Artificial Intelligence 175 (2011) 2129–2154

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

K*: A heuristic search algorithm for finding the *k* shortest paths

Husain Aljazzar [1], Stefan Leue *

Department of Computer and Information Science, University of Konstanz, Box 67, 78457 Konstanz, Germany

ARTICLE INFO

Article history:
Received 20 July 2010
Received in revised form 14 July 2011
Accepted 14 July 2011
Available online 5 August 2011

Keywords:
k-Shortest-paths problem
K*
Heuristic search
On-the-fly search

ABSTRACT

We present a directed search algorithm, called K*, for finding the *k* shortest paths between a designated pair of vertices in a given directed weighted graph. K* has two advantages compared to current k-shortest-paths algorithms. First, K* operates 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 prove the correctness of K* and determine its asymptotic worst-case complexity when using a consistent heuristic to be the same as the state of the art, $\mathcal{O}(m + n \log n + k)$, 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 present an experimental evaluation of K* by applying it to route planning problems as well as counterexample generation for stochastic model checking. The experimental results illustrate that due to the use of heuristic, on-the-fly search K* can use less time and memory compared to the most efficient k-shortest-paths algorithms known so far.

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 **s** to a target vertex **t** in a directed weighted graph G for an arbitrary natural number k. Application domain examples for KSP problems include logistics, scheduling, sequence alignment, networking and many other application areas in which optimization problems need to be solved. A detailed discussion of applications of the KSP problem can be found in [1].

Our interest in the KSP problem stems from our interest in the generation of counterexamples for stochastic model checking [2]. In stochastic model checking we quantitatively reason about system dependability properties such as "the probability of a system failure within 1 year is at most 10%". If such a property is violated, we would like to return a set X of system executions that lead into a failure state so that their cumulated probability mass exceeds the probability threshold given in the property. We refer to such a set of executions as a counterexample, following the parlance used in model checking. Counterexamples are needed for system debugging, and hence we would like them to be informative. This is best achieved if the counterexample contains the executions with the highest probabilities amongst all system executions leading into system states. It is easy to see that enumerating these highest probability executions can be cast as a KSP problem [3–5]. Because of the characteristics of our application domain, we are interested in a variant of the KSP problem in which loops are allowed. We also assume that the number k is unknown at the beginning of the search, since we will not know in general how many paths are needed in order to exceed the probability bound specified in the property.

* Corresponding author.
E-mail addresses: husain.aljazzar@email.de (H. Aljazzar), Stefan.Leue@uni-konstanz.de (S. Leue).

[1] The work described in this paper was entirely performed while this author was with the University of Konstanz. Current permanent affiliation: Bosch Sicherheitssysteme GmbH, 85630 Grasbrunn, Germany.

0004-3702/$ – see front matter
doi:10.1016/j.artint.2011.07.003

More generally, we aim at enumerating the paths from **s** to **t**, including loops, in an order that is non-decreasing with respect to their length. The most advantageous algorithm for solving this problem with respect to worst-case complexity is Eppstein's algorithm. It has a complexity of $\mathcal{O}(m + n \log n + k)$ in terms of both runtime and space, where n is the number of vertices and m is the number of edges in the problem graph [1]. Classical KSP algorithms, including Eppstein's algorithm, require the complete problem graph G to be available when the search starts. They also require that initially an exhaustive search is performed on G in order to compute the shortest path tree, which characterizes the shortest path from every vertex to **t**. The need for a complete exploration is a major performance drawback in practice, in particular if G is large.

In this paper we present an algorithm called K* which addresses this shortcoming of the classical KSP algorithms. When using a consistent heuristic K* maintains an asymptotic worst-case runtime complexity of $\mathcal{O}(m + n \log n + k)$ in terms of both runtime and space. This means that it maintains the best known asymptotic worst-case complexity of the KSP problem as it is, for instance, also ensured by Eppstein's algorithm. On the other hand, the major two advantages of K* over other existing KSP algorithms are the following:

- K* works on-the-fly, which means that it avoids exploring and processing the entire problem graph G. It partially generates and processes portions of the graph as need arises. Solution paths are computed early on and made available as soon as they are computed. This on-the-fly feature is largely responsible for the superiority of K* over classical KSP algorithms in terms of performance and scalability.
- K* takes advantage of heuristic search, which leads to significant improvements in terms of both memory and runtime for many applications.

As our experimental evaluation shall illustrate, K* performs very favorably compared to the state-of-the-art classical KSP algorithms when applied to route planning and to the computation of counterexamples in stochastic model checking.

1.1. Related work

Several variants of the KSP problem have been studied in the literature. In some works the solution paths are restricted to be simple, which means that no state is repeated along any solution path. In other works solution paths must be disjoint in vertices or edges. In this paper we consider a variant of the KSP problem where loops are allowed in the solution paths and where the number k is unknown in advance.

In terms of asymptotic complexity, *Eppstein's algorithm* (EA) [1] is the most advantageous algorithm for solving this variant of the KSP problem. It maintains an asymptotic worst-case complexity of $\mathcal{O}(m + n \log n + k)$ with respect to both runtime and space, which is also the best complexity known so far for this problem. Martins and Santos [6] proposed another algorithm, called MSA. MSA does not meet the asymptotic complexity of EA. The authors, however, show in their experiments that MSA outperforms EA in practice. Jiménez and Marzal [7] presented another variant of Eppstein's algorithm, called the *recursive enumeration algorithm* (REA). Just like MSA, REA is inferior to EA with respect to the asymptotic complexity. The experimental evaluation of this algorithm, however, also shows that REA outperforms MSA and EA when applied to practical problems. In a later publication, Jiménez and Marzal [8] have presented a further optimization of Eppstein's algorithm, which is referred to as the *lazy variant of Eppstein's algorithm* (LVEA). It maintains the same asymptotic worst-case complexity as the original EA algorithm but improves its practical performance in terms of both runtime and space. The authors show that the runtime behavior of LVEA is at least comparable with, and in many cases even superior to that of REA.

All of the KSP algorithms discussed so far, namely MSA, EA, REA and LVEA, share the drawback that they need to exhaustively search the problem graph in order to compute a shortest path tree. This is a tree formed by the shortest paths from the start vertex to each vertex in the graph. We address this shortcoming by designing our proposed algorithm, K*, as an on-the-fly algorithm. On-the-fly approaches to search problems are a standard technique to handle large search spaces in many application domains, for instance in explicit-state model checking [2].

Galand and Perny [9] have presented a multi-objective extension of A*, called kA*, which reduces the multi-objective search problem to a single-objective k-shortest-path problem by a linear aggregation of the multiple search criteria. The algorithmic principle that kA* uses is equivalent to an A* search without duplicate detection. Whenever a vertex is visited via a new path, kA* treats the vertex as a new one. As a result, the search space of kA* is not the problem graph G anymore, but a graph which is potentially exponentially larger than G. Such an algorithmic approach is not expected to scale to the size of the graphs which K* is able to deal with. Furthermore, it does not yield a computational complexity which is comparable to the complexities provided by EA, LVEA and K*.

Pauls and Klein [10] have proposed an algorithm (PKA) for solving the k-best parses problem, which is a variant of the KSP problem known in the linguistics. PKA shares with K* the principle of using A* to take advantage of the performance enhancements offered by directed search. However, there are two main differences between K* and PKA. First, unlike K*, PKA requires a consistent (monotone) heuristic function. In particular, while K* can be applied with an arbitrary heuristic estimate and provides an optimal solution with an admissible one, the correctness of PKA hinges upon the use of a consistent heuristic. Second, contrary to K*, PKA requires the computation of a perfect heuristic (Viterbi outside scores) [10]. A perfect heuristic function gives the exact optimal cost to reach a target vertex from the current one. As the authors themselves state, the computation of a perfect heuristic is very expensive and represents a bottleneck in the PKA. These

points are strong limitations to the applicability of PKA to general large-scale KSP problems. We conclude that K* is a more efficient and less restrictive alternative algorithm for finding the k best parses.

The problem of computing counterexamples for stochastic model checking and how this problem can be interpreted as a variant of the KSP problem, is presented and discussed in detail in [11,3,5,12,4]. The use of K* in this setting has been discussed in [4,5,13]. This paper hence focuses on the description of K* and the discussion of its properties.

1.2. Structure of the paper

We present some preliminaries related to graph search in Section 2. In Section 3 we introduce the KSP problem and discuss existing algorithms for solving it. We introduce our algorithm K* in Section 4 and study its formal properties in Section 5. Section 6 presents an experimental evaluation using two case studies from different application domains. We conclude in Section 7.

2. Preliminaries

2.1. Notation

Let $G = (V, E)$ be a directed graph and $c : E \to \mathbb{R}_{\geqslant 0}$ be a length function mapping edges to non-negative real values. The length of a path $\pi = v_0 \to v_1 \to \cdots \to v_n$ is defined as the sum of the edge lengths, formally,

$$C(\pi) = \sum_{i=0}^{n-1} c(v_i, v_{i+1}).$$

If the graph is not clear from the context, then we make it explicit by using a subscript, such as in $C_G(\pi)$. We refer to the first vertex of π as $\mathit{first}(\pi)$, i.e., $\mathit{first}(\pi) = v_0$. For an arbitrary pair of vertices u and v, $\Pi(u, v)$ 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 selected vertices that we refer to as a source and a target, respectively. The graph G is *finite*, if the sets V and E are finite. G is called *locally finite*, if for each vertex v the number of outgoing edges is finite.

2.2. The shortest-path problem (SP)

The shortest-path problem (SP) is the problem of finding a path $\pi^* \in \Pi(\mathbf{s}, \mathbf{t})$ with $C(\pi^*) = C^*(\mathbf{s}, \mathbf{t})$. *Dijkstra's algorithm* is the most prominent algorithm for solving SP [14]. It finds the shortest path from the start vertex $\mathbf{s}$ to each vertex in G. The set of these paths forms a tree called the *shortest path tree* T. Dijkstra's algorithm stores vertices on the search front in a priority queue which is ordered according to a distance function d. Initially, the search queue contains only the start vertex $\mathbf{s}$ with $d(\mathbf{s}) = 0$. In each search iteration, the head of the search queue, say u, is removed from the queue and *expanded*. More precisely, for each successor vertex v of u, if v has not been explored before, then $d(v)$ is set to $d(u) + c(u, v)$ and v is put into the search queue. If v has been explored before, then $d(v)$ is set to the smaller distance of the old $d(v)$ and $d(u) + c(u, v)$. We distinguish between two types of *explored* vertices, namely *closed* and *open* vertices. Closed vertices are those which have been explored and expanded, whereas open vertices are those which have been explored but not yet expanded. We denote the sets of both vertex types as closed and open, respectively. Notice that open forms the search front that we referred to above. We call explored edges with closed destination vertices *inner edges*. Explored edges with open destination vertices are called *outer edges*.

For each explored vertex v, $d(v)$ is always equal to the length of some path from $\mathbf{s}$ to v that has been discovered so far. We refer to this path as the *solution base* of v. The set of these solution bases forms a *search tree* T. Dijkstra's algorithm ensures that for each closed vertex v it holds that $d(v) = C^*(\mathbf{s}, v)$ which means that the solution base of v is a shortest path from $\mathbf{s}$ to v. In other words, the search tree T is a shortest path tree for all closed vertices. Notice that a shortest $\mathbf{s}$–$\mathbf{t}$ path is found as soon as $\mathbf{t}$ is closed, which means that it is selected for expansion. In order to retrieve the selected shortest path to some vertex the structure of T needs to be maintained, and hence a link $T(v)$ is attached to each explored vertex v referring to the parent of v in T. The solution path can then be constructed by following these links from $\mathbf{t}$ upwards to $\mathbf{s}$.

2.3. On-the-fly search

Some search algorithms can be performed *on-the-fly*. This means that they can be applied to an implicit description of G, which is defined by a start vertex $\mathbf{s}$ and a function $\mathit{succ} : V \to \mathcal{P}(V)$ which returns for each vertex u the set of its successor vertices, i.e., $\mathit{succ}(u) = \{v \in V \mid (u, v) \in E\}$. The on-the-fly strategy enables the partial generation and processing of the problem graph as needed by the search algorithm. This strategy improves the performance and scalability of many search algorithms since it saves runtime effort by not processing the entire graph. It also saves memory since the search algorithm does not need to manage the entire graph in its data structures. The on-the-fly feature finally allows the algorithm to handle graphs which are either infinite, or finite but too large to fit into main memory.

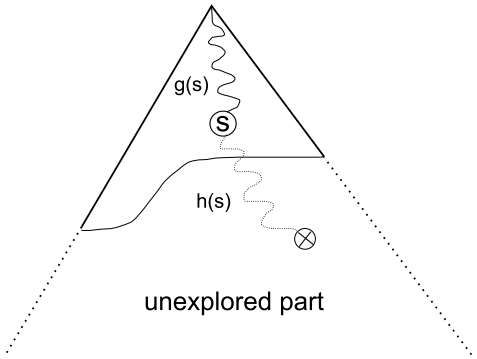


Fig. 1. Illustration of A* search. The evaluation function $f(s) = g(s) + h(s)$ determines the expansion order of vertices.

2.4. Directed search

Directed search algorithms are those search algorithms which can be guided using *heuristic estimates* in order to speed up the search process. These heuristic estimates usually exploit additional knowledge or intuition about the graph structure or the characteristics of the search target in order to guide the search algorithm so that it reaches a target faster. The guiding is accomplished by controlling the order in which open vertices are explored. Most directed search algorithms are structured so that they perform an on-the-fly search.

The most prominent directed search algorithm is A* [15] which is designed for solving SP. Its algorithmic principle is similar to that of Dijkstra's search. The main difference is that a *heuristic evaluation function* f is used instead of the length function d used by Dijkstra (see Section 2.2). The search queue open of A* is sorted using the heuristic evaluation function f, which is computed as the sum of two functions g and h:

$$f = g + h. \tag{1}$$

The function g gives the length of the solution base of a vertex, whereas h is the heuristic estimate of the distance from the considered vertex to the target. As illustrated in Fig. 1, $f(v)$ is then an estimate of the length of an **s**–**t** path through v.

The function g is equal to the distance function d in Dijkstra's algorithm. It is recursively defined as follows. Let u and v be a pair of vertices such that u is the parent of v in the search tree T. The value $g(v)$ is equal to $g(u) + c(u, v)$, where $g(\mathbf{s}) = 0$. The heuristic function h estimates the required cost to reach a target state. Notice that $h(v)$ must be computed based on information external to the graph since at the time of reaching v it is entirely unknown whether a path to the target exists at all. It is required that $h(\mathbf{t}) = 0$. A special variant of a heuristic is the trivial case when $h = 0$. In this case, the evaluation function f degrades to the function g and we obtain an uninformed algorithm equivalent to Dijkstra's algorithm.

The heuristic function h is called *admissible* if it is optimistic, i.e., if $h(v) \leqslant C^*(v, \mathbf{t})$ for any vertex v. An admissible heuristic guarantees the solution optimality of A*, which means that a shortest **s**–**t** path will be found. Moreover, h is called *monotone* or *consistent* if for each edge (u, v) in G it holds that $h(u) \leqslant c(u, v) + h(v)$. It can easily be proven that every monotone heuristic is also admissible.

While most directed search algorithms, including A*, have an exponential worst-case complexity in the number of vertices when an inconsistent heuristic estimate is used, which is due to the need to possibly re-open previously visited nodes, they possess a good average-case performance. In the case of a consistent heuristic estimate, A* has a worst-case complexity of $\mathcal{O}(m + n \log n)$, which is the same complexity as that of Dijkstra's algorithm [16]. Notice that this complexity applies, in particular, to the trivial heuristic estimate $h = 0$, as $h = 0$ is consistent.

3. k-Shortest-paths search

As a generalization of the SP problem, the *k-shortest-paths problem* (KSP) considers finding the k shortest paths from some start vertex **s** to the target vertex **t** for an arbitrary natural number k. Recall that we are interested in a variant of the KSP problem for which k does not need to be specified at the beginning of the computation, and for which loops are allowed in the solution paths. In other words, we aim at enumerating the **s**–**t** paths, including loops, in a non-decreasing order with respect to their length.

3.1. Eppstein's algorithm

The most prominent algorithm for solving this type of KSP problem has been proposed by Eppstein [1]. It is also the most advantageous algorithm described in the literature for solving this problem with respect to worst-case computational complexity. Eppstein's algorithm (EA) first applies Dijkstra's algorithm to a given problem graph G in reverse. The search starts at the target **t** and traces the edges back to their origin vertices. The result is a "reversed" shortest path tree T rooted

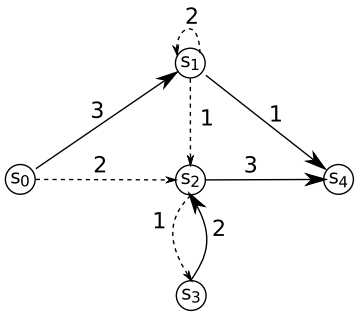


Fig. 2. Example graph, where the solid edges in the graph represent a reversed shortest path tree as computed by Eppstein's algorithm (EA). The dashed edges are the sidetrack edges.

in **t** which characterizes the shortest path from any vertex in G to **t**. Subsequently, a special data structure called *path graph* and denoted by $\mathcal{P}(G)$ is used to save all paths through G. Finally, the k shortest paths are delivered by applying a Dijkstra search to $\mathcal{P}(G)$.

A central concept in EA is that of a *sidetrack representation* of **s**–**t** paths. An edge (u, v) either belongs to the shortest path tree T, in which case we call it a *tree edge*; otherwise we call it a *sidetrack edge*.

Example 1. Fig. 2 shows a simple graph. Let s_0 be the start vertex and s_4 be the target vertex. The reversed shortest path tree is highlighted by solid line arrows. The edges which belong to this tree are *tree edges*. The other edges, which are drawn using dashed line arrows, are *sidetrack edges*.

For any **s**–**t** path π, we denote by $\xi(\pi)$ the subsequence of sidetrack edges which are taken in π. As Eppstein shows, π can be unambiguously described by the sequence $\xi(\pi)$. Formally, the mapping ξ is injective. Consequently, there is a partial injective inverse mapping χ so that $\chi(\xi(\pi)) = \pi$. The mapping χ establishes this unique way of completing the sequence of sidetrack edges $\xi(\pi)$ by adding the missing tree edges in order to obtain π.

Example 2. We consider again the graph from Fig. 2. Let π be the path $s_0\, s_1\, s_2\, s_4$. Notice that $\xi(\pi) = \langle (s_1, s_2) \rangle$. From the sidetrack sequence $\langle (s_1, s_2) \rangle$ we can obtain the preimage $\chi(\langle (s_1, s_2) \rangle) = \pi$ as follows. We start at the start vertex s_0. We add the tree edge (s_0, s_1). At this point we notice that s_1 is the origin vertex of the sidetrack edge (s_1, s_2). Hence, we add the sidetrack edge (s_1, s_2). We next add the tree edge (s_2, s_4) to π. This results in completing the path $\pi = s_0\, s_1\, s_2\, s_4$. The length of π is equal to 7, whereas the length of the shortest path $s_0\, s_1\, s_4$ is 4.

The notion of a sidetrack edge is interesting because selecting any $(u, v) \in G \setminus T$ will entail a certain detour compared to the shortest path. Sidetrack edges are hence closely related to the notion of opportunity cost since they represent the cost of taking an alternative and more expensive path compared to some given **s**–**t** path.

The path graph $\mathcal{P}(G)$ is a very complex data structure. It is very similar to the path graph used in K*, which we will explain in more detail in Section 4.3. For the time being it suffices to note that $\mathcal{P}(G)$ is a directed weighted graph. Its nodes represent sidetrack edges of G. The structure of $\mathcal{P}(G)$ ensures that the i-th shortest path in $\mathcal{P}(G)$ results in a sidetrack sequence which corresponds to the i-th shortest **s**–**t** path in G.

The computational effort of EA can be determined by considering the following three main steps in the algorithm:

(a) The first step is the exhaustive Dijkstra search on G in reverse in order to compute a shortest path tree. This step requires $\mathcal{O}(m + n \log n)$ runtime.
(b) As the second step, the construction of the path graph $\mathcal{P}(G)$ takes $\mathcal{O}(m + n \log n)$. Some optimizations based on data structure implementation techniques described in [17,18] can improve this step to $\mathcal{O}(m + n)$. These optimizations are, however, too complicated from a practical point of view [1,8].
(c) Extracting the k shortest paths from the path graph $\mathcal{P}(G)$ forms the third step in the algorithm. This step can be performed using Dijkstra's search on $\mathcal{P}(G)$, which requires a runtime of $\mathcal{O}(k \log k)$. Frederickson presented an efficient algorithm which allows to improve this step to $\mathcal{O}(k)$ [18].

This results in a total worst-case runtime complexity of $\mathcal{O}(m + n \log n + k)$. With the optimized construction of $\mathcal{P}(G)$, as mentioned in (b), the algorithm requires $\mathcal{O}(m + n + k)$ excluding the effort for computing the shortest path tree. The same asymptotic complexity can be derived for the space effort.

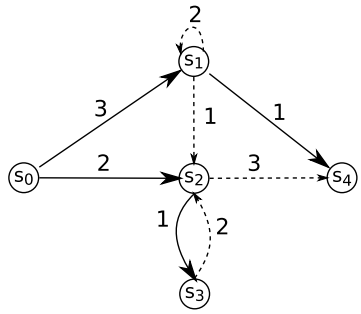


Fig. 3. The same graph from Fig. 2, where solid edges here represent the search tree as computed by A*. The dashed edges are the sidetrack edges.

3.2. A lazy variant of Eppstein's algorithm

Constructing the path graph in EA is an expensive step of the algorithm. A lazy variant of Eppstein's algorithm (LVEA) has been proposed which avoids this expensive step by constructing the path in a lazy manner [8]. The idea is to construct only those parts of the path graph which are necessary for selecting the sought k **s**–**t** paths. This lazy processing feature should not be confused with the on-the-fly feature that we use in K*. LVEA is not on-the-fly since it still requires the shortest path tree to be fully computed in advance using a exhaustive search on G.

LVEA maintains the same asymptotic worst-case complexity in terms of both runtime and space as the original EA algorithm. However, it has a significant performance advantage over EA in practice. Moreover, the runtime behavior of LVEA is comparable to, and in many cases even better than, REA. LVEA outperforms REA already for graphs consisting of a few thousands of vertices and edges. Typical graphs in most application domains which K* is designed for, such as model checking or route planing, consist of hundreds of thousands or even a few million vertices and edges. LVEA is hence considered to be the state-of-the-art and the most efficient KSP algorithm for problems of realistic size. For this reason, we compare K* only with LVEA in our experimental evaluation in Section 6.

4. The K* algorithm

The design of K* was inspired by Eppstein's algorithm (EA). Just like in EA, K* performs a shortest path search on G and uses a path graph structure $\mathcal{P}(G)$. In K* we take advantage of the lazy construction of $\mathcal{P}(G)$ as proposed in LVEA [8]. The path graph is searched using Dijkstra in order to determine the **s**–**t** paths in the form of sidetrack sequences. However, as mentioned earlier, K* is designed to perform on-the-fly and to be guided by a heuristic. The main design principles of K* are the following:

1. We apply A* to G in a forward manner, instead of the backwards Dijkstra search construction on G in EA. This enables an on-the-fly construction of the algorithm, as well as the use of a search heuristic.
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 completion of the search of G by A* and therefore prior to the complete exploration of G.

In order to accommodate this design we have to make some alterations to the structure of $\mathcal{P}(G)$ as it was originally defined in EA.

4.1. A search on G*

K* applies A* search to the problem graph G in order to compute a search tree T. Notice that A*, just like Dijkstra's algorithm, computes a search tree while searching for a shortest **s**–**t** path. This tree is formed by the father-node links that are stored while A* is working in order to be able to reconstruct the **s**–**t** path when a **t** node has been found. Sidetrack edges discovered during the A* search of G will immediately be inserted into the graph $\mathcal{P}(G)$, the structure of which will be explained in Section 4.3.

A* is applied to G in a forward manner, which yields a search tree T rooted at the start vertex **s**. The forward search strategy is necessary in order to be able to work on the implicit description of the problem graph G using the successor function *succ*. In the remainder of the paper, G is assumed to be a locally finite graph, if nothing else is explicitly stated. Notice that A* is correct and complete on locally finite graphs [15]. We follow a convention in the literature on directed search assuming that the cost of an infinite path is unbounded [15].

Example 3. If we apply K* to the graph from Fig. 2, then A* yields a search tree such as the one shown in Fig. 3. Tree edges are drawn with solid lines whereas sidetrack edges are drawn with dashed lines. Unlike the reversed shortest path tree shown in Fig. 2, the search tree of A* is a forward tree rooted at the start vertex s_0.

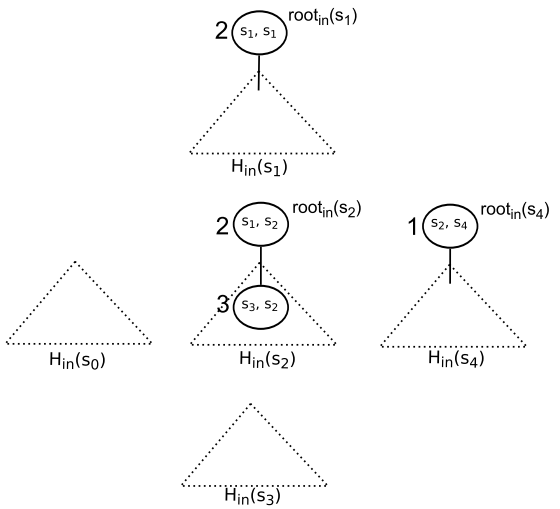


Fig. 4. The incoming heaps $H_{in}(.)$ derived from the graph shown in Fig. 3.

4.2. Detour cost

For an edge (u, v), the detour function $\delta(u, v)$ represents the cost disadvantage entailed by taking the detour edge (u, v) in comparison to the shortest **s**–**t** path via v. Neither the length of the shortest **s**–**t** path through v nor the length of the **s**–**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 an evaluation function f, which uses a heuristic estimate function h. 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., $f_u(v) = g(u) + c(u, v) + h(v)$. $\delta(u, v)$ can then be defined as:

$$\begin{aligned}\delta(u, v) &= f_u(v) - f(v)\\ &= g(u) + c(u, v) + h(v) - g(v) - h(v)\\ &= g(u) + c(u, v) - g(v).\end{aligned} \tag{2}$$

Notice that $\delta(u, v)$ delivers a precise detour metric, since the estimated h-value does not appear in the definition of $\delta(u, v)$.

4.3. Path graph structure

The path graph structure $\mathcal{P}(G)$ is rather complex. In principle, $\mathcal{P}(G)$ will be a directed graph, the vertices of which correspond to edges in the problem graph G. It is organized as a collection of interconnected heaps. 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 $\mathcal{P}(G)$. As we will show later on, the use of these heaps also plays a major role in maintaining the asymptotic complexity of K*, just as in EA and LVEA.

The incoming heap $H_{in}(v)$ contains a node for each incoming sidetrack edge of v which has been discovered by A* so far. 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 at most one child. We denote the root of $H_{in}(v)$ as $root_{in}(v)$.

Example 4. Fig. 4 illustrates the incoming heaps of the graph from Fig. 3. The numbers next to the heap nodes are the corresponding δ-values.

The tree heap $H_T(v)$, for an arbitrary vertex v, is built as follows. If v is the start vertex, which means that $v = \mathbf{s}$, then $H_T(\mathbf{s})$ is created as a fresh empty heap. The node $root_{in}(\mathbf{s})$ is then 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. 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 $H_T(u)$. However, for space efficiency we create only a cheap copy of $H_T(u)$. This is accomplished by creating new copies only of the heap nodes which lie on the updated path in $H_T(u)$. The remaining part of $H_T(u)$ is not copied. In other words, $root_{in}(v)$ is inserted into $H_T(u)$ in a non-destructive way such that the structure of $H_T(u)$ is preserved, cf. [1]. In the heap $H_T(v)$, one or two children may be attached to $root_{in}(v)$. In addition, $root_{in}(v)$ keeps its only child from $H_{in}(v)$. We denote the root of $H_T(v)$ by $R(v)$.

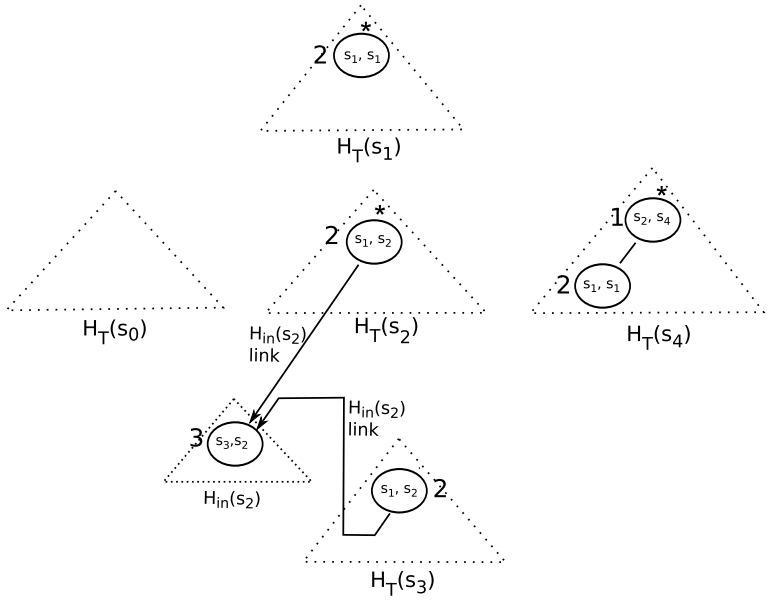


Fig. 5. The tree heaps $H_T(.)$ derived from the graph shown in Fig. 3.

Example 5. Fig. 5 illustrates the tree heaps of the graph from Fig. 3. The numbers attached to the heap nodes are the corresponding δ-values. We mark 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), see Fig. 4. The heap $H_T(s_3)$ is identical to the heap $H_T(s_2)$ since $H_{in}(s_3)$ is empty. 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.

We refer to the edges, which originate from the incoming heaps or the tree heaps, as *heap edges*. Similarly to [1], we can deduce the following lemma.

Lemma 1. *All nodes which are reachable from $R(v)$ via heap edges, for any vertex v, form a* 3*-ary heap that is ordered according to the δ-values. We call this heap the* graph heap *of v and denote it as $H_G(v)$.*

Proof. The nodes which are reachable from $R(v)$ via heap edges are those which are in $H_T(v)$, or in an incoming heap which is referred to by a node in $H_T(v)$. The tree heap $H_T(v)$ is formed by adding the roots of the incoming heaps of all vertices on the search tree path from the start vertex **s** to v into a binary heap structure. Each one of these root nodes has at most three children in total: at most two children in $H_T(v)$ in addition to at most a single child from the incoming heap. Any other node residing in an incoming heap has at most two children. Remember that each incoming heap is a binary heap with the restriction that the root node has a single child. The tree structure of $H_G(v)$ is an immediate result of the tree structure of $H_T(v)$ and the incoming heaps. Moreover, the heap characteristic of the tree heap ensures the heap order according to the δ-values along the edges in $H_T(v)$, whereas the heap characteristic of the incoming heaps ensures the heap order along all H_{in} edges. This implies in conclusion that $H_G(v)$ is a 3-ary heap which is ordered according to the δ-values. □

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 $R(u)$, which is the root node of $H_T(u)$. We call such pointers *cross edges*, whereas the pointers which arise from the heap structures are called *heap edges*, as mentioned before. Moreover, we add a special node $\Re$ to $\mathcal{P}(G)$ with a single outgoing cross edge to $R(\mathbf{t})$.

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

$$\Delta\big(n, n'\big) = \begin{cases} \delta(e') - \delta(e) & \text{if } (n, n') \text{ is a heap edge, and} \\ \delta(e') & \text{if } (n, n') \text{ is a cross edge.} \end{cases} \tag{3}$$

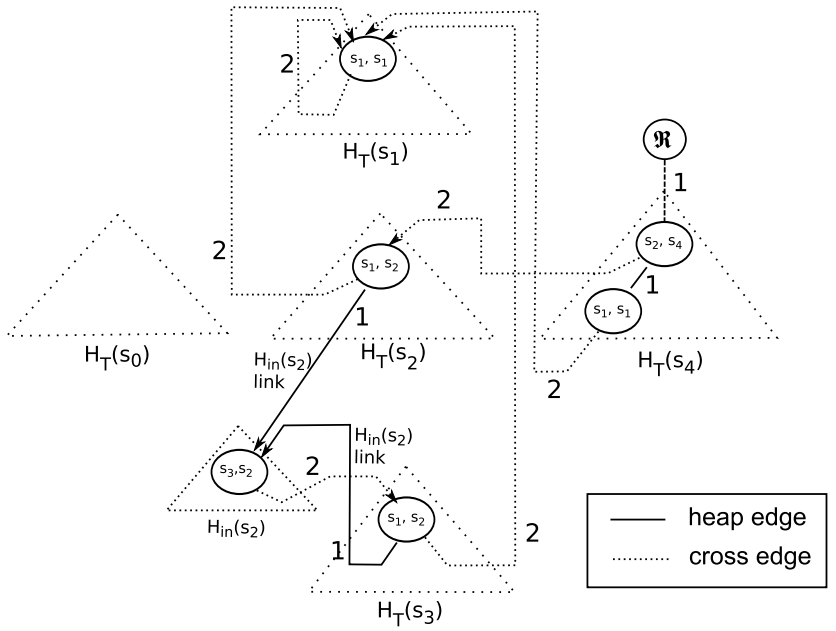


Fig. 6. The path graph $\mathcal{P}(G)$ computed by K* from the graph shown in Fig. 3.

Lemma 1 implies that the heap order according to the δ-values is maintained along any heap edge in $\mathcal{P}(G)$. This heap order implies that $\Delta(n, n')$ is not negative for any heap edge (n, n'). Δ is hence not negative, i.e., $\Delta(n, n') \geqslant 0$, for any edge (n, n') in $\mathcal{P}(G)$. The cost of a path σ, i.e., $C_{\mathcal{P}(G)}(\sigma)$, is equal to $\sum_{e \in \sigma} \Delta(e)$.

Example 6. Fig. 6 shows the final path graph obtained from the graph from Fig. 3. Notice that the weights are now assigned to the edges. These weights are computed according to the weighting function Δ.

In the remainder of this section we will illustrate the characteristics of the path graph structure which are relevant for finding the shortest **s**–**t** paths. We will discuss a few observations, some of which have also been discussed in [1]. We show to what extent these observations still apply to our adapted path graph structure. Notice that the path graph structure presented here differs from the original structure used in EA or LVEA in some aspects. These differences are necessary in order to enable a forward search on the problem graph instead of the backwards search used in EA and LVEA.

The first observation is that $\mathcal{P}(G)$ is a directed weighted graph. Each node in $\mathcal{P}(G)$ carries a sidetrack edge from G. The use of binary heaps in the construction of $\mathcal{P}(G)$ benefits from the following two properties. First, an arbitrary node in $\mathcal{P}(G)$ has at most four outgoing edges. Exactly one of these edges is a cross edge, whereas the remaining edges are heap edges. Second, the weight function Δ is not negative. As it will become clear in Section 5, these properties are essential for the proof of correctness and the determination of the complexity of K*.

The second observation is the existence of a one-to-one correspondence between **s**–**t** paths in G and paths in $\mathcal{P}(G)$ which start at $\mathfrak{R}$. We explain this point in detail and prove it formally in Lemma 3. We conclude this point with Corollary 1. In the sequel, when we refer to paths in $\mathcal{P}(G)$, we mean paths in $\mathcal{P}(G)$ which start at $\mathfrak{R}$.

An arbitrary path $\sigma = n_0 \to \cdots \to n_r$ in $\mathcal{P}(G)$ can be interpreted as a recipe for constructing a unique **s**–**t** path. Each cross edge (n_i, n_{i+1}) in σ represents the selection of the sidetrack edge associated with 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 define a procedure for deriving from σ a sequence of edges $seq(\sigma)$. It also constructs the corresponding **s**–**t** path π from $seq(\sigma)$. This procedure can be carried out in linear time in the length of σ. The procedure for constructing $seq(\sigma)$ is as follows. In the beginning, let $seq(\sigma)$ be an empty sequence. First, we add to $seq(\sigma)$ the edge associated with the last node of σ, i.e., n_r. We then iterate over the edges of σ in a backwards manner starting at the last edge. For each cross edge (n_i, n_{i+1}) in σ, with $n_i \neq \mathfrak{R}$, we add to $seq(\sigma)$ the edge associated with n_i.

The corresponding **s**–**t** path π is constructed from $seq(\sigma)$ in a backwards manner from **t** to **s**. This means that we start at t and proceed as follows. We repeatedly prepend to π the tree edge of the current first vertex of π. If the current first vertex of π is equal to the destination vertex of the next sidetrack edge in $seq(\sigma)$, then we prepend the sidetrack edge to π. In this procedure the sidetrack edges from $seq(\sigma)$ are consumed from the tail to the head. We keep doing so until we reach the start vertex **s**. The following example demonstrates this procedure.

Example 7. We consider the path graph shown in Fig. 6. Let σ be the path $\Re \to (s_2, s_4) \to (s_1, s_2) \to (s_3, s_2)$. Following the description given above we derive that $seq(\sigma) = \langle (s_3, s_2),\ (s_2, s_4) \rangle$. We construct the corresponding s_0–s_4 path as follows. We start at s_4. The next sidetrack edge in $seq(\sigma)$ is (s_2, s_4). We see that we reached the destination vertex of this sidetrack edge, namely s_4. Hence, we prepend this sidetrack edge to get the path $s_2\, s_4$. Again we see that we reached the destination vertex of the next sidetrack edge, namely s_2. Hence, we prepend this sidetrack edge to our path to obtain the path $s_3\, s_2\, s_4$. We now have consumed all sidetrack edges and we hence keep prepending tree edges until we reach s_0 and finally obtain the path $s_0\, s_2\, s_3\, s_2\, s_4$.

We will show that the structure of $\mathcal{P}(G)$ ensures that the sidetrack sequence $seq(\sigma)$ represents a valid **s**–**t** path. We will show furthermore that two different paths in $\mathcal{P}(G)$ induce two different sidetrack sequences and, consequently, two different **s**–**t** paths in G. Altogether, we get a one-to-one correspondence between **s**–**t** paths in G and paths in $\mathcal{P}(G)$. Before we formally prove this fact in Lemma 3, we prove the following lemma which we will use in the proof of Lemma 3.

Lemma 2. *Let n be a node in the graph heap $H_G(w)$ for some vertex w. Let (u, v) be the edge associated with n. There is a path in the search tree T from v to w.*

Proof. Remember that $H_G(w)$ is constructed from the incoming and tree heaps of the vertices on the tree path from the start vertex **s** to w. The node n obviously originates from the incoming heap $H_{in}(v)$. This means that v lies on the search tree path from **s** to w. In particular, there is a path in T from v to w. □

Now we prove the existence of a correspondence between **s**–**t** paths in G and paths in $\mathcal{P}(G)$. We do this by proving that the mapping $p = \chi \circ seq$ is a well-defined bijective mapping. The well-definedness of p ensures that $seq(\sigma)$ is a sidetrack edge sequence which represents a valid **s**–**t** path in G. This **s**–**t** path can be obtained by completing $seq(\sigma)$ from **t** up to **s** with the possibly missing tree edges up to **s**. The bijectivity of p means that each **s**–**t** path in G is represented by exactly one path in $\mathcal{P}(G)$.

Lemma 3. *The mapping $p = \chi \circ seq$ from paths in $\mathcal{P}(G)$ starting at $\Re$ onto* **s**–**t** *paths in G is* (a) *well-defined and* (b) *bijective.*

Proof. We first prove that p is well-defined. Afterwards we show that p is injective and surjective.

Well-definedness: Let σ be a path in $\mathcal{P}(G)$ starting at $\Re$. We show that there is a path $\pi \in \Pi(\mathbf{s}, \mathbf{t})$ such that $\xi(\pi) = seq(\sigma)$. This implies that $seq(\sigma) \in \xi(\Pi(\mathbf{s}, \mathbf{t}))$. Since χ is the inverse mapping of ξ, $\chi(seq(\sigma))$ is defined. This means that p is well-defined.

To establish this point we begin with the single vertex **t**, i.e., $\pi = \mathbf{t}$. Let (u, v) be the last edge in $seq(\sigma)$. Then, (u, v) is contained in $H_G(\mathbf{t})$ since it could otherwise not be the last element in $seq(\sigma)$. From this observation we conclude that there is a path in the search tree T from v to **t** (see Lemma 2). Hence, there is a unique way of prepending tree edges to π backwards to v, i.e., until $first(\pi) = v$. We next prepend the edge (u, v) to π. Furthermore, for each successive pair of edges (q, w) and (u, v) in $seq(\sigma)$, it must be that (q, w) belongs to $H_G(u)$. This means that there is a path in T from w to u (see Lemma 2). We prepend the edges of this tree paths followed by the sidetrack edge (q, w). We then repeat this step until all edges from $seq(\sigma)$ are handled. Afterwards, we repeatedly prepend to π the tree edge of $first(\pi)$ backwards to the start vertex **s**, i.e., $first(\pi) = \mathbf{s}$. As a result, the constructed path π runs from **s** to **t** using no sidetrack edges except for the ones from $seq(\sigma)$. This means that the result is an **s**–**t** path π such that $\xi(\pi) = seq(\sigma)$.

Injectivity: We show that p is injective. Let σ and σ' be two different paths in $\mathcal{P}(G)$ starting at $\Re$. Since χ is injective, it is sufficient to show that $seq(\sigma) \neq seq(\sigma')$. The idea is to show that it is not possible that the tails of σ and σ', i.e., the parts following their common prefix, induce the same sequence of sidetrack edges. Let m be the last node in the common prefix of σ and σ'. We consider the following cases:

1. One of the paths ends at m. Without loss of generality, let σ' end at m and let σ have a postfix after m. Let n, be the next node in σ after m. If (m, n) is a cross edge, then it leads to a heap $H_G(q)$ from which a sidetrack edge will be added to $seq(\sigma)$. Hence, we get $seq(\sigma) \neq seq(\sigma')$. If (m, n) is a heap edge, then σ ends or leaves the graph heap, which contains m, at another node than m because heaps are acyclic. Consequently, $seq(\sigma) \neq seq(\sigma')$ also holds in this case.
2. Neither σ nor σ' ends at m. This implies that σ and σ' branch away from each other with two different edges, say (m, n) and (m, n'). Note that this case cannot occur if $m = \Re$, since $\Re$ has exactly one outgoing edge. Furthermore, it is not possible that both (m, n) and (m, n') are cross edges because any node in $\mathcal{P}(G)$ has, by construction, at most one outgoing cross edge. We hence need to consider the following two cases:
   (a) Both edges (m, n) and (m, n') are heap edges. In this case, since heaps are acyclic the last nodes touched by σ and σ' before the end or the next cross edge must differ from each other. It then holds that $seq(\sigma) \neq seq(\sigma')$.
   (b) Next assume that one edge, say (m, n), is a cross edge and the other is a heap edge. Again, since heaps are acyclic, the last node touched by σ' before either its end or the next cross edge is different from m. The sidetrack associated with m will then be the next sidetrack edge in $seq(\sigma)$ but not in $seq(\sigma')$. This means that $seq(\sigma) \neq seq(\sigma')$.

Altogether, we conclude that $seq(\sigma) \neq seq(\sigma')$. This means that p is injective.

Surjectivity: We now show that p is surjective. Let π be an **s**–**t** path in G. We need to determine a path σ starting at $\Re$ such that $p(\sigma) = \pi$, which means that we need to determine a path σ with $seq(\sigma) = \xi(\pi)$. Remember that $\mathcal{P}(G)$ is constructed incrementally. In order to determine the sought path σ we need to assume that π has been completely explored by A*, which implies that all sidetrack edges taken by π are already included in $\mathcal{P}(G)$. The completeness of A* on locally finite graphs [15] ensures that this will happen at some point in the search process.

If $\xi(\pi)$ is empty, then $\sigma = \Re$ is the sought path. When $\xi(\pi)$ consists of one sidetrack edge (u, v), we then know that there is a path in T from v to **t** since π leads to **t**. We then know that (u, v) belongs to $H_G(\mathbf{t})$. This simply means that a path σ must exist inside $H_G(\mathbf{t})$ between $R(\mathbf{t})$ and (u, v). It then holds that $seq(\sigma) = \xi(\pi)$.

If $\xi(\pi) = \langle e_1, \ldots, e_l \rangle$ with $l > 1$, then we can assume, using an inductive argument over l, that $\mathcal{P}(G)$ contains a path σ_1 from $R(\mathbf{t})$ to the node corresponding to e_2 such that $seq(\sigma_1) = \langle e_2, \ldots, e_l \rangle$. We write e_1 and e_2 as $e_1 = (q, w)$ and $e_2 = (u, v)$. By construction, there is a path in T from w to u. Hence, (q, w) belongs to the heap $H_G(u)$. This means that there is a path σ_2 inside $H_G(u)$ from $R(u)$ to (q, w). Note that $seq(\sigma_2) = \langle e_1 \rangle$. Now, let $\sigma = \sigma_1\sigma_2$ be the path obtained by concatenating σ_1 and σ_2. Then it is easy to show that $seq(\sigma) = \xi(\pi)$.

As a consequence, for any **s**–**t** path in G, there is a path σ in $\mathcal{P}(G)$ starting at $\Re$ with $seq(\sigma) = \xi(\pi)$. This implies that $\chi(seq(\sigma)) = \chi(\xi(\pi))$, which means that $p(\sigma) = \pi$. Thus, p is surjective. □

From Lemma 3 we easily derive the following corollary, from which we can conclude that we can use paths in $\mathcal{P}(G)$ as solution paths in G.

Corollary 1. *There is a one-to-one correspondence between paths in $\mathcal{P}(G)$ starting at $\Re$ and* **s**–**t** *paths in G.*

The third observation is the correlation between the length of a path in $\mathcal{P}(G)$ and the corresponding **s**–**t** path in G. For a path σ in $\mathcal{P}(G)$, we show that the cost of σ is equal to the distance penalty of $p(\sigma)$ compared to the shortest **s**–**t** path in G. This implies that shorter $\mathcal{P}(G)$ paths lead to shorter **s**–**t** paths. This property enables computing shortest **s**–**t** paths using shortest-path search on $\mathcal{P}(G)$ starting at $\Re$. We state this property in the following two lemmas.

Lemma 4. *Let σ be a path in $\mathcal{P}(G)$ starting at $\Re$. It holds that $C_{\mathcal{P}(G)}(\sigma) = \sum_{e \in seq(\sigma)} \delta(e)$.*

Proof. We consider the subsequences $\sigma_0, \ldots, \sigma_r$ which we obtain by splitting σ at cross edges. More precisely, each σ_i starts with a cross edge and continues with only heap edges. Then, for each σ_i it holds that $\sum_{e \in \sigma_i} \Delta(e) = \delta(e_i)$, where e_i is the edge associated to the last node of σ_i,

$$C_{\mathcal{P}(G)}(\sigma) = \sum_{e \in \sigma} \Delta(e) = \sum_{i=0}^{r} \delta(e_i).$$

Note that $seq(\sigma)$ is equal to the edge sequence $\langle e_r, \ldots, e_0 \rangle$. It then holds that:

$$C_{\mathcal{P}(G)}(\sigma) = \sum_{e \in seq(\sigma)} \delta(e). \qquad \square$$

Lemma 5. *Let σ be a path in $\mathcal{P}(G)$ starting at $\Re$. If h is admissible, then it holds that*

$$C\big(p(\sigma)\big) = C^*(\mathbf{s}, \mathbf{t}) + C_{\mathcal{P}(G)}(\sigma).$$

Proof. Let $\pi = p(\sigma)$. We write π as $\pi = v_0 \to \cdots \to v_n$ with $v_0 = \mathbf{s}$ and $v_n = \mathbf{t}$. Since $\delta(e) = 0$ for tree edges, we conclude that

$$\sum_{e \in \xi(\pi)} \delta(e) = \sum_{e \in \pi} \delta(e).$$

We next consider that

$$\begin{aligned} \sum_{e \in \xi(\pi)} \delta(e) &= \sum_{e \in \pi} \delta(e) \\ &= \sum_{i=0}^{n-1} \delta(v_i, v_{i+1}) \end{aligned}$$

$$= \sum_{i=0}^{n-1} g(v_i) + c(v_i, v_{i+1}) - g(v_{i+1})$$
$$= g(v_0) + \sum_{i=0}^{n-1} c(v_i, v_{i+1}) - g(v_n)$$
$$= g(\mathbf{s}) + \sum_{i=0}^{n-1} c(v_i, v_{i+1}) - g(\mathbf{t}).$$

Presuming that h is admissible, it holds that $g(\mathbf{t}) = C^*(\mathbf{s}, \mathbf{t})$. Further, it holds that $g(\mathbf{s}) = 0$. We then obtain that

$$\sum_{e \in \xi(\pi)} \delta(e) = \sum_{i=0}^{n-1} c(v_i, v_{i+1}) - C^*(\mathbf{s}, \mathbf{t}).$$

Note that $\xi(\pi) = seq(\sigma)$ and $\sum_{i=0}^{n-1} c(v_i, v_{i+1}) = C(\pi)$. We may therefore derive that

$$\sum_{e \in seq(\sigma)} \delta(e) = C(\pi) - C^*(\mathbf{s}, \mathbf{t}) \quad \Rightarrow \quad C(\pi) = C^*(\mathbf{s}, \mathbf{t}) + \sum_{e \in seq(\sigma)} \delta(e).$$

Using Lemma 4 we conclude that

$$C(\pi) = C^*(\mathbf{s}, \mathbf{t}) + C_{\mathcal{P}(G)}(\sigma). \qquad \square$$

4.4. The algorithmic structure of K*

The algorithmic principle of K* is as follows. We execute A* to search in G and Dijkstra to search in $\mathcal{P}(G)$ in an interleaved fashion as follows. First, we run A* on G until the target vertex $\mathbf{t}$ is selected for expansion. Then, we run Dijkstra on the available portion of $\mathcal{P}(G)$. Each node expanded by Dijkstra represents a solution path. More precisely, the $\mathcal{P}(G)$ path σ via which Dijkstra reached that node is produced as a solution. The **s**–**t** path can be constructed from σ in linear time by computing the sidetrack edge sequence $seq(\sigma)$ and then **s**–**t** path from it. If Dijkstra finds k shortest paths, then K* terminates successfully. Otherwise, A* is resumed to explore a bigger portion of G. This leads to a grown $\mathcal{P}(G)$, on which the Dijkstra search is then resumed. We repeat this process until Dijkstra succeeds in finding k shortest paths.

Algorithm 1 contains the pseudocode of K*. The code from line 8 to line 25 forms the main loop of K*. The loop terminates when the search queues of both algorithms A* and Dijkstra are empty. The lines before line 8 perform some preparation tasks. After some initialization statements, A* is started at line 5 until $\mathbf{t}$ is selected for expansion, in which case a shortest **s**–**t** path has been found. If $\mathbf{t}$ is not reachable, then the algorithm terminates without a solution. Notice that it would not terminate on an infinite graph. Otherwise, the algorithm adds $\Re$, which is the designated root of $\mathcal{P}(G)$, into the search queue of the Dijkstra algorithm. K* then enters its main iteration loop.

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 $g(\mathbf{t}) + d \leqslant f(u)$ (see 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*. Remember that d is the distance function used in Dijkstra's algorithm as defined in Section 2.2. 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 (see line 14). How long we let A* run is a trade off. If we run it only for a small number of steps, then 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 and therefore need to limit the number of switches. This overhead is caused by the fact that after resuming A* at line 14, the structure of $\mathcal{P}(G)$ may change. We hence need to refresh $\mathcal{P}(G)$ at line 15, as we will extensively discuss in Section 4.5. This requires a subsequent inspection of the status of Dijkstra's search. We have to ensure that Dijkstra's search maintains a consistent state after the changes in $\mathcal{P}(G)$. K* stipulates a condition which governs the decision of when to stop A*, which we refer to as the *extension condition*. In order to maintain the same runtime complexity as EA and LVEA, we have to define the extension condition so that A* runs until the number of expanded vertices and the number of inner edges are doubled or G has been searched completely. We will discuss this issue in more detail later on in this section. As a useful feature, K* allows the definition of other extension conditions which may be more efficient in practice. In our experiments in Section 6, we define the extension condition so that the number of explored 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* has not yet finished exploring the entire graph G. Once A* has explored the entire graph G (see **if**-statement at line 9), the scheduling mechanism is disabled and henceforth only Dijkstra will be executed.

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. In other words, every time a node is generated, it

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 ℜ 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 status.
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.
```

is considered as a new node. This strategy is justified by the observation that a **s**–**t** path may take the same edge several times. Line 24 adds the next **s**–**t** path into the result set $\mathcal{R}$. This is done by constructing the sidetrack sequence $seq(\sigma)$ from the path σ through which Dijkstra reached the node n which has just been expanded. The algorithm terminates when k sidetrack sequences have been added into $\mathcal{R}$ (see line 25).

4.5. Interdependency of A* and Dijkstra search

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 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 status of Dijkstra's search, see line 15 in Algorithm 1. In general, A* may add new nodes, change the δ-values of existing nodes, or even remove nodes. A* may also significantly change the search tree T which will, in the worst case, destroy the structure of all H_T heaps. These changes may lead to a global restructuring or even a reconstructing of $\mathcal{P}(G)$ from scratch. In the worst case, this may make the previous Dijkstra search on $\mathcal{P}(G)$ useless, so that we will have to restart Dijkstra from scratch.

If the used heuristic estimate is admissible, we find ourselves in a better situation. We may still need to reconstruct $\mathcal{P}(G)$, but we will show that this reconstruction does not interfere with the correctness of the Dijkstra search on $\mathcal{P}(G)$. In other words, we do not loose the results so far obtained by the Dijkstra search.

In the case of a consistent heuristic estimate we even do not need to reconstruct or restructure $\mathcal{P}(G)$. If h is consistent, then the search tree of A* is a shortest path tree for all expanded vertices. Consequently, the g-values of the expanded vertices do not change. This implies that the δ-values of all inner edges will never change. The tree edges of expanded vertices will never change either. Hence, updating the δ-values, heaping-up, heaping-down or removing nodes do not entail any changes in $\mathcal{P}(G)$. Only the addition of new nodes leads to changes in $\mathcal{P}(G)$. Consequently, reconstructing $\mathcal{P}(G)$ or globally restructuring it is not required in this case.

In the remainder of this section, we first show that the correctness of the Dijkstra search on $\mathcal{P}(G)$ is maintained in the case of an admissible heuristic estimate. After this, we will show that the changes in $\mathcal{P}(G)$ can interfere with the completeness of Dijkstra's search, no matter whether the heuristic is admissible or even consistent. Hence, we will suggest a mechanism to ensure that completeness is maintained.

We focus next on the correctness of the Dijkstra search on $\mathcal{P}(G)$ in the case of an admissible heuristic estimate. First, we state that, if h is admissible, then the nodes in the explored section of $\mathcal{P}(G)$ will not change their δ-values.

Lemma 6. *Let n be an arbitrary node in $\mathcal{P}(G)$ and let (u, v) be the edge associated with n. If h is admissible, then the value of $\delta(u, v)$ will never change after n has been explored by Dijkstra.*

Proof. Let σ be the path in $\mathcal{P}(G)$ via which n was reached. Let π be the **s**–**t** path which corresponds to σ. Note that n is the last node in σ, which means that (u, v) is the first sidetrack edge in π. Thus, π is of the form $\pi = \pi_u \to u \to v \to \pi_{tail}$ with some path π_{tail} from v to **t**. Due to Lemma 5 it holds that $C(\pi) = C^*(\mathbf{s}, \mathbf{t}) + C_{\mathcal{P}(G)}(\sigma)$. This implies that $C(\pi) = C^*(\mathbf{s}, \mathbf{t}) + d(n)$.

The value of $\delta(u, v)$ can only be changed if u or v are relaxed, which means that either $g(u)$ or $g(v)$ is reduced. We prove the claim separately for each of these two cases.

Case 1: We first consider the case that v is relaxed, i.e., $g(v)$ is reduced. This means that A* detected a new tree path π_v to v which is shorter than the old one. Let q be the vertex from π_v which was in the search queue of A* when n was explored. The scheduling mechanism of K* ensures that $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant f(q)$. Let π' be the path $\pi_v \to \pi_{tail}$. Notice that $q \in \pi'$ which implies that $f(q) \leqslant C(\pi')$, since h is admissible. This implies that $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant f(q) \leqslant C(\pi')$. On the other hand, π_v is the new tree path to v. Thus, it is shorter than $\pi_u \to u \to v$. This implies that $C(\pi') < C(\pi)$. Together we get $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant C(\pi') < C(\pi)$. Notice that $C^*(\mathbf{s}, \mathbf{t}) + d(n) = C(\pi)$, which implies the contradiction $C(\pi) \leqslant C(\pi') < C(\pi)$. We conclude that the claim holds in this case.

Case 2: We now consider the case that u is relaxed, i.e., $g(u)$ is reduced. This means that A* detected a new tree path π'_u to u which is shorter than the old one. Let q be the vertex from π'_u which was in the search queue of A* when n was explored. The scheduling mechanism of K* ensures that $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant f(q)$. Let π' be the path $\pi'_u \to v \to \pi_{tail}$. Notice that $q \in \pi'$, which implies that $f(q) \leqslant C(\pi')$ since h is admissible. This implies that $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant f(q) \leqslant C(\pi')$. On the other hand, π'_u is the new tree path to u which is shorter than the old path π_u. This implies that $C(\pi') < C(\pi)$. Together we obtain $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant C(\pi') < C(\pi)$. Notice that $C^*(\mathbf{s}, \mathbf{t}) + d(n) = C(\pi)$ which implies the contradiction $C(\pi) \leqslant C(\pi') < C(\pi)$. This implies that the claim also holds in this case. □

From Lemma 6 we can deduce the following corollary.

Corollary 2. *Let n be an arbitrary node in $\mathcal{P}(G)$. If h is admissible, then n will never be removed from $\mathcal{P}(G)$ after n has been explored by Dijkstra.*

Proof. The node n can only be removed, if the associated edge becomes a tree edge during the A* search process. This can happen if the delta value of the associated edge is decreased from an arbitrary positive number to 0. However, this cannot happen due to Lemma 6. □

Furthermore, we prove that the structure of the explored section of $\mathcal{P}(G)$ will not change.

Lemma 7. *Let n be an arbitrary node in $\mathcal{P}(G)$. If h is admissible, then n will never change its position after it has been explored by Dijkstra.*

Proof. Let (u, v) be the edge which is associated with n. Lemma 6 ensures that $\delta(u, v)$ will never be changed after n has been explored. This means that n will not be heaped-up or heaped-down. Thus, the position of n could be altered by changes which are applied to another node m. Let (r, w) be the edge associated with m. We can exclude the case that m is located above n in the heap structure. This is because m would be explored before n and $\delta(r, w)$ will never change afterwards. Consequently, we only need to be concerned with the case that m is heaped-up from below to a position above n. This can happen in three situations, which we separately discuss in the following. Let σ be the shortest path which leads to m in $\mathcal{P}(G)$. Moreover, let π be the **s**–**t** path which corresponds to σ.

Case 1: In this case m is added into $\mathcal{P}(G)$ when A* expands r and discovers (r, w) as a sidetrack edge. Let q be the vertex of the search tree path π_r which is in the A* search queue when n is explored by Dijkstra. Such a vertex must exist, since otherwise r would be unreachable. The scheduling mechanism of K* ensures that $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant f(q)$. Note that m is the last node in σ which means that (r, w) is the first sidetrack edge in π. Thus, it holds that $q \in \pi$. Since h is admissible, then $f(q) \leqslant C(\pi)$. This implies that $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant f(q) \leqslant C(\pi)$. It follows that $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant C^*(\mathbf{s}, \mathbf{t}) + d(m)$. It immediately follows that $d(n) \leqslant d(m)$. This means that m will not be heaped-up above n.

Case 2: In this case m is added into $\mathcal{P}(G)$ when A* discovers a new edge (r', w) which is selected as a new tree edge of w instead of (r, w). Like in the first case, we argue for the existence of a vertex q of the search tree path $\pi_{r'}$ which is in the A* search queue when n is explored by Dijkstra. The scheduling mechanism of K* ensures that $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant f(q)$. Note that m is the last node in σ which means that (r, w) is the first sidetrack edge in π. This entails that $\pi = \pi_r \to w \to \pi_{tail}$ for some path π_{tail} in G from w to **t**. Let $\pi' = \pi_{r'} \to w \to \pi_{tail}$. Note that $\pi_{r'} \to w$ is the new tree path to w. Hence, $\pi_{r'} \to w$ is shorter than $\pi_r \to w$, which means that $C(\pi') < C(\pi)$. Moreover, note that $q \in \pi'$. Since h is admissible, then $f(q) \leqslant C(\pi') < C(\pi)$. This means that $C^*(\mathbf{s}, \mathbf{t}) + d(n) \leqslant f(q) < C(\pi)$. From this we deduce that $C^*(\mathbf{s}, \mathbf{t}) + d(n) < C^*(\mathbf{s}, \mathbf{t}) + d(m)$. This immediately implies that $d(n) < d(m)$. We conclude that m will not be heaped-up above n.

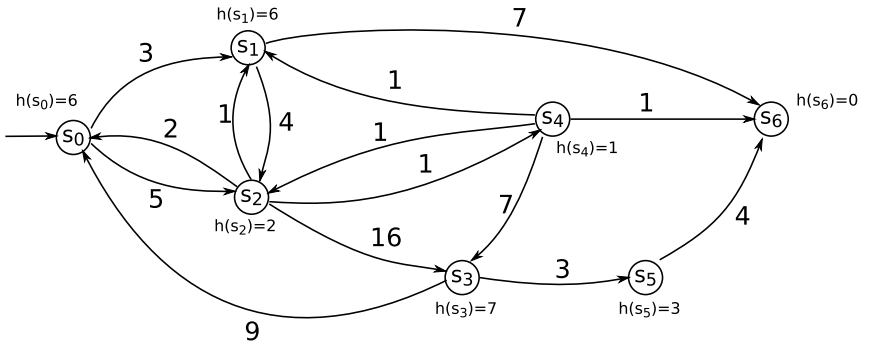


Fig. 7. The problem graph G which is considered in the example explained in Section 4.6.

Case 3: In this case m exists already in $\mathcal{P}(G)$. It is heaped-up because the value of $\delta(r, w)$ is reduced when A* discovers a new shorter search tree path to w via (r', w). We can prove m will not be heaped-up above n using the same argument as in Case 2. □

Lemmas 6 and 7 ensure that the changes in $\mathcal{P}(G)$, which are induced by A*, do not influence the portion of $\mathcal{P}(G)$ which Dijkstra has already explored. This guarantees the correctness of the Dijkstra search on $\mathcal{P}(G)$, if the used heuristic is admissible. Thus, each path which Dijkstra provides is correct and its length is valid. However, this does not ensure the completeness of the Dijkstra search on $\mathcal{P}(G)$.

It is possible that a node n' is 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'. We accomplish this by applying the lines from 20 to 22 of Algorithm 1 to n' for each expanded direct predecessor of n'. If n' does not yet fulfill the scheduling condition, A* will be repeatedly resumed until the scheduling mechanism allows Dijkstra to put n' into its search queue. Notice that doing so does not require any extra effort during the typical Dijkstra search.

We can be sure that no explored node was forced down by the heaping-up of n'. Otherwise, we would have a node n'' which was a child of n and subsequently replaced by n'. Notice that n'' must have been explored, since n was expanded. However, this is contradictory to Lemma 7 which assures that this cannot happen.

Moreover, the following corollary ensures that the best $d(n')$ is not better than the d-values of any explored node, in particular, any expanded node. This means that we did not miss the opportunity to expand n'.

Corollary 3. *Let n be a node in $\mathcal{P}(G)$ which has been explored by Dijkstra. Furthermore, let m be a node which newly added into $\mathcal{P}(G)$ or its position has been modified, after n has been explored. If h is admissible, then it holds that*:

$$C_{\mathcal{P}(G)}(\mathfrak{R}, m) \geqslant d(n).$$

Proof. The proof can be derived from the proof of Lemma 7. Notice that $C_{\mathcal{P}(G)}(\mathfrak{R}, m)$ is equal to $C_{\mathcal{P}(G)}(\sigma)$ where σ is the shortest path from the root $\mathfrak{R}$ to m. □

4.6. Example

We illustrate how K* works using the following example. We examine the directed, weighted graph G in Fig. 7. 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. The heuristic values are given by the labels $h(s_0)$ to $h(s_6)$ in Fig. 7. It is easy to see that this heuristic function is admissible.

A* first searches the graph G until s_6 is found. The section of G explored so far is illustrated in Fig. 8. The edges that are depicted using solid lines indicate the tree edges, while all of the other edges are sidetrack edges. They are stored in H_{in} heaps, as shown in Fig. 9. The numbers attached to the heap nodes are the corresponding δ-values. At this point of the search, A* is suspended and $\mathcal{P}(G)$ is constructed. Initially, only the designated root $\mathfrak{R}$ is explicitly available in $\mathcal{P}(G)$. Dijkstra's algorithm is initialized. This means, the node $\mathfrak{R}$ is added into Dijkstra's search queue. The scheduler needs to access the successors of $\mathfrak{R}$ in order to decide whether Dijkstra or A* 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 they were not needed for building $H_T(s_6)$. The result is shown in Fig. 10, where solid lines represent heap edges and dashed lines indicate cross edges. In order to avoid clutter in the image some of the edges are not completely drawn in the figure. We indicate each of them using a short arrow with a specified target.

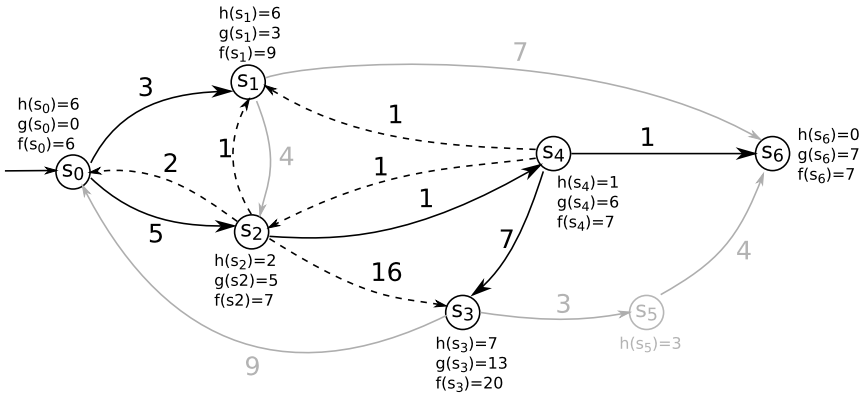


Fig. 8. The explored part of G (1), see the example in Section 4.6. The faintly gray vertices and edges have not yet been explored by A*. The solid line edges highlight the current search tree of A*, whereas dashed line edges are the sidetrack edges.

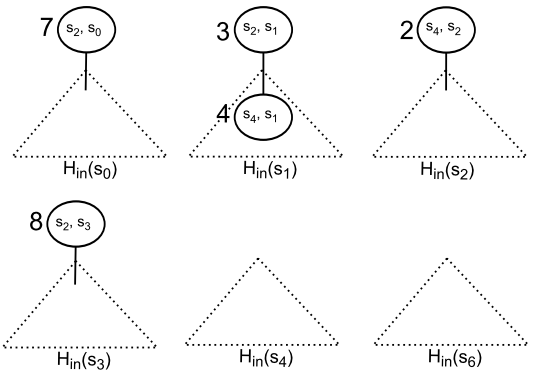


Fig. 9. The H_{in} heaps constructed by K* (1), see the example in Section 4.6.

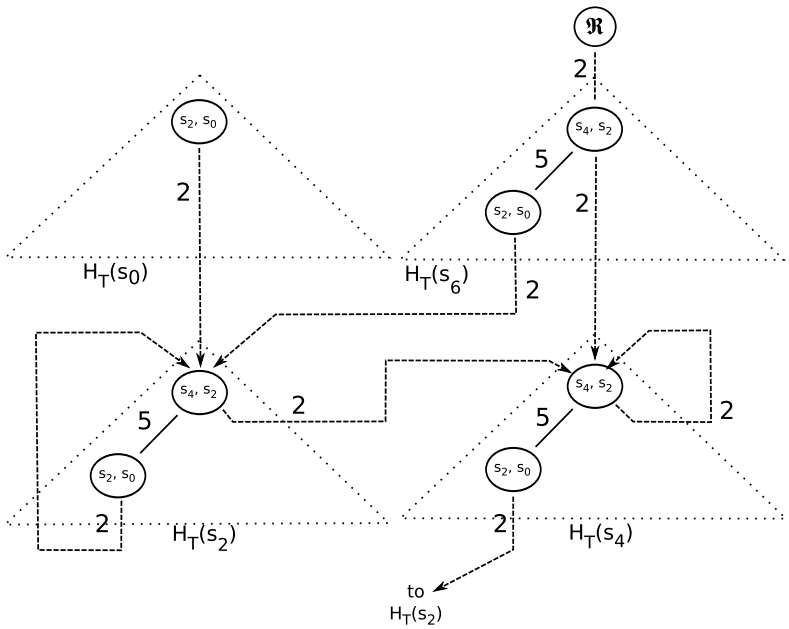


Fig. 10. The path graph $\mathcal{P}(G)$ (1) constructed by K* at the point of A* search presented in Fig. 8, see the example in Section 4.6.

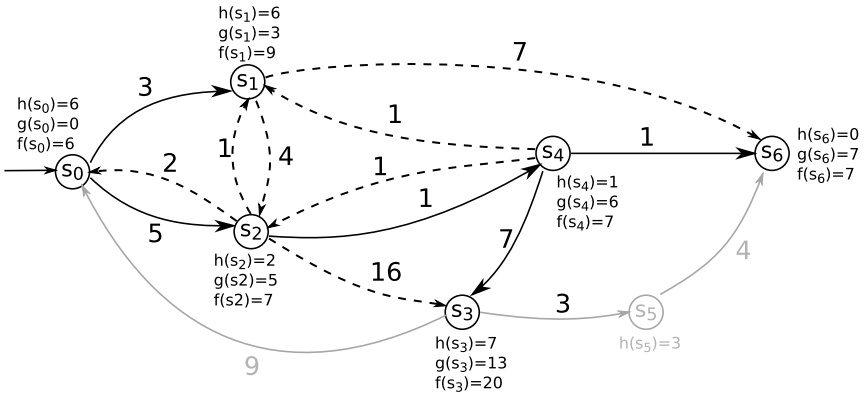


Fig. 11. The explored part of G (2), see the example in Section 4.6. The faintly gray vertices and edges have not yet been explored by A*. The solid edges indicate the current search tree of A*, whereas dashed line edges are the sidetrack edges.

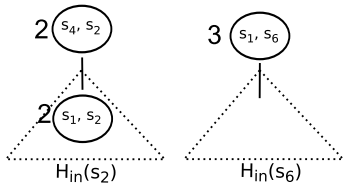


Fig. 12. The modified H_{in} heaps after the extension, see the example in Section 4.6.

Table 1
The result of K* applied to the graph G from Fig. 7.

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

After constructing $\mathcal{P}(G)$, as shown in Fig. 10, the scheduler checks for the only child (s_4,s_2) of $\Re$ whether $g(s_6)+d(s_4,s_2)\leqslant f(s_1)$. Note that s_1 is the head of the search queue of A*. The value $d(s_4,s_2)$ is equal to 2. It holds that $g(s_6)+d(s_4,s_2)=7+2=9=f(s_1)$. Hence, the scheduler allows Dijkstra's algorithm to expand $\Re$ and insert (s_4,s_2) into its search queue. On expanding $\Re$ the first solution path is delivered. It is constructed from the $\mathcal{P}(G)$ path consisting of the single node $\Re$. This path results in an empty sequence of sidetrack edges. Recall that the empty sidetrack edge sequence corresponds to the tree path s_0 to s_6, namely $s_0\,s_2\,s_4\,s_6$ with the length 7. The Dijkstra search is then suspended because the successors of (s_4,s_2) do not fulfill the scheduling condition $g(s_6)+d(n)\leqslant f(s_1)$. Hence, A* is resumed.

We assume the extension condition to be defined as the expansion of one vertex in order to keep the example simple and illustrative. Consequently, A* expands s_1 and stops. The explored part of G at this point is given in Fig. 11. This extension results in 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 Fig. 12. The other H_{in} heaps remain unchanged as in Fig. 9. The path graph $\mathcal{P}(G)$ is rebuilt as shown in Fig. 13. Dijkstra's algorithm is then resumed. Notice that the Dijkstra search queue contains only (s_4,s_2) with $d=2$ at this point. Using manual execution we can easily see that Dijkstra will deliver the solution paths enumerated in Table 1.

5. Properties of K*

In this section we examine fundamental properties of K*. Following a convention in the literature on directed search, we separately address the correctness of K*, which means that K* delivers valid **s**–**t** paths, and its solution optimality [15]. We first show that K* is correct and complete. We then establish the admissibility of K*, which is the key precondition for

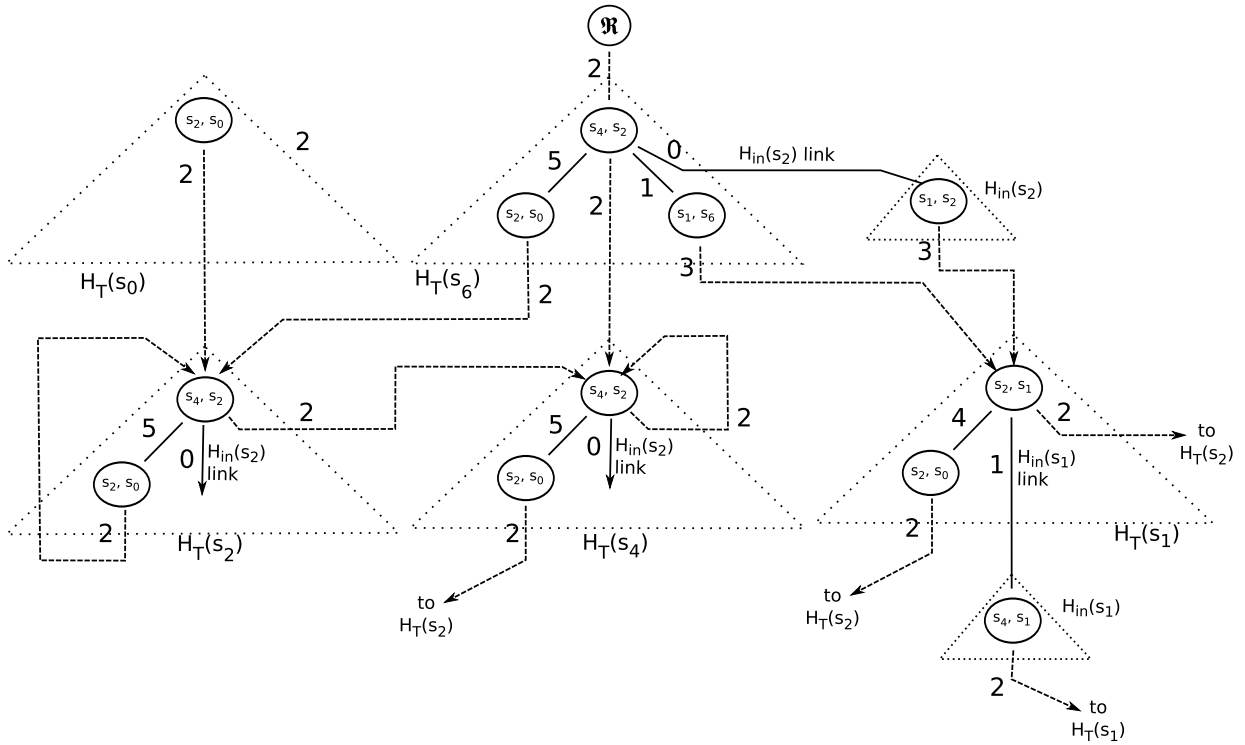


Fig. 13. The path graph $\mathcal{P}(G)$ (2) after the extension presented in Fig. 11, see the example in Section 4.6.

solution optimality of a heuristic search based algorithm. We finally discuss the asymptotic worst-case runtime and space complexity of K*.

5.1. Correctness and completeness

The correctness of K* means that K*, when applied to an arbitrary directed graph, delivers valid **s**–**t** paths. On the other hand, the completeness of K* means that it finds k **s**–**t** paths for any natural number k. If $k > |\Pi(\mathbf{s}, \mathbf{t})|$, then K* generates all **s**–**t** paths. Recall that $\Pi(\mathbf{s}, \mathbf{t})$ is the set of all **s**–**t** paths. In the following theorem we prove the correctness and completeness of K* on locally finite graphs. We follow a common argument in the literature on directed search in that we assume the cost of an infinite path to be unbounded, cf. [15].

Theorem 1. *K* is correct and complete on locally finite graphs.*

Proof. Let G be a locally finite graph. Corollary 1 ensures a one-to-one correspondence between paths in $\mathcal{P}(G)$ and **s**–**t** paths in G. This implies the correctness of K* since any $\mathcal{P}(G)$ path from $\mathfrak{R}$ to any node results in a valid **s**–**t** path. Notice that the Dijkstra search performed by K* on $\mathcal{P}(G)$ delivers paths from $\mathfrak{R}$ to some nodes in $\mathcal{P}(G)$. In other words, the result of K* consists of valid **s**–**t** paths.

The completeness of A* on locally finite graphs [15] ensures that any edge in G will be explored by A* after a finite number of A* iterations. This implies that all sidetrack edges of any **s**–**t** path π will be added to $\mathcal{P}(G)$ after a finite number of A* iterations. Consequently, we can use the surjectivity of the mapping of p to conclude that a path σ in $\mathcal{P}(G)$ with $p(\sigma) = \pi$ exists. Due to the completeness of Dijkstra on $\mathcal{P}(G)$, σ will be found by Dijkstra following a finite number of iterations. In conclusion, K* is able to deliver any **s**–**t** path after a finite number of iterations.

Let k be an arbitrary natural number and $k' = min\{k, |\Pi(\mathbf{s}, \mathbf{t})|\}$. A* will explore all k' **s**–**t** paths after a finite number of iterations. Consequently, Dijkstra will detect all corresponding paths in $\mathcal{P}(G)$ at some point. Hence, K* delivers these k' **s**–**t** paths. □

Lemma 8. *K* always terminates on finite graphs.*

Proof. If $k \leqslant |\Pi(\mathbf{s}, \mathbf{t})|$, then K* will find k **s**–**t** paths and terminate after executing the **if**-statement at line 25 in Algorithm 1. If $k > |\Pi(\mathbf{s}, \mathbf{t})|$, then K* will behave as follows. A* will be resumed repeatedly since Dijkstra cannot find k solution paths.

This continues until A* has explored the complete graph G. Afterwards, open_D will become empty once Dijkstra has delivered $|\Pi(\mathbf{s},\mathbf{t})|$ paths. The K* loop will then terminate since both search queues are empty (see line 8 in Algorithm 1). □

From the proof of Lemma 8 we can easily infer the following corollary.

Corollary 4. *For $k \leqslant |\Pi(\mathbf{s},\mathbf{t})|$, K* terminates even on infinite graphs.*

5.2. Admissibility

We now show that, if the heuristic used is admissible, then the result of K* is optimal. This means that the delivered **s**–**t** paths are indeed the shortest ones.

Theorem 2. *If h is admissible, then the* **s**–**t** *paths that are delivered at any point by K* are the shortest possible paths, and we then say that K* is admissible.*

Proof. We have to prove that any **s**–**t** path which has been delivered at any point of the search is at least as short as any **s**–**t** path which has not yet been delivered. Actually, Dijkstra's algorithm ensures that shorter $\mathcal{P}(G)$ paths are explored first. However, we still need to prove that A* will never explore new edges which lead to shorter paths after an **s**–**t** path π has been delivered. This is assured by the scheduling mechanism which K* uses to schedule Dijkstra and A*. Let π be the last delivered **s**–**t** path. Let σ be the $\mathcal{P}(G)$ path from which π is formed, i.e., $\pi = p(\sigma)$. Let n be the last node in σ. Moreover, let π' be another **s**–**t** path which has not yet been completely explored by A*. When Dijkstra explores σ and is expanding n at line 18, at least one vertex v of π' must be in the queue of A*. K* only allows Dijkstra to expand n if $g(\mathbf{t}) + d \leqslant f(u)$, where $d = \max\{d(n) + \Delta(n,n') \mid n' \in succ(n)\}$ and u is the head of the A* queue. Note that $d \geqslant d(n) = C_{\mathcal{P}(G)}(\sigma)$ and $f(u) \leqslant f(v)$. Thus, it holds that $g(\mathbf{t}) + C_{\mathcal{P}(G)}(\sigma) \leqslant f(v)$. Since h is admissible we know that $g(\mathbf{t}) = C^*(\mathbf{s},\mathbf{t})$ and $f(v) \leqslant C(\pi')$. From this fact we conclude that $C^*(\mathbf{s},\mathbf{t}) + C_{\mathcal{P}(G)}(\sigma) \leqslant C(\pi')$. Furthermore, due to Lemma 5, we know that $C(\pi) = C^*(\mathbf{s},\mathbf{t}) + C_{\mathcal{P}(G)}(\sigma)$. We hence conclude that $C(\pi) \leqslant C(\pi')$. □

From the previous theorem we can immediately conclude that, when using an admissible heuristic, K* enumerates **s**–**t** paths in a non-decreasing order with respect their length. The delivered paths are the shortest in G.

Theorem 3. *K* solves the KSP problem, if h is admissible.*

Proof. Theorem 1 ensures the correctness and completeness of K*. In other words, K* terminates and delivers k **s**–**t** paths. If $k > |\Pi(\mathbf{s},\mathbf{t})|$, then K* delivers all solution paths. In this case termination is only guaranteed on finite graphs. Moreover, Theorem 2 ensures that the paths, which K* delivers, are the shortest ones. This allows us to make the following observations:

1. The result set $\mathcal{R}$ contains the shortest **s**–**t** paths, which means that each path in $\mathcal{R}$ is at least as short as any path in $\Pi(\mathbf{s},\mathbf{t})\backslash\mathcal{R}$.
2. **s**–**t** paths are found and added into $\mathcal{R}$ in a non-decreasing order with respect to their length.

From these observations we conclude that K* solves the KSP problem. □

An important property of K* is that it does not require the desired number k of solution paths to be defined in advance. K* is able to enumerate more and more shortest **s**–**t** paths in a non-decreasing order until the user decides to terminate the algorithm, or G is completely explored.

5.3. Complexity

The computation of K* comprises the following steps:

(a) An A* search on G.
(b) The construction and restructuring of $\mathcal{P}(G)$.
(c) A Dijkstra's search on $\mathcal{P}(G)$ to find the k shortest paths.
(c) The scheduling mechanism between A* and Dijkstra's algorithm.

The runtime behavior of these steps depends on the characteristic of the heuristic estimate h used by A*. We therefore distinguish the following three cases in our complexity analysis: h is not admissible, h is admissible but not consistent, and h is consistent.

The asymptotic worst-case complexity of K^* is dominated by that of A*, which is exponential in the number of vertices if an inconsistent heuristic is used. Nonetheless, with an informative heuristic, A* can perform very efficiently on practical problems, even if the heuristic is inconsistent. We are therefore interested in determining the worst case complexity of K* in all of the above mentioned cases of heuristic estimates used, considering both cases of excluding and including the effort necessary for exploring G using A*. When comparing with the complexity of an uninformed algorithm, such as LVEA, we assume the trivial heuristic estimate, $h = 0$. Notice that when a trivial heuristic is used, A* turns into a Dijkstra search which has an asymptotic worst-case complexity of $\mathcal{O}(m + n \log n)$.

Case I: h is not admissible. With a non-admissible heuristic estimate, it is difficult to predict the impact of resuming A* on $\mathcal{P}(G)$, which consequently needs to be constructed from scratch in each K* iteration. The construction includes building the H_{in} and H_T heaps. Let n_i be the number of vertices and m_i the number of inner edges which A* has explored after it is resumed for the i-th time. If we assume the use of Fibonacci heaps, then adding a node into a heap can be done in constant time, i.e., with worst-case complexity $\mathcal{O}(1)$. Constructing all H_{in} heaps at this point will therefore take $\mathcal{O}(m_i)$ time. Furthermore, all H_T together contain $\mathcal{O}(n_i \log n_i)$ nodes since the tree edge of each node is represented in H_T by one node, which means that constructing all H_T heaps will take $\mathcal{O}(n_i \log n_i)$ time. We assume that the extension condition is defined so that the number of expanded vertices and the number of inner edges are doubled within each run of A*, which means that $n_i \geqslant 2\, n_{i-1}$ and $m_i \geqslant 2\, m_{i-1}$. We can deduce that the effort for the repeated construction of H_{in} and H_T heaps, up to the i-th iteration, is $\mathcal{O}(2 \cdot m_i) = \mathcal{O}(m_i)$ and $\mathcal{O}(2 \cdot n_i \log n_i) = \mathcal{O}(n_i \log n_i)$ respectively. Thus, the effort required for the repeated reconstruction of H_{in} heaps is $\mathcal{O}(m)$ and for H_T heaps is $\mathcal{O}(n \log n)$. This means that the total effort invested in constructing and restructuring $\mathcal{P}(G)$ is $\mathcal{O}(m + n \log n)$. Note that this is asymptotically equal to constructing the complete $\mathcal{P}(G)$ only once.

Due to doubling the number of expanded vertices in each A* run, K* performs at most $\log n$ iterations. Notice that reopening of vertices by A* would certainly interfere with the speed of A* and slow down the doubling of the vertices in each A* run. However, due to the scheduling policy used this slowdown neither increase the number of K* iterations, nor does it influence the effort for constructing $\mathcal{P}(G)$.

Notice that due to the definition of the extension condition the scheduling mechanism runs at most $\log n$ times. In each run, the successors of the node on the head of the Dijkstra search queue are generated and their d-values are compared with the f-value of the head of the A* queue. This effort is constant since every node in $\mathcal{P}(G)$ has 4 successors at most. Moreover, the look-up operation on a heap can be performed in constant time. Hence, the effort caused by the scheduling mechanism is $\mathcal{O}(\log n)$.

In order to compute k paths, Dijkstra performs k iterations on $\mathcal{P}(G)$. Since each node in $\mathcal{P}(G)$ has at most 4 successors, in k Dijkstra iterations no more than $4k$ nodes are added into the Dijkstra queue. Operations on the Dijkstra queue have a logarithmic runtime, which leaves us with an effort for Dijkstra of $\mathcal{O}(k \log k)$. With an arbitrary heuristic estimate we cannot predict the impact of the continuous changes in $\mathcal{P}(G)$ on the correctness of Dijkstra's search. We will therefore need to restart Dijkstra's search from scratch after each resumption of A*. This yields a runtime of $\mathcal{O}(k \log k \log n)$.

Frederickson has presented an algorithm which selects the k smallest nodes in a heap in time $\mathcal{O}(k)$ [18]. If this algorithm is used on $\mathcal{P}(G)$ instead of Dijkstra's search, then we improve the effort for getting the k shortest paths to $\mathcal{O}(k \log n)$.

The above considerations leave us with a total complexity of $\mathcal{O}(m + n \log n + k \log n)$ for K* excluding the effort of exploring G using A*. As argued above, however, the total complexity in this case is dominated by the exponential complexity of A*.

Case II: h is admissible, but not consistent. For an admissible h we have proved in Section 4.5 that the results of Dijkstra's search on $\mathcal{P}(G)$ are not destroyed when A* changes $\mathcal{P}(G)$. Hence, restarting Dijkstra from scratch is not necessary. This means that Dijkstra performs in total k iterations. We hence obtain an effort for Dijkstra of $\mathcal{O}(k \log k)$. Using Frederickson's algorithm we can improve this to $\mathcal{O}(k)$.

This results in the total complexity $\mathcal{O}(m + n \log n + k)$ for K* excluding the effort of exploring G using A*. As in the previous case, the total complexity of K* is, however, dominated by the exponential complexity of A* in this case.

Case III: h is consistent. Since a consistent heuristic estimate is admissible, the argumentation in Case II applies here too. We obtain a complexity of $\mathcal{O}(m + n \log n + k)$ for K* excluding A*. We argued in Section 4.5 that it is not necessary to reconstruct $\mathcal{P}(G)$ from scratch or to globally restructure it. This results in a major improvement in the performance of K*, although it does not improve the asymptotic complexity.

Moreover, if h is consistent, A* will never expand a vertex more than once. This leads to an asymptotic worst-case complexity of $\mathcal{O}(m + n \log n)$ for A* on G. We obtain a total asymptotic worst-case complexity of $\mathcal{O}(m + n \log n + k)$ for K* including the exploration of G using A*.

With a consistent heuristic K* hence maintains an asymptotic worst-case complexity of $\mathcal{O}(m + n \log n + k)$, including the effort necessary for exploring G using A*. Note that the trivial heuristic $h = 0$ is consistent. Hence, we consider this complexity to be applicable when we compare K* to EA and LVEA.

Space complexity. The asymptotic space complexity of K* is determined by (1) the space needed for A*, (2) the space consumed by the data structure of $\mathcal{P}(G)$, and (3) the space needed for the Dijkstra search on $\mathcal{P}(G)$. The worst-case space

complexity of A* is $\mathcal{O}(n)$. $\mathcal{P}(G)$ contains $\mathcal{O}(m + n \log n)$ nodes and, at most, 4 edges for each node. Hence, $\mathcal{P}(G)$ consumes, in total, a space of $\mathcal{O}(m + n \log n)$. As mentioned in the previous paragraph, Dijkstra would visit $\mathcal{O}(k)$ nodes of $\mathcal{P}(G)$ in order to find k shortest paths. This means a space complexity of $\mathcal{O}(k)$. Altogether, we get obtain a space complexity of $\mathcal{O}(m + n \log n + k)$ for K*.

6. Experimental evaluation

We demonstrate the advantages of K* by applying it to two case studies from different application domains. The first one is from the context of *route planning*, whereas the other one is from the domain of *counterexample generation for stochastic model checking*. We experimentally compare K* to LVEA. We argued in Section 3.2 that LVEA is the state-of-the-art KSP algorithm, particularly for the class of large graphs with hundreds of thousands or even millions of vertices and edges that K* is designed for and which we use in our experimental evaluation. For this class of graphs, LVEA outperforms REA, and consequently MSA and EA. Our experiments show that K* provides the solution before LVEA finishes the exhaustive exploration of the problem graph, which is the first step that LVEA performs. Since MSA, EA, REA and LVEA require such an exhaustive exploration, it is obvious that K* outperforms all these algorithms on large graphs.

We implemented K* and LVEA in Java. We ran all experiments on a machine equipped with an Intel Pentium dual core CPU with 3.2 GHz speed and 2 GB of memory. Notice that we could not benefit from the parallel hardware since no parallel implementations for the investigated algorithms are available.

6.1. Case study: route planning

Methods for route planning have been extensively researched in recent years. For a very comprehensive overview of route planning algorithms and methods see, for instance, [19]. The original route planning problem is to find an optimal, or even sub-optimal, route from a start to a goal location. The typical domain for the application of this problem is logistics, where one is interested in optimizing the routes of vehicles in a road map. KSP algorithms are used when alternative routes are required or when additional constraints on routes need to be satisfied. Our goal in this section is not to propose a new method for route planning. Instead we use this case study as a benchmark to investigate the performance and scalability of K*. We demonstrate that (a) K* can serve as a basis for route planning methods, and (b) that K* scales nicely when applied to large problem instances in this domain. Notice that K* provides solution paths which may contain loops. In route planing, loops in the route are usually not desired. Algorithms for finding k shortest loop-less paths like Yen's algorithm [20] seem to be more suitable for this application. However, this restriction makes the problem significantly harder [1]. For example, the computational complexity of Yen's algorithm is $\mathcal{O}(kn(m + n \log n))$ [20]. This is very high compared to $\mathcal{O}(m + n \log n + k)$ which is the complexity of EA, LVEA and K*. Moreover, we are not aware of an algorithm for finding the k shortest loop-less paths which is able to operate on-the-fly and to be guided using search heuristics. For these reasons, applying K* and discarding those routes which contain loops represents an efficient solution for this problem.

In our experiments we use a road map of the USA as it is available from the home page of the 9th DIMACS Implementation Challenge [21]. The road maps are stored as adjacency lists in an SQL database. This database is stored on the hard-drive of the computer on which we run the experiments. For each vertex in the map, obtaining the successor vertices means an SQL query to the database. Such a query is an expensive operation since it requires several hard-disk accesses. Due to its on-the-fly feature and the heuristic guidance it takes advantage of, K* minimizes number of database accesses, the processing effort, and the memory consumption necessary to perform the route planning task.

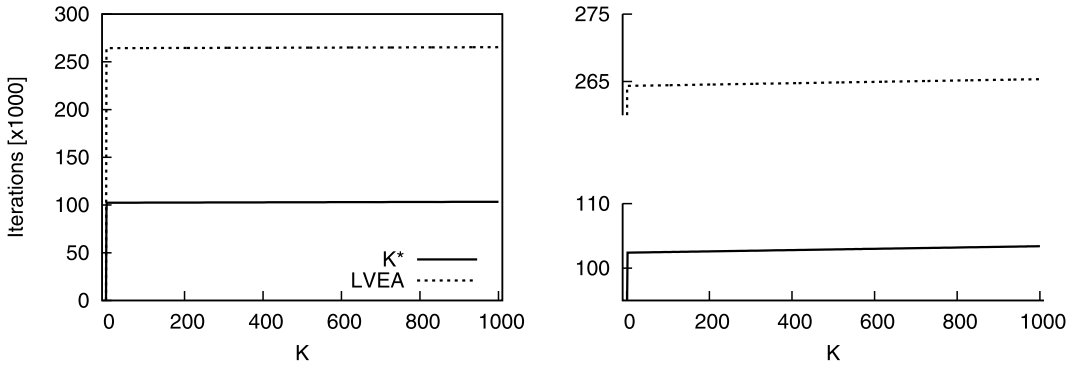
Experiment 1 – New York City. In the first experiment we consider a map of New York City. This map consists of 264,346 nodes and 733,846 edges. We apply LVEA and K* to the graph in order to find the first 1000 optimal routes from a starting point in the city center to various targets. We select four targets so that they are located in different directions from the starting point, with a shortest target distance of approximately 50 km. We use the *airline distance*, computed according to the *cosine law*, as heuristic estimate for K*. The cosine law computes the airline distance between two points as follows:

$$a = \sin(lat_1) \cdot \sin(lat_2) + \cos(lat_1) \cdot \cos(lat_2) \cdot \cos(lon_2 - lon_1)$$

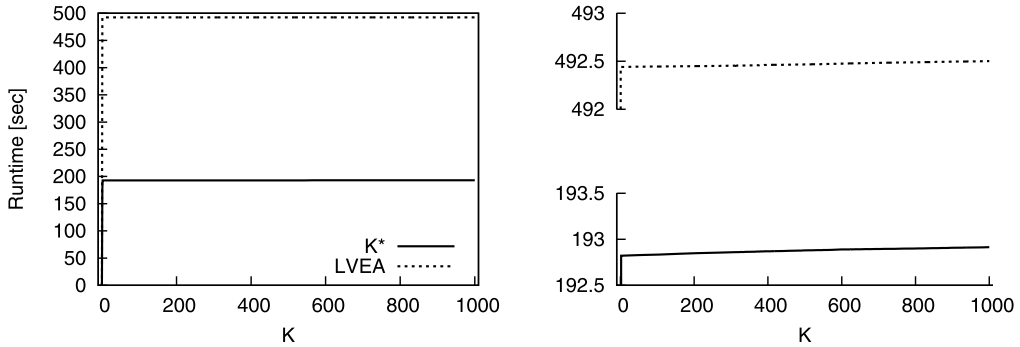
$$\text{Airline distance} = \arccos(a) \cdot \text{Earth radius}$$

lon_i and lat_i are the longitude and latitude of the i-th point in the radian system. Notice that the radius of the Earth is not constant since it is not perfectly spherical. We therefore assume a radius of 6350 km, which is slightly smaller than the minimal actual value, in order to ensure that the airline distance heuristic is admissible.

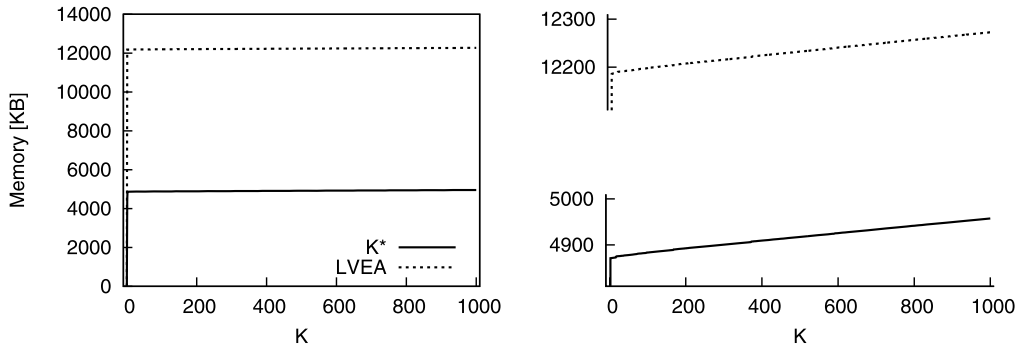
We next compute the mean effort required for each algorithm to find the 1000 optimal routes in the above setting. The results are illustrated in Fig. 14. Fig. 14a reports the mean number of iterations required by the algorithm. It is computed as the total number of iterations executed while exploring the problem graph plus the number of iterations executed while



(a) Mean number of iterations needed for finding the routes. The diagrams on the right-hand side zoom into the interesting segments of the curves.



(b) Mean runtime needed for finding the routes. The diagrams on the right-hand side zoom into the interesting segments of the curves.



(c) Mean memory consumption for finding the routes. The diagrams on the right-hand side zoom into the interesting segments of the curves.

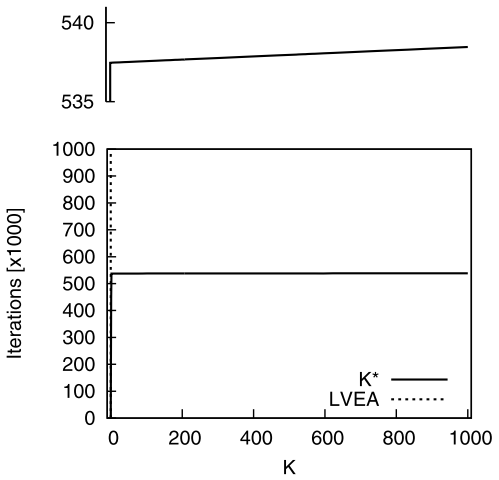
Fig. 14. Results of searching k shortest routes in New York City using K* and LVEA.

searching the path graph. Figs. 14b and 14c show the runtime and the memory consumption[2] for both algorithms. In the diagrams on the right-hand side of each figure we zoom into the interesting segments of the respective curves.

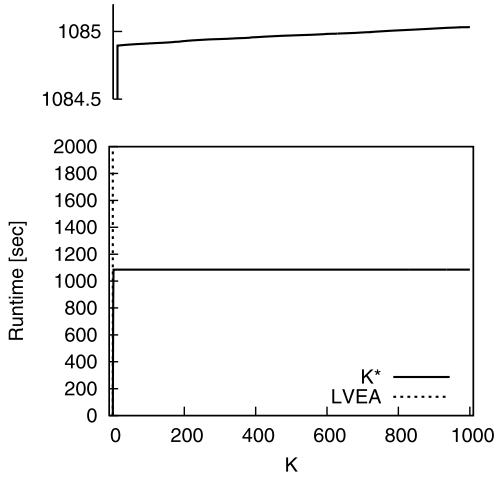
We see that K* requires less than half of the computation effort and memory required by LVEA. Even though the problem graph is not extremely large, we see that K* outperforms LVEA even for such a medium size problem instance. We observe that the K* curves start in all diagrams with a steep increase which indicates the initial A* exploration of the map. Afterwards we see a flat curve which corresponds to the Dijkstra search selecting the shortest routes from the path graph. We computed that A* explores on average about 102,960 vertices and 499,004 edges in order to enable K* to provide the required 1000 routes. This is about the half of the map. In this example, resuming A* in K* is not necessary.

Experiment 2 – Eastern USA: In this experiment we consider a map of the east coast of the USA. This map consists of 3,598,623 nodes and 8,778,114 edges. It is more than 10 times larger than the map of New York City and hence qualifies as a large problem instance. We keep the same starting point as in the first experiment, however, we choose 4 different targets

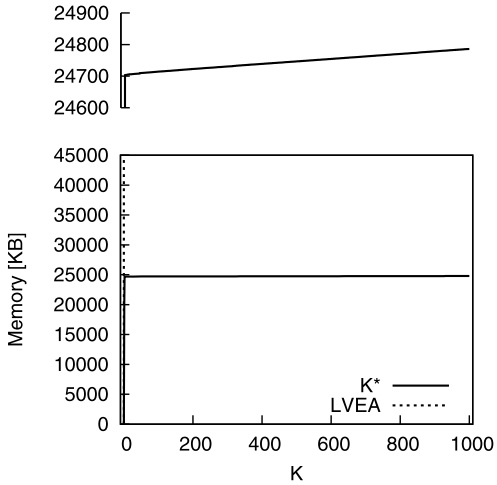
[2] We would like to point out that the algorithms are implemented as a part of a larger toolbox. The stated memory consumption reflects the net usage of the algorithms obtained by calculating the size of their data structures. The total amount of memory claimed by the Java Runtime Environment running this toolbox is much higher.



(a) Mean number of iterations needed for finding the routes. The diagram on the top zooms into the interesting segment of the curve of K*.



(b) Mean runtime needed for finding the routes. The diagram on the top zooms into the interesting segment of the curve of K*.



(c) Mean memory consumption for finding the routes. The diagram on the top zooms into the interesting segment of the curve of K*.

Fig. 15. Results of searching k shortest routes in Eastern USA using K* and LVEA.

at approximately 200 km distance from the starting point. We use the same airline distance heuristic as in Experiment 1. Fig. 15 shows the results of both algorithms for finding the shortest 1000 routes. The plots show the mean number of iterations, runtime and memory consumption as in the first experiment. At the top of each diagram we zoom into the interesting segment of the respective curves.

After approximately 30 minutes (1800 seconds) of runtime LVEA failed to find a route. Hence, the curves of LVEA run parallel to the Y-axis at $K = 0$. LVEA consumed 45 MB of memory and explored about one million vertices and 4 million edges. This amounts to 30% of the graph. Remember that LVEA has to explore the entire graph before computing any route. LVEA is therefore expected to need three times as much time and memory as it consumed for this aborted run in order to be able to return the first result path.

On the other hand, K* succeeds in producing all 1000 shortest routes for all four targets. Its mean runtime was approximately 1100 seconds per search. It required approximately 25 MB of memory on average. Again we observe the steep increase of the K* curves at the beginning which corresponds to the initial A* exploration of the map. Afterwards we see a curve with a flat gradient which indicates the Dijkstra search selecting the shortest routes from the path graph. We observe furthermore that A* explores on average about 538,454 vertices and 2,314,865 edges in order to enable K* to provide the required 1000 routes. This is about 15% of the map vertices and 26% of the map edges. Resuming A* in K* is, again, not necessary.

6.2. Case study: counterexamples for stochastic model checking

We now consider a case study from the domain of computing the counterexample for a stochastic model checking problem. As we explained in Section 1, a counterexample in this context is set of system executions which lead to system failure so that their accumulated probability mass exceeds a probability limit. To facilitate debugging we are interested in enumerating the failure paths starting with the highest probabilities in a decreasing order. The counterexample generation problem in stochastic model checking can hence be cast as a KSP problem.

In this case study we use a continuous-time Markov chain modeling a dependable cluster of workstations. This case study was first presented in [22] and it is frequently used as a benchmark in the literature. It represents a system consisting of two sub-clusters connected via a backbone. Each sub-cluster consists of N workstations with a central switch that provides the interface to the backbone. Each of the components of the system (workstations, switches, and backbone) can break down probabilistically with a certain rate. At least $\frac{2}{3}N$ workstations have to be operational and connected to each other via operational switches in order to provide minimum quality of service (QoS).

The Markov chain is provided in a high-level process-algebra like modeling language which serves as an input language for the stochastic model checker PRISM [23]. The PRISM model describes the system's initial configuration and an implicit transition function. The transition function computes for a given system configuration the possible successor configurations reachable by executing one of the enabled actions of the system. K* operates on-the-fly using this representation of the model. As our experiments will show, just like in the route planning case studies K* succeeds to generate and process only a part of the model, whereas LVEA tries to generate and explore the entire model.

We are interested in determining the likelihood that the QoS provided by this system drops below a desired minimum within 100 time units. We set the parameter N to 64, which results in a continuous-time Markov chain with 151,060 states and 733,216 transitions. The PRISM model checker computes the total probability of reaching a state with insufficient QoS within 100 time units as $5.022 \cdot 10^{-5}$. Notice that it is not necessary for our counterexample computation algorithm to perform the model checking beforehand. However, having the total probability available allows us to know that the desired property is violated for any probability upper-bound which is smaller than $5.022 \cdot 10^{-5}$. We apply two counterexample generation methods based on K* and LVEA in order to generate counterexamples for all probability bounds up to $5.022 \cdot 10^{-5}$. In this case study we use K* without a heuristic estimate.

The results of the counterexample generation are illustrated in Fig. 16. The X-axis indicates the probability of the counterexample that has been selected so far. Fig. 16a shows the number of iterations executed by each algorithm in order to generate a counterexample with a certain probability. Fig. 16b shows the runtime and 16d shows the memory consumption. Fig. 16c shows the number of failure paths needed to form a counterexample with a certain probability.

We immediately see that K* is significantly superior to LVEA with respect to both computational effort and memory consumption. We also notice an overhead effort in LVEA before it delivers any counterexample paths. The reason for this is that LVEA has to explore the entire Markov chain before it can deliver the first counterexample paths. We hence observe that K* can outperform LVEA due to its on-the-fly nature even if no heuristic estimate is used.

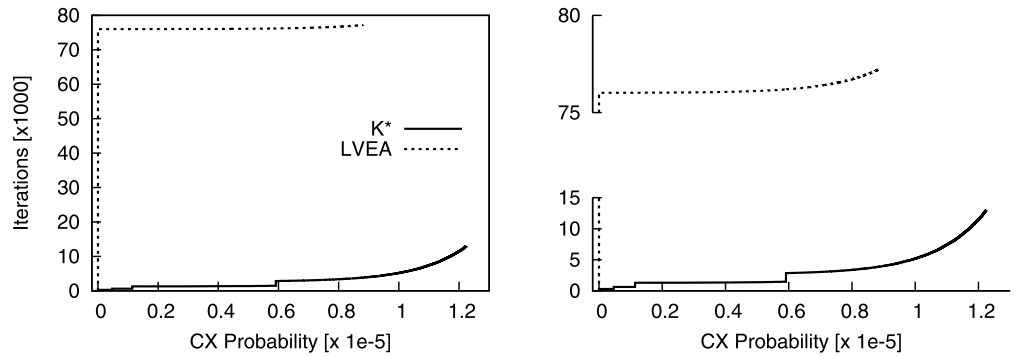
Unlike in the experiments of the route planing case study we do not see the steep increase in the K* curves at the beginning. Instead we observe, for instance in Fig. 16a, a flat start of the K* curve. This indicates that A* was able to locate a goal vertex quickly and produce counterexample paths early on in the computation. We also observe several vertical jumps in the K* curve in Fig. 16a. These jumps indicate search iterations which do not result in new counterexample paths being produced. Such iterations can only be due to A* since Dijkstra would find a new counterexample in each of its iterations. These jumps hence point at the fact that A* was resumed several times during the search process. Due to figures which are not reported in the diagrams included in this paper we know that A* explores 5289 vertices and 20,547 edges by the time it returns the result. This corresponds to about 3% of the state space of the entire model.

Both algorithms failed to provide counterexamples for probability bounds larger than $1.25 \cdot 10^{-5}$ within one hour of runtime. The reason is the large number of paths needed to compile a counterexample with such an amount of probability mass for this example. As shown in Fig. 16c, about 10,000 paths were needed in order to obtain a counterexample with the probability $1.2 \cdot 10^{-5}$. We see that the number of counterexample paths grows in a seemingly exponential fashion. This is a challenge which needs to be addressed in future research in the domain of counterexample computation for stochastic model checking.

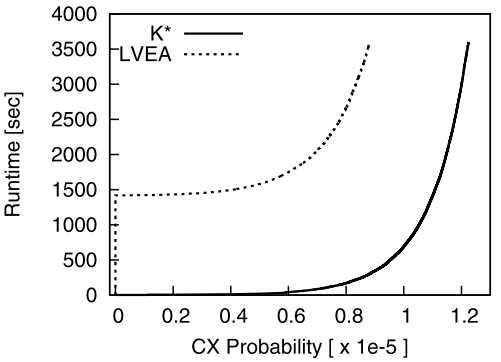
7. Conclusion

We presented a new directed algorithm, K*, for solving the KSP problem. K* operates on-the-fly and can be guided using heuristic estimates. We have argued for its correctness and determined its asymptotic worst-case complexity as $\mathcal{O}(m + n \log n + k)$, which matches that of the state of the art KSP algorithms. However, we presented a number of experiments which show the efficiency and scalability of K*. They also establish that when applied to large problems, K* performs better than the state of the art KSP algorithms.

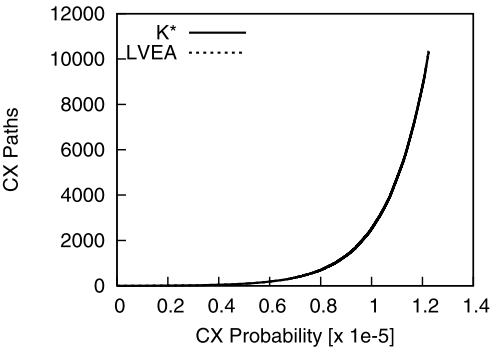
Future work includes the parallelization of K* and the applicability of directed search to other variants of the KSP problem.



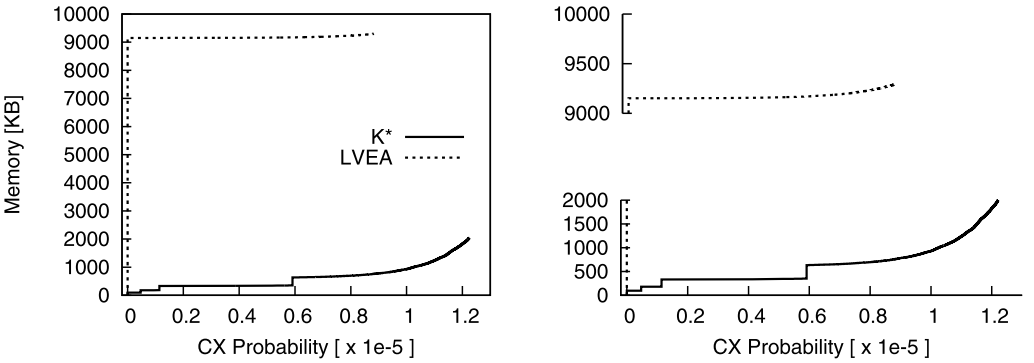
(a) The number of iterations needed for computing a counterexample with for a certain probability. The diagrams on the right-hand side zoom into the interesting segments of the curves.



(b) The runtime needed for computing a counterexample with for a certain probability.



(c) The number of paths included in a counterexample with for a certain probability.



(d) The memory consumed for computing a counterexample with for a certain probability. The diagrams on the right-hand side zoom into the interesting segments of the curves.

Fig. 16. Results of the counterexample generation for the Workstation Cluster model using K* and LVEA.

Acknowledgements

The authors are grateful to Ulrik Brandes for his comments on an earlier version of this work. They also wish to express their gratitude for the helpful and constructive criticism provided by the anonymous reviewers.

References

[1] D. Eppstein, Finding the *k* shortest paths, SIAM J. Computing 28 (2) (1998) 652–673. URL http://dx.doi.org/10.1137/S0097539795290477.
[2] C. Baier, J.-P. Katoen, Principles of Model Checking, The MIT Press, 2008.
[3] T. Han, J.-P. Katoen, Counterexamples in probabilistic model checking, in: 13th International Conference on Tools and Algorithms for the Construction of Systems (TACAS '07), in: Lecture Notes in Computer Science, vol. 4424, Springer, 2007, pp. 72–86.
[4] H. Aljazzar, Directed diagnostics of system dependability models, PhD thesis, University of Konstanz, 2009. URL http://kops.ub.uni-konstanz.de/volltexte/2009/9188/.
[5] H. Aljazzar, S. Leue, Generation of counterexamples for model checking of Markov decision processes, in: 6th International Conference on the Quantitative Evaluation of Systems (QEST '09), IEEE Computer Society Press, 2009, pp. 197–206.

[6] E. de Queirós Vieira Martins, J.L.E. dos Santos, A new shortest paths ranking algorithm, Tech. rep., Departamento de Matemática, Universidade de Coimbra, Portugal, July 1999, URL: http://www.mat.uc.pt/~eqvm.
[7] V.M. Jiménez, A. Marzal, Computing the k shortest paths: A new algorithm and an experimental comparison, in: 3rd International Workshop on Algorithm Engineering (WAE '99), in: Lecture Notes in Computer Science, vol. 1668, Springer, 1999, pp. 15–29.
[8] V.M. Jiménez, A. Marzal, A lazy version of Eppstein's shortest paths algorithm, in: 2nd International Workshop on Experimental and Efficient Algorithms (WEA '03), in: Lecture Notes in Computer Science, vol. 2647, Springer, 2003, pp. 179–190.
[9] L. Galand, P. Perny, Search for compromise solutions in multi-objective state space graphs, in: 17th European Conference on Artificial Intelligence (ECAI'06), in: Frontiers in Artificial Intelligence and Applications, vol. 141, IOS Press, 2006, pp. 93–97.
[10] A. Pauls, D. Klein, K-best A* parsing, in: 46th Annual Meeting of the Association for Computational Linguistics (ACL '09), ACL-IJCNLP '09, Association for Computational Linguistics, 2009, pp. 958–966.
[11] H. Aljazzar, H. Hermanns, S. Leue, Counterexamples for timed probabilistic reachability, in: 3rd International Conference of Formal Modeling and Analysis of Timed Systems (FORMATS '05), in: Lecture Notes in Computer Science, vol. 3829, Springer, 2005, pp. 177–195.
[12] H. Aljazzar, S. Leue, Directed explicit state-space search in the generation of counterexamples for stochastic model checking, IEEE Transaction on Software Engineering 36 (1) (2010) 37–60.
[13] H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner, S. Leue, Safety analysis of an airbag system using probabilistic FMEA and probabilistic counter examples, in: 6th International Conference on the Quantitative Evaluation of Systems (QEST '09), IEEE Computer Society Press, 2009, pp. 299–308.
[14] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1 (1959) 269–271.
[15] J. Pearl, Heuristics – Intelligent Search Strategies for Computer Problem Solving, Addision–Wesley, 1986.
[16] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality of A*, J. ACM 32 (3) (1985) 505–536.
[17] G.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, IEEE, 1991, pp. 632–641.
[18] G.N. Frederickson, An optimal algorithm for selection in a min-heap, Information and Computation 104 (2) (1993) 197–214.
[19] P. Sanders, D. Schultes, Engineering fast route planning algorithms, in: 6th International Workshop on Experimental Algorithms (WEA '07), in: Lecture Notes in Computer Science, vol. 4525, Springer, 2007, pp. 23–36.
[20] J.Y. Yen, Finding the k shortest loopless paths in a network, Management Science 17 (11) (1971) 712–716.
[21] The 9th DIMACS implementation challenge: The shortest path problem, Department of Computer and System Sciences Antonio Ruberti, Sapienza University of Rome, http://www.dis.uniroma1.it/~challenge9/, 2006.
[22] B.R. Haverkort, H. Hermanns, J.-P. Katoen, On the use of model checking techniques for dependability evaluation, in: 19th IEEE Symposium on Reliable Distributed Systems (SRDS 2000), 2000, pp. 228–237.
[23] A. Hinton, M.Z. Kwiatkowska, G. Norman, D. Parker, PRISM: A tool for automatic verification of probabilistic systems, in: 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06), in: Lecture Notes in Computer Science, vol. 3920, Springer, 2006, pp. 441–444.