussion. Notice that the structure of $\mathcal{P}(G)$ is very similar to the structure of the path graph used in K*, which we describe in the next Section.

3 The K* Algorithm

As in Eppstein's algorithm, K* performs a shortest path search on G and uses a path graph structure $\mathcal{P}(G)$. The path graph is searched using Dijkstra in order to determine the $\mathbf{s}$-$\mathbf{t}$ paths in the form of sidetrack sequences. The main design principles of K* are the following:

1. K* applies A* to G instead of the backwards Dijkstra construction in Eppstein's algorithm.
2. We execute A* on G and Dijkstra on $\mathcal{P}(G)$ in an interleaved fashion, which allows Dijkstra to deliver solution paths prior to the the completion of the search of G by A*.

A Search on G.* K* applies the A* search to the problem graph G in order to determine a search tree T. Unlike Eppstein's algorithm, in K*, A* is applied to G in a forward manner, which yields a search path tree T rooted at the start vertex $\mathbf{s}$. This is necessary in order to be able to work on the implicit description of the problem graph G using the successor function $succ$. Each edge discovered during the A* search of G will immediately be inserted into the graph $\mathcal{P}(G)$, the structure of which will be explained next.

Example 1. If we apply K* to the graph from Figure 1, then A* yields a search tree such as the one shown in Figure 2. Tree edges are drawn with heavy lines whereas sidetrack edges are drawn with thin lines. Unlike the reversed shortest path tree shown in Figure 1, the search tree of A* is a forward tree rooted at the start vertex s_0.

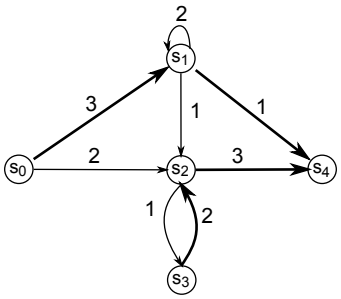

Fig. 1. Tree and sidetrack edges

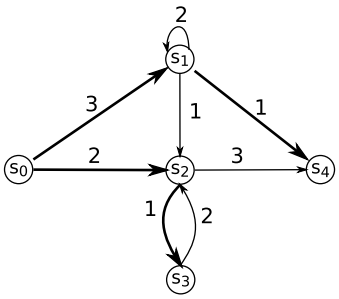

Fig. 2. Search tree of A*

As mentioned before, a sidetrack edge may lead to take a certain *detour* instead of the shortest path. We can measure this detour using the *detour function* δ. For an edge (u, v), $\delta(u, v)$ indicates the disadvantage of taking this edge in comparison to the shortest $\mathbf{s}$-$\mathbf{t}$ path via v. Neither the length of the shortest $\mathbf{s}$-$\mathbf{t}$ path through v nor the length of the $\mathbf{s}$-$\mathbf{t}$ path which includes the sidetrack edge (u, v) are known when (u, v) is discovered by A*. Both lengths can only be estimated using the evaluation function f.

Let $f(v)$ be the f-value of v according to the search tree T and $f_u(v)$ be the f-value of v according to the parent u, i.e., $f_u(v) = g(u) + c(u,v) + h(v)$. $\delta(u,v)$ is then defined as:

$$\begin{aligned} \delta(u,v) &= f_u(v) - f(v) \\ &= g(u) + c(u,v) + h(v) - g(v) - h(v) \\ &= g(u) + c(u,v) - g(v) \end{aligned} \tag{1}$$

Path Graph Structure. The structure $\mathcal{P}(G)$ will be a directed graph, the vertices of which correspond to edges in the problem graph G. The path graph $\mathcal{P}(G)$ is organized as a heap landscape. Two binary min heap structures are assigned to each vertex v in G, namely an *incoming heap* $H_{in}(v)$ and a *tree heap* $H_T(v)$. These heap structures are the basis of $\mathcal{P}(G)$. The incoming heap $H_{in}(v)$ contains a node for each incoming sidetrack edge of v which has been discovered. The nodes of $H_{in}(v)$ will be ordered according to the δ-values of the corresponding transitions. The node possessing the edge with minimal detour is placed on the top of the heap. We constrain the structure of $H_{in}(v)$ so that its root, unlike all other nodes, has one child at most. We denote the root of $H_{in}(v)$ as $root_{in}(v)$. Moreover, we refer to the incoming tree edge of v as $edge_T(v)$.

Example 2. Figure 3 illustrates the incoming heaps of the graph from Figure 2. The numbers attached to the heap nodes are the corresponding δ-values.

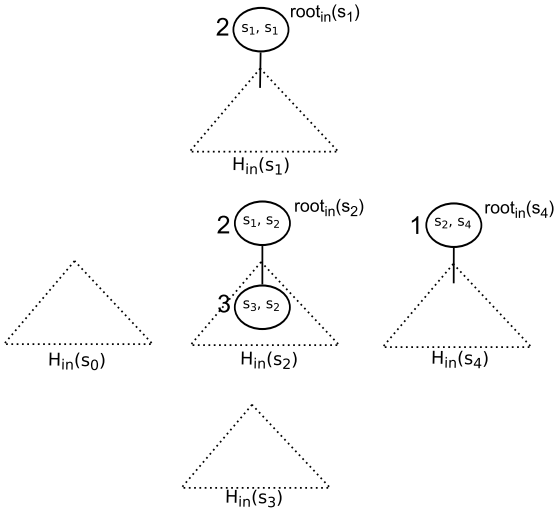

Fig. 3. The incoming heaps of the graph from Figure 2

The tree heap $H_T(v)$, for an arbitrary vertex v, is built as follows. If v is the start vertex $\mathbf{s}$, then $H_T(\mathbf{s})$ is created as a fresh empty heap. Next, $root_{in}(\mathbf{s})$ is added into it, if $H_{in}(\mathbf{s})$ is not empty. If v is not the start vertex, then let u be the parent of v in the search tree T. The tree heap $H_T(v)$ is constructed by inserting $root_{in}(v)$ into $H_T(u)$

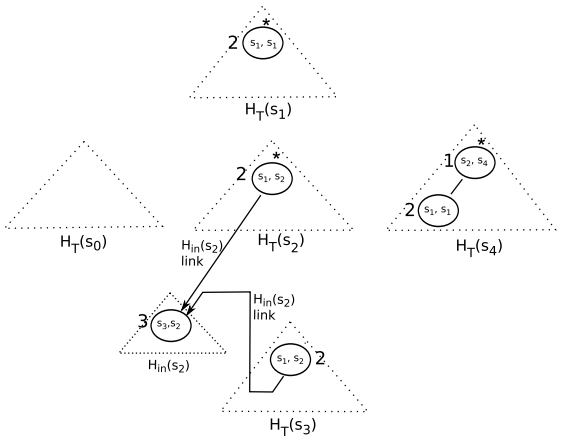

Fig. 4. The tree heaps of the graph from Figure 2

if $H_{in}(v)$ is not empty. In addition to maintaining the pointers attached to $root_{in}(v)$ when it is added into $H_T(u)$, we ensure that $root_{in}(v)$ keeps referring to its only child in $H_{in}(v)$. The insertion of $root_{in}(v)$ into $H_T(u)$ is done in a non-destructive fashion as explained in [7]. This is accomplished by creating new copies of the heap nodes which lie on the updated path in $H_T(u)$ such that the heap $H_T(u)$ will not be changed. In order to simplify matters we can imagine that $H_T(v)$ is constructed as a copy of $H_T(u)$ into which $root_{in}(v)$ is added. If $H_{in}(v)$ is empty, then $H_T(v)$ is identical to $H_T(u)$. We refer to the root of $H_T(v)$ as $R(v)$.

Example 3. Figure 4 illustrates the tree heaps of the graph from Figure 2. The numbers attached to the heap nodes are the corresponding δ-values. We denote the newly created or copied nodes using asterisks. $H_T(s_0)$ is empty since s_0 has no incoming sidetrack edges at all. The heap $H_T(s_1)$ is constructed by adding $root_{in}(s_1)$ into $H_T(s_0)$ since s_0 is the predecessor of s_1 in the search tree. Notice that the heap $H_T(s_0)$ is preserved. The heap $H_T(s_2)$ is built in the same way as $H_T(s_1)$. Notice that $root_{in}(s_2) = (s_1, s_2)$ has a child in $H_{in}(s_2)$ which is the node (s_3, s_2), cf. Figure 3. This child persists after adding $root_{in}(s_2)$ into the tree heap. The heap $H_T(s_3)$ is identical to the heap $H_T(s_2)$ since $H_{in}(s_3)$ is empty, cf. Figure 3. The heap $H_T(s_4)$ is constructed by adding $root_{in}(s_4)$, i.e. (s_2, s_4), into the heap $H_T(s_1)$. Notice that s_1 is the predecessor of s_4 in the search tree.

The final structure of $\mathcal{P}(G)$ is derived from the incoming and tree heaps as follows. To each node n of $\mathcal{P}(G)$ carrying an edge (u, v), we attach a pointer referring to $R(u)$. We call such pointers *cross edges*, whereas the pointers which arise from the heap structures are called *heap edges*. Moreover, we add a special node $*$ to $\mathcal{P}(G)$ with a single outgoing cross edge to $R(\mathbf{t})$. As from now, when we refer to paths in $\mathcal{P}(G)$, we mean paths in $\mathcal{P}(G)$ which start at $*$. Furthermore, we define a length function Δ on the edges of $\mathcal{P}(G)$. Let (n, n') denote an edge in $\mathcal{P}(G)$, and let e and e' denote the edges from G

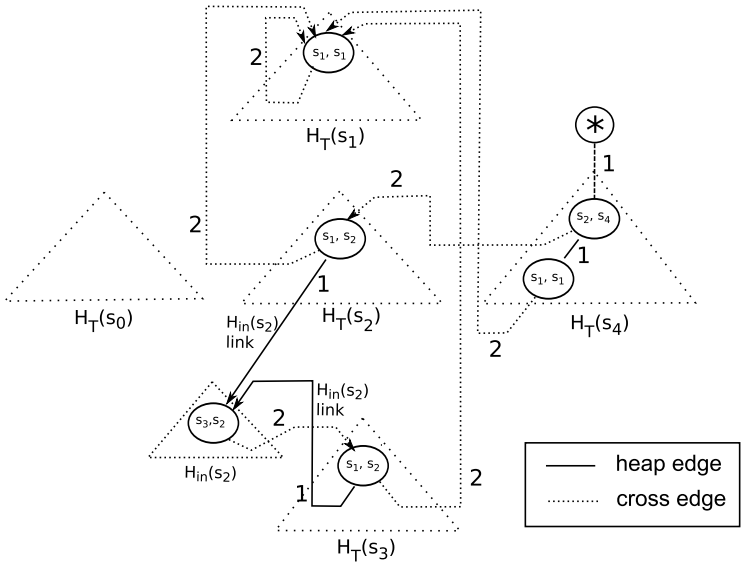

Fig. 5. The path graph of the graph from Figure 2

corresponding to n and n'. Then we define $\Delta(n, n')$ as follows:

$$\Delta(n, n') = \begin{cases} \delta(e') - \delta(e), & (n, n') \text{ is a heap edge} \\ \delta(e'), & (n, n') \text{ is a cross edge} \end{cases} \tag{2}$$

Similar to [7], we can deduce that all nodes, which are reachable via heap edges from $R(v)$ for any vertex v, form a 3-heap $H_G(v)$ that is ordered according to the δ values. This heap order implies that Δ is not negative, i.e. $\Delta(n, n') \geq 0$, for any edge (n, n') in $\mathcal{P}(G)$. The length of path σ, i.e. $C(\sigma)$, is equal to $\sum_{e \in \sigma} \Delta(e)$. Each node in $H_G(v)$ corresponds to a sidetrack edge (q, r) where there is a path in the search tree T from r to v.

Example 4. Figure 5 shows the final path graph obtained from the graph from Figure 2. Notice that the weights are now assigned to the edges. These weights are computed according to the weighting function Δ.

An arbitrary path $\sigma = n_0 \rightarrow \ldots \rightarrow n_r$ through the path graph $\mathcal{P}(G)$ (starting at $*$, i.e., $n_0 = *$) can be interpreted as a recipe for constructing a unique $\mathbf{s}$-$\mathbf{t}$ path. Each cross edge (n_i, n_{i+1}) in σ represents the selection of the sidetrack edge associated to n_i. The same holds if n_i is the last node of σ. A heap edge (n_i, n_{i+1}) represents considering the sidetrack edge associated with the node n_{i+1} instead of the one associated with n_i. Based on this interpretation we derive from σ a sequence of edges $seq(\sigma)$ using the following procedure. At the beginning, let $seq(\sigma)$ be an empty sequence. Then, we iterate over the edges of σ. For each cross edge (n_i, n_{i+1}) in σ, with $n_i \neq *$, we add to $seq(\sigma)$ the edge associated with n_i. Finally, we add to $seq(\sigma)$ the edge associated with the last node of σ, i.e. n_r. The structure of $\mathcal{P}(G)$ ensures that $seq(\sigma)$ represents a

valid $\mathbf{s}$-$\mathbf{t}$ path. Formally, $seq(\sigma)$ is in the range of the mapping ξ. The full $\mathbf{s}$-$\mathbf{t}$ path is $\chi(seq(\sigma))$. In other words, we obtain the full $\mathbf{s}$-$\mathbf{t}$ from $seq(\sigma)$ by completing it with the possibly missing tree edges up to $\mathbf{s}$. The structure of $\mathcal{P}(G)$ ensures that two different paths in $\mathcal{P}(G)$ induce two different sequences of sidetrack edges and, consequently, two different $\mathbf{s}$-$\mathbf{t}$ paths in G. Altogether, we get a one-to-one correspondence between $\mathbf{s}$-$\mathbf{t}$ paths in G and paths in $\mathcal{P}(G)$. Thus, there is a well-defined, bijective mapping $p = \chi \circ seq$ from paths in $\mathcal{P}(G)$ onto $\mathbf{s}$-$\mathbf{t}$ paths in G. Moreover, we can establish a correlation between the length of a path in $\mathcal{P}(G)$ and the corresponding $\mathbf{s}$-$\mathbf{t}$ path in G. We state this in the following lemma.

Lemma 1. *Let σ be a path in $\mathcal{P}(G)$ starting at $*$. If h is admissible, then it holds that $C_G(p(\sigma)) = C^*_G(\mathbf{s}, \mathbf{t}) + C_{\mathcal{P}(G)}(\sigma)$.*

We now know that shorter $\mathcal{P}(G)$ paths lead to shorter $\mathbf{s}$-$\mathbf{t}$ paths. This property enables computing shortest $\mathbf{s}$-$\mathbf{t}$ paths using Dijkstra for shortest path on $\mathcal{P}(G)$ starting at $*$.

*The algorithmic structure of K** can be described as follows. We execute A* to search in G and Dijkstra to search in $\mathcal{P}(G)$ in an interleaving fashion. First, we run A* on G until the target vertex $\mathbf{t}$ is found. Then, we run Dijkstra on the portion of $\mathcal{P}(G)$ that A* made available. If Dijkstra finds k shortest paths, then K* terminates successfully. Otherwise, A* is resumed to explore a bigger portion of G and, thereafter, Dijkstra is resumed to search on the incremented $\mathcal{P}(G)$. We repeat this process until Dijkstra succeeds in finding k shortest paths. Algorithm 1 contains the pseudocode of K*.

K* maintains a *scheduling mechanism* to control whether A* or Dijkstra should be resumed. If the queue of A* is not empty, which means that A* has not yet finished exploring the whole graph G, then Dijkstra will be resumed if and only if $g(\mathbf{t}) + d \leq f(u)$ (c.f. Line 13). The value d is the maximum d value of all successors of the head of Dijkstra's search queue n. The vertex u is the head of the search queue of A*. If Dijkstra's search queue is empty or $g(\mathbf{t}) + d > f(u)$, then A* will be resumed in order to explore a bigger portion of G (c.f. Line 14). How long we let A* run is a trade off. If we run it only for a short time we give Dijkstra the chance to find the needed number of paths sooner once they are available in $\mathcal{P}(G)$. On the other hand, we cause an overhead by switching between A* and Dijkstra. Note that after resuming A* at Line 14, the structure of $\mathcal{P}(G)$ may change. Thus, we need to refresh $\mathcal{P}(G)$ at Line 15. This requires a subsequent inspection of the state of Dijkstra's search. We have to ensure that Dijkstra's search retains a consistent state after the changes in $\mathcal{P}(G)$. K* stipulates a condition, which we refer to as *extension condition*, which governs the decision of when to stop A*. We can show that A* must run until the number of closed vertices is doubled or G has been searched completely, in order to maintain the same worst case runtime complexity as Eppstein's algorithm. However, other conditions can be more effective in practice. In our experiments we define the extension condition so that the number of closed vertices or the number of explored edges grows by 20 percent in each run of A*. The scheduling mechanism is enabled as long as A* remains incomplete. Once A* has explored the entire graph G (c.f. **if**-statement at Line 9) the scheduling mechanism is disabled and henceforth, only Dijkstra will be executed.

Algorithm 1: The K* Algorithm

```
   Data: A graph given by its start vertex s ∈ V and its successor function succ and a natural number k
   Result: A list R containing k sidetrack edge sequences representing k solution paths
 1 open_D ← empty priority queue.
 2 closed_D ← empty hash table.
 3 R ← empty list.
 4 P(G) ← empty path graph
 5 Run A* on G until t is selected for expansion.
 6 if t was not reached then Exit without a solution.
 7 Add ∗ into open_D.
 8 while A* queue or open_D is not empty do
 9     if A* queue is not empty then
10         if open_D is not empty then
11             Let u be the head of the search queue of A* and n the head of open_D.
12             d ← max{ d(n) + Δ(n, n') | n' ∈ succ(n) }.
13             if g(t) + d ≤ f(u) then Go to Line 17.
14         Resume A* in order to explore a larger portion of G.
15         Refresh P(G) and bring Dijkstra's search into a consistent state.
16         Go to Line 8.
17     if open_D is empty then Go to Line 8.
18     Remove from open_D and place on closed_D the node n with the minimal d-value.
19     foreach n' referred by n in P(G) do
20         d(n') := d(n) + Δ(n, n')
21         Attach to n' a parent link referring to n.
22         Insert n' into open_D.
23     Let σ be the path in P(G) via which n was reached.
24     Add seq(σ) at the end of R.
25     if |R| = k then Go to Line 26.
26 Return R and exit.
```

The lines from 18 to 22 represent the usual node expansion step of Dijkstra. Note that when a successor node n' is generated, K* does not check whether n' has previously been visited. This strategy is justified by the observation that a $\mathbf{s}$-$\mathbf{t}$ path may take the same edge several times.

The fact that both algorithms A* and Dijkstra share the path graph $\mathcal{P}(G)$ gives rise to concerns regarding the correctness of the Dijkstra's search on $\mathcal{P}(G)$. Resuming A* results in changes in the structure of $\mathcal{P}(G)$. Thus, after resuming A*, we refresh $\mathcal{P}(G)$ and inspect the state of Dijkstra's search, see Line 15. A* may add new nodes, change the δ-values of existing ones or even remove ones. It can also significantly change the search tree T which destroys, in the worst case, the structure of all H_T heaps. This would make the previous Dijkstra's search on $\mathcal{P}(G)$ useless. This means that, in the worst case, we have to fully reconstruct $\mathcal{P}(G)$ and restart Dijkstra from scratch. However, if the used heuristic is admissible we find ourselves in a better situation. We may still need to restructure the $\mathcal{P}(G)$ considerably, but we do not lose the results of Dijkstra's search thus far. We can prove that the subsequent changes do not influence the segment which Dijkstra has already explored, if the heuristic h is admissible. In other words, the correctness of Dijkstra is maintained. However, the changes in $\mathcal{P}(G)$ can interfere with the completeness of Dijkstra's search. It is possible for a node n' to be attached to another node n, as a child, after n has been expanded. In this case the siblings

of n' will have been explored before n' became a child of n. We must then consider what has been missed during the search due to the absence of n'. It can be proven that it is sufficient in such a case to apply the lines from 20 to 22 to n' for each expanded direct predecessor n'. Notice that if n' does not fulfill the scheduling condition, A* will be repeatedly resumed until the scheduling mechanism allows Dijkstra to put n' into its search queue. Notice also that catching up the exploration of n' does not require extra effort during the typical Dijkstra search.

Example. We examine the directed, weighted graph G in Figure 6. The start vertex is s_0 and the target vertex is s_6. We are interested in finding the 9 best paths from s_0 to s_6. To meet this objective we apply K* to G. We assume that a heuristic estimate exists which indicates the heuristic values $h(s_0)$ to $h(s_6)$ annotated in Figure 6. A simple check will ensure that this heuristic function is admissible.

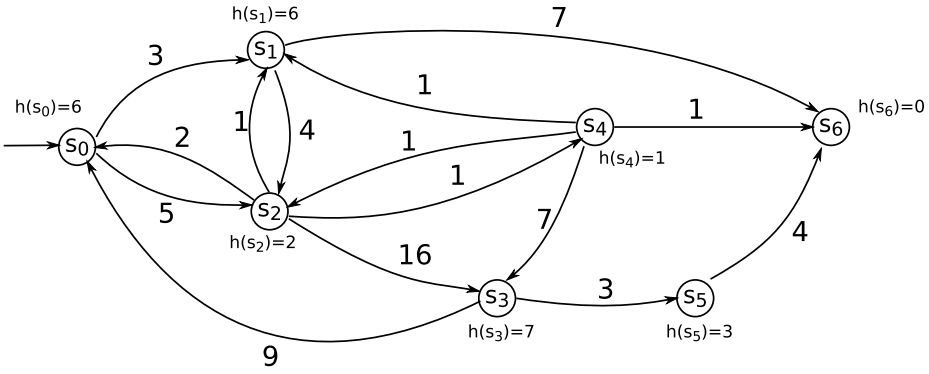

Fig. 6. The problem graph G

At first, A* searches graph G until s_6 is found. The section of G explored so far is illustrated in Figure 7. The edges that are highlighted with heavy lines signify the tree edges, while all of the other edges are sidetrack edges which are stored in H_{in} heaps, as shown in Figure 8. The numbers attached to the heap nodes are the corresponding δ-values. At this point A* is suspended and $\mathcal{P}(G)$ is constructed. Initially, only the designated root $*$ is explicitly available in $\mathcal{P}(G)$. Dijkstra's algorithm is initialized. This means, the node $*$ is added into Dijkstra's search queue. The scheduler needs to access the successors of $*$ in order to decide whether it is Dijkstra or A* that should be resumed. At this point the tree heap $H_T(s_6)$ should be built. The heap $H_T(s_4)$ is required for the building of $H_T(s_6)$. Consequently, the tree heaps $H_T(s_6)$, $H_T(s_4)$, $H_T(s_2)$ and $H_T(s_0)$ are built. The tree heaps s_1 and s_3 are not built because the were not needed for building $H_T(s_6)$. The result is shown in Figure 10, where solid lines represent heap edges and dashed lines indicate cross edges.

After constructing $\mathcal{P}(G)$, as shown in Figure 10, the scheduler checks for the only child (s_4, s_2) of $*$ whether $g(s_6) + d(s_4, s_2) \leq f(s_1)$. Note that s_1 is the head of the search queue of A*. The value $d(s_4, s_2)$ is equal to 2. Then, it holds that $g(s_6) +$

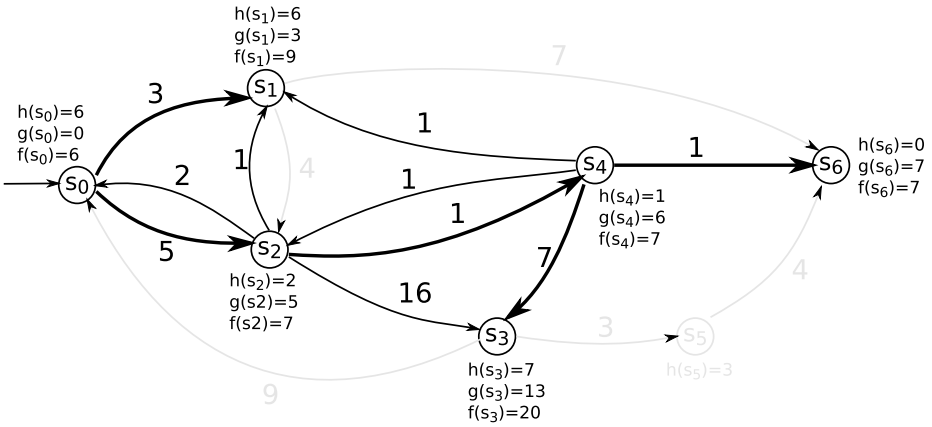

Fig. 7. The explored part of the graph G

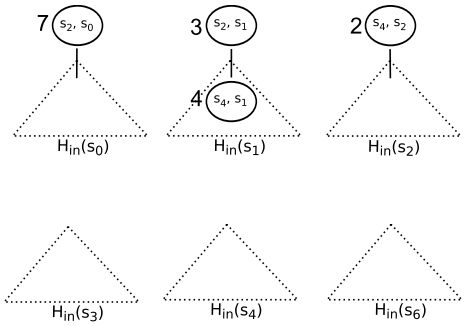

Fig. 8. The H_{in} heaps constructed by K*

$d(s_4, s_2) = 7 + 2 = 9 = f(s_1)$. Hence, the scheduler allows Dijkstra's algorithm to expand $*$ and insert (s_4, s_2) into its search queue. On expanding $*$ the first solution path is delivered. It is constructed from the $\mathcal{P}(G)$ path consisting of the single node $*$. This path results in an empty sequence of sidetrack edges. The empty sidetrack sequence corresponds to the tree path s_0 to s_6, namely $s_0\, s_2\, s_4\, s_6$ with the length 7. After this step the Dijkstra's search is suspended because the successors of (s_4, s_2) do not fulfil the scheduling condition $g(s_6) + d(n) \leq f(s_1)$.

For simplicity we assume the extension condition to be defined as the expansion of one vertex. Consequently, A* now expands s_1 and stops. The explored part of G at this point is given in Figure 11. This extension results in the the detection of two new sidetrack edges (s_1, s_2) and (s_1, s_6) which are added into $H_{in}(s_2)$ and $H_{in}(s_6)$ respectively. The modified heaps $H_{in}(s_2)$ and $H_{in}(s_6)$ are represented in Figure 9. The other H_{in} heaps remain unchanged as in Figure 8. The path graph $\mathcal{P}(G)$ is rebuilt as shown in Figure 12 and Dijkstra's algorithm is resumed. We recall that, at this point, Dijkstra's search queue contains only (s_4, s_2) with $d = 2$. It is easy to see that Dijkstra will deliver the solution paths enumerated in Table 1.

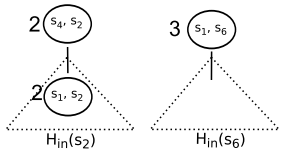

Fig. 9. The modified H_{in} heaps after the extension

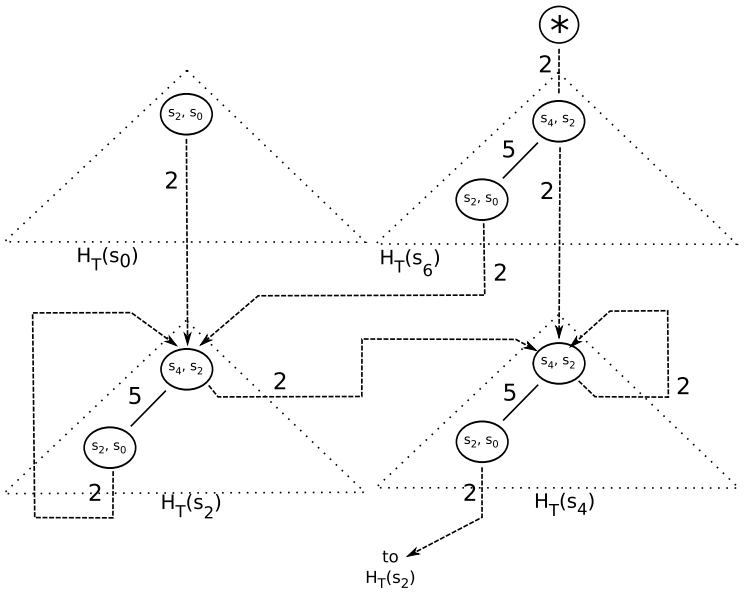

Fig. 10. The path graph $\mathcal{P}(G)$ constructed by K*

Properties of K.* The properties of K* have been studied in detail in [1]. For reasons of conciseness we only summarize those findings here:

- K* was proven to be correct, which means that applied to an arbitrary locally finite directed graph, it delivers valid $\mathbf{s}$-$\mathbf{t}$ paths.
- K* was shown to be complete when applied to a locally finite directed graph. This means that it finds k $\mathbf{s}$-$\mathbf{t}$ paths for any natural number k if $|\Pi(\mathbf{s},\mathbf{t})| \geq k$, or $|\Pi(\mathbf{s},\mathbf{t})|$ such paths otherwise. $\Pi(\mathbf{s},\mathbf{t})$ denotes the set of all $\mathbf{s}$-$\mathbf{t}$ paths.
- K* was shown to terminate on finite graphs. It could even be shown that for $k \leq |\Pi(\mathbf{s},\mathbf{t})|$, K* terminates on infinite graphs.
- K* was proven to be admissible. This means, if h is admissible, then, at any point of the search, the $\mathbf{s}$-$\mathbf{t}$ paths that are delivered are the shortest possible paths. From this result it can immediately be concluded that K* indeed solves the KSP problem if h is admissible.
- The worst-case runtime complexity of K* was proven to be of $\mathcal{O}(m + n \log n + k \log k)$, where n is the number of vertices and m is the number of edges of the graph. Using the results from [9] it can even be improved to $\mathcal{O}(m + n \log n + k)$.

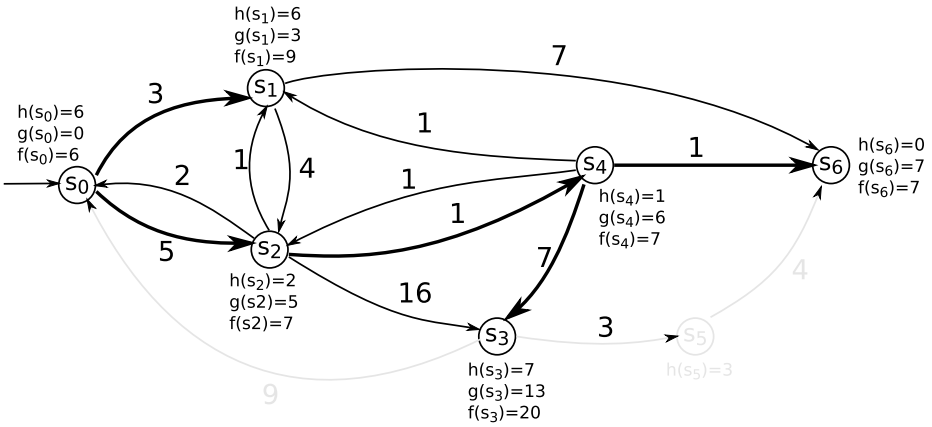

Fig. 11. The explored part of G after the extension

	$\mathcal{P}(G)$ Path	Sidetrack Seq.	s_0-s_6 Path (π)	$C(\pi)$
1.	$*$	$\langle\rangle$	$s_0\ s_2\ s_4\ s_6$	7
2.	$*, (s_4, s_2)$	$\langle(s_4, s_2)\rangle$	$s_0\ s_2\ s_4\ s_2\ s_4\ s_6$	9
3.	$*, (s_4, s_2), (s_1, s_2)$	$\langle(s_1, s_2)\rangle$	$s_0\ s_1\ s_2\ s_4\ s_6$	9
4.	$*, (s_4, s_2), (s_1, s_6)$	$\langle(s_1, s_6)\rangle$	$s_0\ s_1\ s_6$	10
5.	$*, (s_4, s_2), (s_4, s_2)$	$\langle(s_4, s_2), (s_4, s_2)\rangle$	$s_0\ s_2\ s_4\ \ s_2\ s_4\ s_2\ s_4\ s_6$	11
6.	$*, (s_4, s_2), (s_4, s_2), (s_1, s_2)$	$\langle(s_4, s_2), (s_1, s_2)\rangle$	$s_0\ s_1\ s_2\ s_4\ s_6$	11
7.	$*, (s_4, s_2), (s_1, s_2), (s_2, s_1)$	$\langle(s_1, s_2), (s_2, s_1)\rangle$	$s_0\ s_2\ s_1\ s_2\ s_4\ s_6$	12
8.	$*, (s_4, s_2), (s_4, s_2), (s_4, s_2)$	$\langle(s_4, s_2), (s_4, s_2), (s_4, s_2)\rangle$	$s_0\ s_2\ s_4\ s_2\ s_4\ s_2\ s_4\ s_2\ s_4\ s_6$	13
9.	$*, (s_4, s_2), (s_1, s_6), (s_2, s_1)$	$\langle(s_1, s_6), (s_2, s_1)\rangle$	$s_0\ s_2\ s_1\ s_6$	13

Table 1. The result of K* applied to the graph G from Figure 6

– The asymptotic space complexity of K* was shown to be of $\mathcal{O}(m + n \log n + k)$.

We conclude that K* maintains the same asymptotic worst-case complexity as Eppstein's algorithm in terms of both runtime and space.

4 Experimental Evaluation: Route Planning

The original route planning problem (see, for instance, [13]) is to find an optimal (or sub-optimal) route from one point to another. KSP algorithms are used when alternative routes are required or some additional constraints on the routes are given. We now illustrate the scalability of K* by applying it to a benchmark route planning problem, based on a US road map model [5]. Due to space limitations the description of the experiments remains brief, for more detail we refer to [1], which also contains an experimental evaluation of K* when applied in the context of stochastic model checking.

Experiment 1 – New York City: The map we use here consists of 264 346 nodes and 733 846 edges. We applied Eppstein's algorithm and K* to the graph in order to find the first 1 000 optimal routes from a selected point in the city center to various targets. The

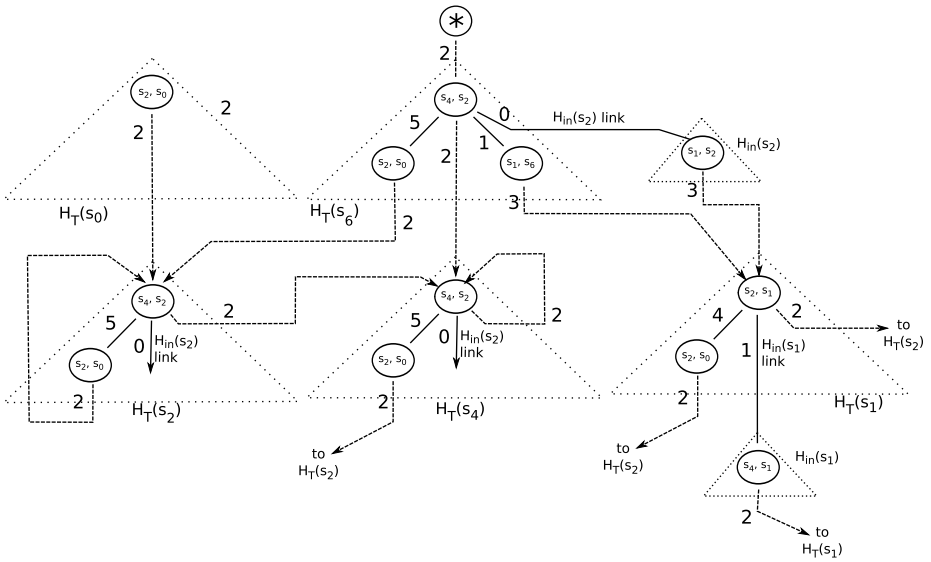

Fig. 12. The new path graph $\mathcal{P}(G)$ after the extension

4 targets we selected lie in different directions from the starting point with a shortest distance of approximately 50 km. As a heuristic in K* we used the *airline distance*, computing it according to the *cosine law*[1], and ensured that the resulting heuristic is admissible by underestimating the earth radius.

We determined the mean runtime and memory consumption required for each algorithm. The numbers we obtained indicate that with 200 sec and 5 MByte K* requires less than the half of the runtime and memory required by Eppstein's algorithm. Although the graph is not extremely large we notice that K* clearly outperforms Eppstein's algorithm.

Experiment 2 – Eastern USA: The map used here consists of 3 598 623 nodes and 8 778 114 edges and is hence more than 10 times larger than the map of New York City. We kept the same starting point as in the first experiment, however, we chose 4 different targets at approximately 200 km distance from the starting point.

We observed that Eppstein's algorithm failed to find a route. It crashed after approximately 2 000 seconds and 45 MByte of memory consumption with an out-of-memory exception. Note that the 45 MByte measured are the space used by the data structures of the algorithm. On the other hand, K* succeeded in providing all routes in all four cases. Its mean runtime was approximately 1 100 seconds. It required approximately 25 MB of memory on average.

[1] The cosine law computes the airline distance between two points as follows: $a = sin(lat_1) \cdot sin(lat_2) + cos(lat_1) \cdot cos(lat_2) \cdot cos(lon_2 - lon_1)$ and *Airline Distance* $= arccos(a) \cdot$ *Earth Radius*, where lon_i and lat_i are the longitude and latitude of ith point in the radian system.

5 Conclusion

We presented a new algorithm, called K*, for solving the KSP problem. K* performs on-the-fly and can be guided using heuristic estimates. We discussed its properties, including its asymptotic worst-case complexity of $\mathcal{O}(m + n \log n + k)$. We briefly derscribed experiments which show the superiority of K* over Eppstein's algorithm when applied to route planning problems. [1, 4, 2] report on experiments illustrating the application of K* to the generation of counterexamples in stochastic model checking. It is shown to perform much better than Eppstein's algorithm in this domain as well, thus improving results reported in [10].

Future research includes an analysis of the potential for parallelization of K* as well as an investigation of the applicability of heuristics guided search to other variants of the KSP problem.

Acknowlegement. The authors wish to thank Ulrik Brandes for discussions on an earlier version of this work.

References

1. Husain Aljazzar. *Directed Diagnostics of System Dependability Models*. PhD thesis, University of Konstanz, http://kops.ub.uni-konstanz.de/volltexte/2009/9188/, 2009.
2. Husain Aljazzar, Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. Directed and heuristic counterexample generation for probabilistic model checking - a comparative evaluation. In *Proc. of the First International Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems (QUOVADIS)*. IEEE Computer Society Press, 2010.
3. Husain Aljazzar and Stefan Leue. Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Trans. Softw. Eng.*, 2009.
4. Husain Aljazzar and Stefan Leue. Generation of counterexamples for model checking of markov decision processes. In *Proceedings of 6th International Conference on the Quantitative Evaluation of SysTems (QEST '09)*. IEEE Computer Society Press, 2009.
5. The Ninth DIMACS Implementation Challenge. The shortest path problem, 2006.
6. Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
7. David Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.
8. Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *32nd Annual Symposium on Foundations of Computer Science FOCS 1991*, pages 632–641. IEEE, 1991.
9. Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104(2):197–214, 1993.
10. Tingting Han and Joost-Pieter Katoen. Counterexamples in probabilistic model checking. In *TACAS'07, 13th International Conference*, 2007.
11. Víctor M. Jiménez and Andrés Marzal. A lazy version of eppstein's shortest paths algorithm. In *WEA 2003*, volume 2647 of *Lecture Notes in Computer Science*, pages 179–190. Springer, 2003.
12. Judea Pearl. *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addision–Wesley, 1986.
13. Peter Sanders and Dominik Schultes. Engineering fast route planning algorithms. In *WEA 2007*, volume 4525 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2007.