K*: A Directed On-The-Fly Algorithm for Finding the k Shortest Paths

Husain Aljazzar and **Stefan Leue**
Department of Computer and Information Science
University of Konstanz, Germany
{Husain.Aljazzar,Stefan.Leue}@uni-konstanz.de

Abstract

We present a new algorithm, called K*, for finding the k shortest paths between a designated pair of vertices in a given directed weighted graph. Compared to Eppstein's algorithm, which is the most prominent algorithm for solving this problem, K* has two advantages. First, K* performs on-the-fly, which means that it does not require the graph to be explicitly available and stored in main memory. Portions of the graph will be generated as needed. Second, K* is a directed algorithm which enables the use of heuristic functions to guide the search. This leads to significant improvements in the memory and runtime demands for many practical problem instances. We prove the correctness of K* and show that it maintains a worst-case runtime complexity of $\mathcal{O}(m+k\,n\,log(k\,n))$ and a space complexity of $\mathcal{O}(k\,n+m)$, where n is the number of vertices and m is the number of edges of the graph. We provide experimental results which illustrate the scalability of the algorithm.

Introduction

In this paper we consider the *k-Shortest-Paths* problem (KSP) which is about finding the k shortest paths in a directed weighted graph G for an arbitrary natural number k. In other words, for a pair of designated vertices $\mathbf{s}$ and $\mathbf{t}$ in a given digraph G, we aim at enumerating the paths from $\mathbf{s}$ to $\mathbf{t}$ in a non-increasing order with respect to their length. Application domain examples for KSP problems include computer chess, sequence alignment and probabilistic model checking. Quite a few approaches to solve the KSP problem or particular variants of it have been published. With respect to the worst-case runtime complexity the most advantageous approach is the algorithm presented by Eppstein (Eppstein 1998). A lazy version of Eppstein's algorithm, also referred to as *Lazy Eppstein*, has been presented in (Jiménez & Marzal 2003). It improves the practical performance of Eppstein's algorithm in terms of runtime and memory consumption. Both algorithms require the graph to be completely available when the search starts. They also require that in the beginning an exhaustive search must be performed on G in order to determine the shortest path from every vertex to $\mathbf{t}$. This is a major performance drawback, in particular if the graph G is large.

In order to solve the KSP problem more efficiently when the availability of memory is constrained, we propose an algorithm called K*. It is inspired by Eppstein's algorithm. For a graph with n vertices and m edges, the asymptotic worst-case runtime complexity of K* is $\mathcal{O}(m + k\,n\,log(k\,n))$. This is actually worse than the worst-case complexity of Eppstein's algorithm, which is $\mathcal{O}(m + n\,log(n) + k\,log(k))$. However, K* achieves the following major advantages over the existing KSP algorithms cited above:

- K* performs on-the-fly in the sense that it does not require the graph to be explicitly available, i.e., to be stored in main memory. The graph is partially generated as the need arises.
- K* takes advantage of heuristics-guided search, which often leads to significant improvements in terms of memory and runtime effort.

For Eppstein's algorithm we determined the worst-case space complexity to be $\mathcal{O}(n^2+m+k)$ (c.f. the supplemental materials), whereas K* possesses a space complexity of $\mathcal{O}(k\,n+m)$. Since n is usually significantly greater than k, we consider this an improvement.

Preliminaries

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

The Shortest-Path Problem (SP)

In many application domains one is interested in finding a shortest $\mathbf{s}$-$\mathbf{t}$ path, i.e., a path $\pi^* \in \Pi(\mathbf{s},\mathbf{t})$ with $l(\pi^*) =$

$d^*(\mathbf{s}, \mathbf{t})$. This is commonly referred to as the *Shortest-Path* problem (SP).

Dijkstra's algorithm is the most prominent algorithm for solving the SP problem (Dijkstra 1959). Dijkstra's algorithm stores vertices on the search front in a priority queue open which is ordered according a distance function d. Initially, open contains only the start vertex $\mathbf{s}$ with $d(\mathbf{s}) = 0$. In each search iteration, the head of the queue open, say u, is removed from the queue and *expanded*. More precisely, for each successor vertex v of u, if v has not been visited before, then $d(v)$ is set to $d(u) + w(u, v)$ and v is put into open. If v has been visited before, then $d(v)$ is set to smaller distance of the old $d(v)$ and $d(u) + w(u, v)$. We distinguish between two sets of *visited* vertices, namely *Closed* and *Open* vertices. Closed vertices are those which have been visited and expanded, where as open vertices are those which have been visited but have not yet been expanded, i.e., vertices in the search queue. For each visited vertex v, $d(v)$ is always equal to the length of the shortest path from $\mathbf{s}$ to v that has been discovered so far. The set of these paths forms a *search tree* T. Dijkstra's algorithm ensures that for each **closed** vertex v it holds that $d(v) = d^*(\mathbf{s}, v)$ which means that the path selected in the search tree T 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 **s**-**t** path is found as soon as $\mathbf{t}$ is closed, i.e., selected for expansion. In order to retrieve the selected shortest path to some vertex the structure of T is needed, and hence a link $T(v)$ is attached to each visited vertex v referring to the parent of v in T. The retrieved path can be constructing by backtracking these T references to $\mathbf{s}$.

On-The-Fly: In many application domains the graph G is not explicitly given. Only an implicit description of G is available where a function $succ : V \rightarrow \mathfrak{P}(V)$ returns for each vertex u an explicit representation of the set consisting of its successor vertices, i.e., $succ(u) = \{\ v \in V\ |(u, v) \in E\ \}$. Search algorithms which can directly perform the search on the implicit representation of G as given through *succ* are called *on-the-fly*. Dijkstra's algorithm can be modified to follow an on-the-fly strategy using an additional set closed, which is usually implemented as a hash table, to save closed vertices.

A*. The A* algorithm (Pearl 1986) is derived from the on-the-fly version of Dijkstra by exploiting additional graph or problem structure information in order to determine the order in which vertices should be visited. In A*, the open queue is sorted by a *heuristic evaluation function* f which indicates the desirability of expanding a vertex. For each vertex v, $f(v)$ is defined as $d(v) + h(v)$, where $d(v)$ is as defined in Dijkstra's algorithm. $h(v)$ denotes a heuristic estimate of $d^*(v, \mathbf{t})$ which is the shortest distance from v to the target vertex $\mathbf{t}$. 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 a target node exists at all. We can convince ourselves easily that $f(v) = d(v) + h(v)$ is an estimate of the length of the shortest **s**-**t** path going through v. The heuristic function h is called *admissible* if it is optimistic, i.e., if $h(v) \leq d^*(v, \mathbf{t})$. h is called *monotone* or *consistent* if for each edge (u, v) in G it holds that $h(u) \geq w(u, v) + h(v)$. It can be easily proven that every monotone heuristic is also admissible. Using an admissible heuristic guaranties the optimality of A*, i.e., that a shortest **s**-**t** path will be found. If h is monotone, then A* ensures that the search tree T is a shortest path tree for all closed vertices. Vertices are expanded by A* in a non-decreasing order regarding their f-values in case the heuristic is monotone. If $h(v) = 0$ for all v, then A* behaves exactly like Dijkstra.

The k-Shortest-Paths Problem (KSP)

When solving the KSP problem one is interested in enumerating the **s**-**t** paths in G in a non-decreasing order with respect to their lengths.

Eppstein's Algorithm finds the k shortest paths in a given directed graph $G = (V, E)$. It first applies Dijkstra's algorithm on G in a backwards manner, i.e., starting from $\mathbf{t}$ following edges backwards. The result is a "reversed" shortest path tree T rooted at $\mathbf{t}$, i.e., the shortest path from any vertex in G to $\mathbf{t}$. Then, a special data structure called *path graph* $\mathcal{P}(G)$ is used to save all paths through G using an implicit representation. Finally, the k shortest paths are delivered by applying Dijkstra search on $\mathcal{P}(G)$. A central notion in our approach that we inherit from Eppstein's algorithm is that of a *sidetrack edge*: an edge $(u, v) \in E$ either belongs to the shortest path tree T, in which case we call it a tree edge, otherwise we call it a sidetrack edge. As Eppstein shows, any **s**-**t** path π can be uniquely described by the subsequence of sidetrack edges taken in π, denoted as $\alpha(\pi)$. The notion of a sidetrack edge is also interesting because any such edge $(u, v) \in G - T$ may represent the fact that a certain detour compared to the shortest path is taken. Sidetrack edges are therefore closely related to the notion of opportunity costs for choosing an alternate path to some given **s**-**t** path.

The asymptotic runtime complexity of Eppstein's algorithm in worst-case is $\mathcal{O}(m + n\,log(n) + k\,log(k))$. Unfortunately, the memory space complexity has not been declared. We computed for Eppstein's algorithm a worst-case space complexity of $\mathcal{O}(n^2 + m + k)$. The main point in our computation is that $\mathcal{P}(G)$ contains amongst others n tree heaps. Each heap may contain up to n elements. Hence we obtain a complexity of $\mathcal{O}(n^2)$. To store the explicitly given graph G we need $\mathcal{O}(m+n)$ memory locations. In order to find k shortest **s**-**t** paths the algorithm visits k nodes, which induces a complexity of $\mathcal{O}(k)$. This gives a total complexity of $\mathcal{O}(n^2 + m + k)$.

The K* Algorithm

The design of K* is inspired by Eppstein's algorithm. In K* we determine a shortest path tree T of G and use a graph structure $\mathcal{P}(G)$ which, as in Eppstein's algorithm, is searched using Dijkstra to determine **s**-**t** paths in the form of sidetrack edge sequences. However, as mentioned before, K* is designed to perform on-the-fly and to be guided by heuristics. The following are the main design ideas for K*:

1. We apply A* on G instead of the backwards Dijkstra construction in Eppstein's algorithm in order to determine the shortest path tree T.

2. We concurrently execute A* on G and Dijkstra on $\mathcal{P}(G)$. Consequently, Dijkstra will be able to deliver solution paths before G is completely searched by A*.

In order to accommodate this design we have to make fundamental changes compared to Eppstein's algorithm to the structure of $\mathcal{P}(G)$.

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

Path Graph Structure. $\mathcal{P}(G)$ will be incrementally constructed by adding the edges as they are discovered by A*. For an edge (u, v), $\delta(u, v)$ indicates the disadvantage of taking this edge as a detour compared to the shortest **s**-**t** path via v. Both values are usually not known when using an on-the-fly search. They can only be estimated using the evaluation function f. Let $f(v)$ be the f-value of v according the search tree T and $f_u(v)$ be the f-value of v according to the parent u, i.e., $f_u(v) = d(u) + w(u, v) + h(v)$. $\delta(u, v)$ is then defined as:

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

The following lemma, which was originally proposed by Eppstein, shows that the length of an **s**-**t** path can be computed using the δ function.

Lemma 1. *For any* **s**-**t** *path* π*, it holds that*

$$l(\pi) = d^*(\mathbf{s},\mathbf{t}) + \sum_{e\in\alpha(\pi)} \delta(e) = d^*(\mathbf{s},\mathbf{t}) + \sum_{e\in\pi} \delta(e).$$

Proof. Let π be $v_0 \to \ldots \to v_n$ with $v_0 = \mathbf{s}$ and $v_n = \mathbf{t}$. Then, consider

$$\begin{aligned} \sum_{e\in\alpha(\pi)} \delta(e) &= \sum_{e\in\pi} \delta(e) \\ &= \sum_{i=0}^{n-1} \delta(v_i, v_{i+1}) \\ &= \sum_{i=0}^{n-1} d(v_i) + w(v_i, v_{i+1}) - d(v_{i+1}) \\ &= d(v_0) + \sum_{i=0}^{n-1} w(v_i, v_{i+1}) - d(v_n) \\ &= d(\mathbf{s}) + \sum_{i=0}^{n-1} w(v_i, v_{i+1}) - d(\mathbf{t}) \end{aligned}$$

Presuming that h is admissible, it holds that $d(\mathbf{t}) = d^*(\mathbf{s}, \mathbf{t})$. Further, it holds that $d(\mathbf{s}) = 0$. Then, we get that

$$\begin{aligned} d^*(\mathbf{s},\mathbf{t}) + \sum_{e\in\alpha(\pi)} \delta(e) &= d^*(\mathbf{s},\mathbf{t}) + \sum_{e\in\pi} \delta(e) \\ &= \sum_{i=0}^{n-1} w(v_i, v_{i+1}) = l(\pi). \end{aligned}$$

□

For each visited vertex v, a min heap structure $H_{in}(v)$ is created in order to build $\mathcal{P}(G)$. $H_{in}(v)$ contains a node for every incoming edge of v discovered so far. The nodes of $H_{in}(v)$ will be ordered according the δ-values of the corresponding edges such that the node possessing the edge with minimal detour is on the top of the heap. We refer to the root of $H_{in}(v)$ as $R_{in}(v)$. When adding a new edge (u, v) into $\mathcal{P}(G)$, it is determined whether $H_{in}(v)$ exists. This is not the case in case v is a new vertex. An empty heap is then created and assigned to v and (u, v) will be inserted into $H_{in}(v)$. The construction ensures that the top node of $H_{in}(v)$ is the tree edge of v. For each node n of $H_{in}(v)$ carrying an edge (u, v), a pointer referring to $R_{in}(u)$ is attached to n. We call such pointers *cross edges*, whereas the pointers which reflect the heap structuring are called *heap edges*. The derived structure $\mathcal{P}(G)$ is a directed graph, the vertices of which correspond to edges in G. An arbitrary path $\sigma = n_0 \to \ldots \to n_r$ through $\mathcal{P}(G)$ which starts at $R_{in}(\mathbf{t})$, i.e., $n_0 = R_{in}(\mathbf{t})$ can be interpreted as a recipe for constructing a **s**-**t** path. Each heap edge (n_i, n_{i+1}) in σ represents the decision to take the incoming edge associated with the node n_{i+1} instead of the one associated with n. The move via a cross edge (n_i, n_{i+1}), where n_i corresponds to an edge (u, v), to the heap $H_{in}(u)$ with root n_{i+1} represents the selection of (u, v) as an incoming edge. Based on this interpretation we derive from σ a sequence of edges $seq(\sigma)$ using the following procedure. For each cross edge (n_i, n_{i+1}) in σ we add to $seq(\sigma)$ the edge associated with n_i. Finally, we add to $seq(\sigma)$ the edge associated with the last node of σ, i.e., n_r. We obtain the **s**-**t** path from $seq(\sigma)$ after completing it with the possibly missing tree edges up to **s**. We recall that any **s**-**t** path is fully characterized by the subsequence of sidetrack edges it takes. Thus, the presence or absence of tree edges in $seq(\sigma)$ does not influence the construction of the **s**-**t** path. For simplicity we hence assume, w.l.o.g., that $seq(\sigma)$ contains only

sidetrack edges. The structure of $\mathcal{P}(G)$ ensures that this procedure results in a valid **s**-**t** path:

Lemma 2. *For an arbitrary path σ through $\mathcal{P}(G)$ starting at $R_{in}(\mathbf{t})$ it holds that $\exists\, \pi \in \Pi(\mathbf{s},\mathbf{t}) : \pi = \beta(seq(\sigma))$.*

Proof. W.l.o.g., we assume that $seq(\sigma)$ contains only sidetrack edges. We build a path π as follows. We begin with the single vertex $\mathbf{t}$, i.e., $\pi = \mathbf{t}$. Let (u, v) be the last edge in $seq(\sigma)$. Then, v is either equal to $\mathbf{t}$ or there is a path in the shortest path tree T leading form v to $\mathbf{t}$. Otherwise, there would be no way to get from $R_{in}(\mathbf{t})$ to $R_{in}(v)$ without sidetrack edges in which case (u, v) can not be the last element in $seq(\sigma)$. Then, there is a unique way of prepending tree edges to π until v is reached, i.e., until $first(\pi)$ is equal to v. We prepend the edge (u, v) to π. Further, for each successive pair of edges (u_1, v_1) and (u_2, v_2) we can argue in a similar way as before that there is a path in T from v_1 to u_2. We prepend the edges of this tree paths followed by the sidetrack edge (u_1, v_1). We repeat this step until all edges from $seq(\sigma)$ are handled. Afterwards, we repeatedly prepend the tree edge of $first(\pi)$ until the start vertex $\mathbf{s}$ is reached, i.e., $first(\pi) = \mathbf{s}$. At the end, the constructed path π is a path from $\mathbf{s}$ to $\mathbf{t}$ using no side edges but the ones from $seq(\sigma)$. This means, the result is a solution path π such that $\alpha(\pi) = seq(\sigma)$. Consequently it holds that $\pi = \beta(seq(\sigma))$. □

We now define a length function Δ on the edges of $\mathcal{P}(G)$. Let (n, n') denote an edge in $\mathcal{P}(G)$, and let e and e' denote the corresponding edges from G. Then we define $\Delta(n, n')$ as follows:

$$\Delta(n,n') = \begin{cases} \delta(e') - \delta(e), & (n,n') \text{ is a heap edge} \\ \delta(e') = 0, & (n,n') \text{ is a cross edge} \end{cases} \quad (2)$$

This means that $l(\sigma)$, i.e., the length of σ, is equal to $\sum_{e\in\sigma} \Delta(e)$. Note that Δ gives zero for all cross edges. We can also infer from the definition of Δ that the length of any sequence of heap edges is equal to the δ-value of the edge associated with the target node of the last edge in the sequence. Note that this is the edge which will be added to $seq(\sigma)$, if this sequence is a part of the path σ. Consequently, we can deduce that $l(\sigma) = \sum_{e\in seq(\sigma)} \delta(e)$. Further, let π be the **s**-**t** path derived from σ, i.e., $\pi = \beta(seq(\sigma))$. Note, that π exists due to Lemma 2. From Lemma 1 we know that $l(\pi) = d^*(\mathbf{s},\mathbf{t}) + \sum_{e\in\alpha(\pi)} \delta(e)$ which is equal to $d^*(\mathbf{s},\mathbf{t}) + \sum_{e\in seq(\sigma)} \delta(e)$.

Lemma 3. *Let π be the path obtained from a $\mathcal{P}(G)$ path σ, i.e., $\pi = \beta(seq(\sigma))$. Then, it holds that*

$$l(\pi) = d^*(\mathbf{s},\mathbf{t}) + l(\sigma) = d^*(\mathbf{s},\mathbf{t}) + \sum_{e\in seq(\sigma)} \delta(e).$$

Proof. Let $\sigma = n_0 \to \ldots \to n_r$ be a path starting at $R_{in}(\mathbf{t})$. We consider the subsequences $\sigma_0, \ldots, \sigma_l$ which we get by splitting σ at cross edges. Note that each σ_i contains 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. Note that $seq(\sigma)$ is equal to the edge sequence $\langle e_0, \ldots, e_l\rangle$. Together with the fact that Δ is zero for all cross edges, we imply that

$$l(\sigma) = \sum_{i=0}^{r-1} \Delta(n_i, n_{i+1}) = \sum_{e\in seq(\sigma)} \delta(e).$$

Now, let π be the **s**-**t** path obtained from σ, i.e., $\pi = \beta(seq(\sigma))$. It holds then, assuming that $seq(\sigma)$ consists of sidetrack edges, that $\alpha(\pi) = seq(\sigma)$. Due to Lemma 1, it holds then that

$$\begin{aligned} l(\pi) &= d^*(\mathbf{s},\mathbf{t}) + \sum_{e\in\alpha(\pi)} \delta(e) \\ &= d^*(\mathbf{s},\mathbf{t}) + \sum_{e\in seq(\sigma)} \delta(e) \\ &= d^*(\mathbf{s},\mathbf{t}) + l(\sigma). \end{aligned}$$

□

This means that the length of σ is equal to the distance penalty of π compared to the shortest **s**-**t** path π^*.

The importance of the Lemma 3 lies in the fact that it establishes a correlation between the length of a path through $\mathcal{P}(G)$ and the length of the corresponding **s**-**t** path in the problem graph G. We now know that shorter $\mathcal{P}(G)$ paths lead to shorter **s**-**t** paths. This property enables the use of a Dijkstra shortest path search on $\mathcal{P}(G)$ starting at $R_{in}(\mathbf{t})$ in order to compute the shortest **s**-**t** paths.

The Algorithmic Structure of K*

Algorithm 1 contains the pseudocode for K*. The code in lines 10 to 17 represent the A* search on G, whereas lines 18 to 28 define the Dijkstra search on $\mathcal{P}(G)$. The **if**-statement starting at line 6 is responsible for scheduling the concurrent interleaving of both algorithms.

As can be seen from lines 11 to 5, the version of A* used here does not terminate when $\mathbf{t}$ is selected for expansion, i.e., a shortest **s**-**t** path π^* is found. It simply adds an empty sidetrack edge sequence representing π^* to $\mathcal{R}$ and continues with the search. Note that the found **s**-**t** path is completely contained in the search tree T and, consequently, consists only of tree edges. Then, A* initializes the Dijkstra search to run on $\mathcal{P}(G)$ by adding $R_{in}(\mathbf{t})$ into open_D. From this point on Dijkstra will participate in the scheduling competition, c.f., Line 6, to search the graph $\mathcal{P}(G)$ for solution paths.

The lines from 18 to 22 represent the usual Dijkstra search steps. Note that when an arbitrary node n is visited K* does not check whether n was visited before. In other words, every time a node is visited it is considered as a new node. This strategy is justified by the observation that an **s**-**t** path may take the same edge several times. The if-statement starting at Line 23 comprises the step needed to deliver the next shortest **s**-**t** path. This is done by constructing the edge sequence $seq(\sigma)$

from the path σ via which Dijkstra reached the node n which has just been expanded. Note that we check whether n corresponds to a sidetrack edge or not. As we will show later, this condition is necessary to prevent duplicates in the result, i.e., delivering the same **s-t** path more than once.

Algorithm 1: The K* Algorithm

```
1  open ← empty priority queue, closed ← empty hash table;
2  open_D ← empty priority queue, closed_D ← empty hash table;
3  P(G) ← empty path graph, R ← empty list;
4  Insert s into open ;
5  if open and open_D are empty then Return R and exit;
6  if open_D is not empty then
7      if open is empty then Go to Line 18;
8      Let u be the head of open and n the head of open_D;
9      if d*(s,t) + d(n) ≤ f(u) then Go to Line 18;
10 Remove from open and place on closed the vertex v with the
   minimal f-value;
11 if v = t then
12     Insert an empty sidetrack edge sequence into R;
13     Insert R_in(v) into open_D;
14 else
15     Expand v ;
16     Insert all edges outgoing from v into P(G);
17 Go to Line 5;
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     Set d(n') = d(n) + Δ(n,n') ;
21     Attach to n' a parent link referring to n ;
22     Insert n' into open_D ;
23 if The edge associated with n is a sidetrack edge then
24     Let σ be the path in P(G) via which n was reached;
25     Add seq(σ) at the end of R;
26     if R contains k or more elements then
27         Return R and exit;
28 Go to Line 5;
```

The fact that both algorithms A* and Dijkstra share the path graph $\mathcal{P}(G)$ may give rise to concern regarding the correctness of the Dijkstra search on $\mathcal{P}(G)$. Dijkstra's algorithm runs on $\mathcal{P}(G)$ while A* is still adding new nodes into it. For instance, a node m might be added to $\mathcal{P}(G)$ after Dijkstra has already expanded its parent n. In this case, the edge (n, m) will not be captured. In order to be able to handle this issue we need to study the scheduling mechanism on line 6, and its consequences in more details. We can see that Dijkstra is only resumed if and only if its search queue is not empty and it holds that $d^*(\mathbf{s}, \mathbf{t}) + d(n) \leq f(v)$, where n is the node on the head of Dijkstra's queue open_D and v is the vertex on the head of the search queue of A*. Using this scheduling mechanism K* assures that adding new nodes into $\mathcal{P}(G)$ during the search does not disturb the order in which nodes are expanded. This insight is captured by the following lemma.

Lemma 4. *The d-value of any node m, which is added into $\mathcal{P}(G)$ after expanding a node n, is never better than $d(n)$.*

Proof. Let σ be the shortest path leading to m through $\mathcal{P}(G)$. Then, the best d-value of m is equal to $l(\sigma)$. Consequently, according to Lemma 3 it holds that $d^*(\mathbf{s}, \mathbf{t}) + d(m) = l(\pi)$, where $\pi = \beta(seq(\sigma))$. Now, let $(u, v) \in E$ be the edge associated to m. The admissibility of f assures that $f(u) \leq l(\pi)$. Thus it holds that $f(u) \leq d^*(\mathbf{s}, \mathbf{t}) + d(m)$. Moreover, by construction, m will be added into $\mathcal{P}(G)$ on expanding the vertex u. Because n was expanded before, it must hold, according to our scheduling mechanism, that $d^*(\mathbf{s}, \mathbf{t}) + d(n) \leq f(u)$. Together, we get $d^*(\mathbf{s}, \mathbf{t}) + d(n) \leq f(u) \leq d^*(\mathbf{s}, \mathbf{t}) + d(m)$ which means that $d(n) \leq d(m)$. □

The lemma implies that when an **s-t** path is delivered, it is not possible afterwards that A* discovers new edges which lead to a shorter path. We can hence assume that K* delivers the next solution path only if it is really the next shortest **s-t** path.

We now turn to the issue of Dijkstra's correctness on the dynamically growing graph $\mathcal{P}(G)$. The only update operation on $\mathcal{P}(G)$ is that of adding a node. Let (u, v) be an edge which has just been discovered by A* and a corresponding node n' is added into $\mathcal{P}(G)$ at Line 16. Then, n' is inserted into the heap $H_{in}(v)$. For any new edge (u, v) the heap structure of $H_{in}(v)$ stays intact even if the f-value of v has changed since the δ-values of all nodes of $H_{in}(v)$ are all changed by the same amount. Hence, a global restructuring of $H_{in}(v)$ is never needed and it suffices to consider the case of heaping up the new node n'. We need to consider the following two cases when heaping up n'.

- The easy case is that Dijkstra has not already expanded any direct predecessor of n'. The absence of n' did then not influence the previous search at all. No matter whether any of the direct predecessors of n' has been visited or not, n' is completely unknown for the algorithm even if the complete $\mathcal{P}(G)$ were available.
- The more involved case is if at least one direct predecessor n of n' that has been expanded at least once. In this case the siblings of n' have been visited when n' has not existed in $\mathcal{P}(G)$ yet. We then have to catch up what was missed during the search because of the absence of n'. We do this by applying the lines from 20 to 22 to n' for each expanded direct predecessor n of n'. Consequently, n' will be inserted into the search queue open_D. Lemma 4 ensures that the best $d(n')$ is not better than the d-values of all nodes which have been expanded before. This means that we did not miss out on expanding n'. Moreover, let n'' be the last node pushed down while n' was heaping up. Before n' has been added, n'' was a direct successor of n. Hence, n'' has been visited like all of its siblings. Currently, n'' is a successor of n'. However, n' has not yet been expanded and hence, n'' should not have been visited yet. This expansion therefore needs to

be undone. Note that n'' has been heaped down by n' which means that the edge associated with n' has a smaller δ-value than the one associated with n''. This implies that it always holds that $d(n') < d(n'')$. We can therefore be sure that n'' has not been expanded yet, see Lemma 4. We can hence assume that n'' is still in the queue open_D and only need to remove it from there.

K* performs these operations upon adding every node into $\mathcal{P}(G)$ at Line 16. This maintains the correctness of the Dijkstra search on $\mathcal{P}(G)$ in K*.

Correctness

Termination of K* for finite k and G can easily be derived from the termination of A* and Dijkstra. We now turn to the question of partial correctness.

Theorem 1. *For a weighted directed graph $G = (V, E)$, a start vertex $\mathbf{s}$ and a target vertex $\mathbf{t}$, the K* algorithm solves the KSP problem.*

In order to prove the previous theorem we show that K* finds k shortest **s**-**t** paths for any $k \in \mathbb{N}$ with $k \leq |\Pi(\mathbf{s}, \mathbf{t})|$. Our proof strategy is to verify the following points for the result list $\mathcal{R}$.

- **Point one**: Each element from $\mathcal{R}$ is a sequence of edges which represents a valid **s**-**t** path in the problem graph G.
- **Point two**: All **s**-**t** paths obtained from $\mathcal{R}$ are pairwise distinct.
- **Point three**: For each **s**-**t** path π in G, an edge sequence representing π will be added into $\mathcal{R}$ for some k (which is big enough).
- **Point four**: For any point in the search, each solution path, i.e., **s**-**t** path obtained from an edge sequence from $\mathcal{R}$, is at least as short as any **s**-**t** path which is not represented in $\mathcal{R}$ yet.

The first three points ensure that $\mathcal{R}$ enumerates k different **s**-**t** paths in a non-decreasing order with respect to their lengths. Point four implies that the found paths are the shortest ones.

Point one is a direct inference of Lemma 2 which assures that each path in $\mathcal{P}(G)$ starting at $R_{in}(\mathbf{t})$ induces a valid **s**-**t** path. Due to the structure of $\mathcal{P}(G)$, two different paths through $\mathcal{P}(G)$ can induce the same **s**-**t** path in G. This makes the proof of **point two** a bit more involved. Note that K* adds the edge sequence of a $\mathcal{P}(G)$ only if it ends with a sidetrack edge, see Line 23 in Algorithm 1. As the following lemma ensures, such $\mathcal{P}(G)$ paths lead to unique **s**-**t** paths. Consequently, all paths represented in $\mathcal{R}$ are distinct.

Lemma 5. *Let σ and σ' be two different paths in $\mathcal{P}(G)$ such that both σ and σ' end with nodes corresponding to a sidetrack edge. It then holds that $\beta(seq(\sigma)) \neq \beta(seq(\sigma'))$.*

Proof. The Lemma is established if we can show that σ and σ' induce two different sidetrack sequences. The idea of the proof 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' and let (u, v) be the edge which corresponds to m. We distinguish between the following two cases. Again we assume here that tree edges are excluded from $seq(\sigma)$ and $seq(\sigma')$.

1. One of the paths ends at m. W.l.o.g. let σ' end at m. Then, σ has a postfix after m. Let n be the next node after m. If (m, n) is a cross edge, then (u, v) will be added into $seq(\sigma)$. Note that σ ends with a sidetrack edge which will also 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 $H_{in}(v)$ at another node which, because of the tree structure of $H_{in}(v)$, is not m. Consequently, $seq(\sigma) \neq seq(\sigma')$ holds in this case too.
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'). Again, we distinguish between two cases:
   (a) First, we consider the case that both edges (m, n) and (m, n') are heap edges. In this case, because of the tree structure of $H_{in}(v)$, the last nodes touched by σ and σ' in the heap $H_{in}(v)$ must differ from each other. Thus, it holds that $seq(\sigma) \neq seq(\sigma')$.
   (b) Now, we assume that one edge, say (m, n) w.l.o.g., is a cross edge. Then, m is the last node touched by σ in $H_{in}(v)$. Further, (m, n') must be a heap edge because m has, by construction, at most one outgoing cross edge. Again because of the tree structure of $H_{in}(v)$, the last node touched by σ' in $H_{in}(v)$ is different from m. If (u, v) is a sidetrack edge, then (u, v) will be the next sidetrack edge added to $seq(\sigma)$ but not to $seq(\sigma')$. This means that $seq(\sigma) \neq seq(\sigma')$. In the case that (u, v) is a tree edge, i.e., m is the root of $H_{in}(v)$, no sidetrack edge is added to $seq(\sigma)$ here. Moreover, because the search tree T does not contain cycles, it is not possible that σ enters the heap $H_{in}(v)$ again without touching at least one sidetrack edge in-between which will be added to $seq(\sigma)$. It is hence ensured that the next sidetrack edge added to $seq(\sigma)$ differs from the next one added to $seq(\sigma)$. Consequently, it holds that $seq(\sigma) \neq seq(\sigma')$.

□

We can reason about the satisfiability of **point three** as follows. For $k = |\Pi(\mathbf{s}, \mathbf{t})|$, K* would not stop before all **s**-**t** paths are found or both search queues are empty. In the first case the claim trivially holds. In the other case, in particular if open is empty, we know that A* has added all G edges into $\mathcal{P}(G)$. Let π be an arbitrary **s**-**t** path. Since our Dijkstra search is complete on $\mathcal{P}(G)$ we just need to show the existence of a path σ in the

complete $\mathcal{P}(G)$ such that $\beta(seq(\sigma)) = \pi$. The existence of σ can be proven by considering the structure of $\mathcal{P}(G)$.

Lemma 6. *For any path $\pi \in \Pi(\mathbf{s}, \mathbf{t})$, the complete path graph $\mathcal{P}(G)$ contains a path σ starting at $R_{in}(\mathbf{t})$ such that $\pi = \beta(seq(\sigma))$*

Proof. We only need to determine a path σ starting at $R_{in}(\mathbf{t})$ such that $seq(\sigma) = \alpha(\pi)$. We recall that we can assume that $seq(\sigma)$ does not include any tree edges.

In case $\alpha(\pi)$ consists of one side edge (u, v), then we know that there is a path in T from v to $\mathbf{t}$, since π leads to $\mathbf{t}$. Consequently, there is a path p in $\mathcal{P}(G)$ from $R_{in}(\mathbf{t})$ to $R_{in}(v)$ which goes only through the roots of the heaps, i.e., without any sidetrack edges. Further, $H_{in}(v)$ contains a node n corresponding to (u, v) since we assume here that $\mathcal{P}(G)$ is complete. Then, there is a path p' of heap edges in $H_{in}(v)$ from $R_{in}(v)$ to n. Together, $\sigma = pp'$ is a path from $R_{in}(\mathbf{t})$ to n. It trivialy holds that $seq(\sigma) = \alpha(\pi)$.

If $\alpha(\pi) = \langle e_1, \ldots, e_r \rangle$ with $r > 1$, then we can assume by induction over r that $\mathcal{P}(G)$ contains a path q from $R_{in}(\mathbf{t})$ to the node m corresponding to e_2 such that $seq(q) = \langle e_2, \ldots, e_n \rangle$. We write e_2 and e_1 as $e_1 = (u, v)$ and $e_2 = (u', v')$. By definition, there is a path in T from v to u'. Then, as argued before, there is a path p in $\mathcal{P}(G)$ from m to $R_{in}(v)$ touching only tree edges. Further, the node n corresponding to e_1 must be reachable from $R_{in}(v)$ by a path p' through $H_{in}(v)$. Again here, it is trivial to show that $seq(\sigma) = \alpha(\pi)$ where σ is the path $\sigma = qpp'$. □

We now consider **point four**. Dijkstra's algorithm ensures that edge sequences corresponding to shorter $\mathcal{P}(G)$ paths are delivered sooner than those corresponding to longer ones. The claim follows from the correlation due to Lemma 3 between the lengths of $\mathcal{P}(G)$ paths and the associated **s**-**t** paths.

Complexity

Let n be the number of vertices and m be the number of edges in G, i.e., $n = |V|$ and $m = |E|$.

Runtime Complexity. The runtime complexity of K* is determined by A*, the construction of $\mathcal{P}(G)$, and the complexity of finding k paths using Dijkstra on $\mathcal{P}(G)$. The complexity of A* is $\mathcal{O}(m + n\,log(n))$ in the case that the search queue is realized as a Fibonacci heap (Cormen *et al.* 2001) and a monotone heuristic is used. K* will add, in linear time, as many nodes into the Fibonacci-heap based $\mathcal{P}(G)$ as there are edges in G. The complexity of the construction of $\mathcal{P}(G)$ is hence $\mathcal{O}(m)$. Dijkstra iterates as long as the expanded nodes correspond to tree edges and delivers a solution path only when a node corresponding to a sidetrack edge is expanded. There are $(n-1)$ tree edges. Assuming that all vertices in G are reachable from **s**, the k shortest paths will be found within at most $(k\,n)$ iterations. Because each node in $\mathcal{P}(G)$ has at most 3 successors, in $(k\,n)$ search iterations at most $(3\,k\,n)$ nodes will be added into Dijkstra's queue. Heap operations have a logarithmic runtime which leaves us with a complexity of $\mathcal{O}(k\,n\,log(k\,n))$. We obtain the total runtime complexity of $\mathcal{O}(m + n\,log(n) + k\,n\,log(k\,n))$, which is equal to $\mathcal{O}(m + k\,n\,log(k\,n))$, for K*.

Space Complexity: The asymptotic space complexity of K* consists of (1) the space needed for A*, i.e., saving the explored part of the graph G, (2) the space consumed by the data structure of $\mathcal{P}(G)$, and (3) the space needed for the Dijkstra search on $\mathcal{P}(G)$. In the worst case, A* would explore the complete graph G, which results in a space complexity of $\mathcal{O}(n+m)$. $\mathcal{P}(G)$ contains at most m nodes and $\mathcal{O}(m)$ edges. Hence, $\mathcal{P}(G)$ consumes a space of $\mathcal{O}(m)$. As mentioned in the previous paragraph, Dijkstra would visit $\mathcal{O}(k\,n)$ nodes of $\mathcal{P}(G)$ in order to find k shortest paths. That means a space complexity of $\mathcal{O}(k\,n)$. Together, we get for K* a space complexity of $\mathcal{O}(n + m + k\,n)$ which is equal to $\mathcal{O}(k\,n + m)$.

Experimental Evaluation

We implemented K* as well as a variant of K* called Blind K* that does not use a heuristic estimate function. Blind K* represents the situation where no heuristic estimate is available, A* then performs like Dijkstra. We also implemented the lazy version of Eppstein's algorithm. In our experiments we use 5 randomly generated graphs, each of size $|V| = 10^5$. Every node has randomly generated between 1 and 20 outgoing edges with random weights. In order to synthesize heuristic estimates we first computed, for every vertex, the shortest path to **t** using a backwards search starting from **t**. Starting with $h(\mathbf{t}) = 0$, for each vertex u, we computed $h(u)$ by multiplying a random factor from $[0.5, 1]$ with $(w(u, v) + h(v))$, where v is the vertex following u on the shortest u-**t** path. This yields a monotone heuristic. Figures 1 and 2 depict average values for the behavior of the three algorithms over the 5 generated graphs depending on k.

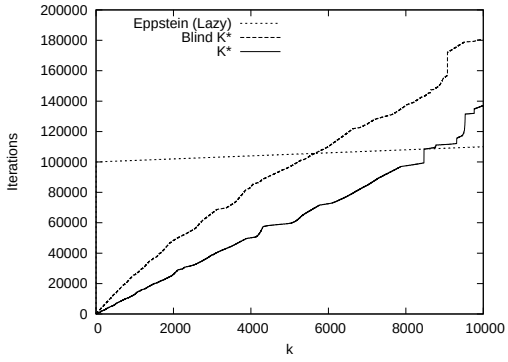

Figure 1: Runtime as measured by iteration count

The experiments show that both in runtime and memory consumption K* dominates Blind K*, which

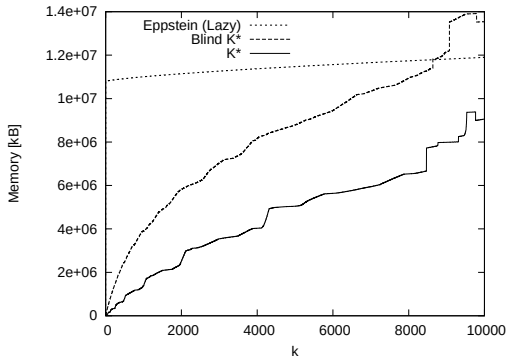

Figure 2: Memory consumption

proves the beneficial effect of using a heuristics guided search. The runtime effort for K* and Blind K* obviously depends on k and offers advantages over Eppstein for the lower k range that we ran experiments on. K* offers advantages over Eppstein when it is not necessary to explore a large portion of the state graph in order to reach k. It should be noticed that K* starts yielding result paths after very few iterations, whereas Eppstein returns results only after the full graph has been searched. This makes K* usable in an on line situation. In terms of memory consumption, both Blind K* and K* outperform Eppstein in the range up to values of k in the multiple thousands. This is largely due the on-the-fly nature of K*.

Conclusion

We have presented K*, an on-the-fly, heuristics guided search algorithm to solve the KSP problem. We have argued for its correctness, analyzed its complexity, and provided evidence that it improves on the algorithm of Eppstein up to fairly sizable numbers for k. Future work addresses the use of K* in various application domains, including stochastic model checking.

References

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2001. *Introduction to algorithms (2nd Ed.)*. The MIT Press.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

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

Jiménez, V. M., and Marzal, A. 2003. A lazy version of eppstein's shortest paths algorithm. In Jansen, K.; Margraf, M.; Mastrolilli, M.; and Rolim, J. D. P., eds., *WEA*, volume 2647 of *Lecture Notes in Computer Science*, 179–190. Springer.

Pearl, J. 1986. *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addision–Wesley.