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

by

Husain Aljazzar and Stefan Leue
The task

- Finding K shortest paths
- Many algorithms – state of art by Eppstein (EA)
- Usually assume entire graph explicitly as input
  - That's why we don't use those algorithms for graphical models ;)

The task

- Finding K shortest paths
- Many algorithms – state of art by Eppstein (EA)
- Usually assume entire graph explicitly as input
  - That's why we don't use those algorithms for graphical models ;)
- Aljazzar and Leue: graph is not explicit, parts of it are generate “on-the-fly” while exploring (using current state and successor function)
  - Which is closer to what we are doing
Very high-level comparison of K* to our m-A*  

**m-A***  
- No explicit graph  
- 1 starting node, **any number of target nodes**  
- Paths found don't have loops  
- Very simple concept  

**K***  
- No explicit graph  
- 1 starting node, 1 **target node**  
- Paths found might have loops  
- More complicated procedure, involves a very fancy datastructure
K* at a glance

- A* is used to explore the problem graph, until the best path to the goal is found – search tree is constructed in the process.
- All sidetrack edges (i.e. not on the best path) are stored in path graph (using a heap-based data structure).
- A path in the path graph (of any length) has a one to one correspondence to a path in the problem graph (and the latter can be easily reconstructed from the former).
- Path graph is searched, using Dijkstra's algorithm.
- It is shown that the shortest path in path graph corresponds to the 2\text{nd} best path in the original graph, 2\text{nd} shortest in the in the path graph to the 3\text{rd} best in the original etc.
- If we don't have k solutions after searching entire path graph – we use A* to further explore original graph and add new sidetrack edges to the path graph. - we keep interleaving A* and Dijkstra.
K* at a glance

- A* is used to explore the problem graph, until the best path to the goal is found – search tree is constructed in the process.
- All sidetrack edges (i.e. not on the best path) are stored in path graph (using a heap-based data structure).
- A path in the path graph (of any length) has a one to one correspondence to a path in the problem graph (and the latter can be easily reconstructed from the former).
- Path graph is searched, using Dijkstra's algorithm.
- It is shown that the shortest path in path graph corresponds to the 2nd best path in the original graph, 2nd shortest in the original graph to the 3rd best in the original etc.
- If we don't have k solutions after searching entire path graph – we use A* to further explore original graph and add new sidetrack edges to the path graph. - we keep interleaving A* and Dijkstra.

Complexity O(m+n log n+k)
And now with more details...
Search tree computed by A* and sidetrack edges

If we have a sidetrack sequence <(s1,s2),(s2,s4)>, we can reconstruct the path s0s1s2s4
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 \(v\)

\[
\delta(u, v) = f_u(v) - f(v) = g(u) + c(u, v) + h(v) - g(v) - h(v) = g(u) + c(u, v) - g(v).
\]
Path graph

- On the high level – it's a graph: nodes represent sidetrack edges of the original graph
- Has very complicated structure – several interconnected heaps, ordered by detour costs

Edges are weighted:

$$\Delta(n, n') = \begin{cases} 
\delta(e) - \delta(e') & \text{if } (n, n') \text{ is a heap edge}, \\
\delta(e') & \text{if } (n, n') \text{ is a cross edge}.
\end{cases}$$
Path graph $P(G)$, properties

- Path graph is a directed weighted graph
- Nodes represent sidetrack edges of the original graph
- Edges are weighted, all weights are non-negative:
  \[
  \Delta(n, n') = \begin{cases} 
  \delta(e') - \delta(e) & \text{if } (n, n') \text{ is a heap edge, and} \\
  \delta(e') & \text{if } (n, n') \text{ is a cross edge.}
  \end{cases}
  \]
- Each node has at most 4 outgoing edges (allows to bound complexity of Dijkstra)
- An arbitrary path $\sigma = n_0 \to \cdots \to n_r$ in $P(G)$ can be interpreted as a recipe for constructing a unique $s$–$t$ path.
- Sidetrack sequence from $P(G)$ always yields a valid $s$–$t$ path.

**Lemma 5.** Let $\sigma$ be a path in $P(G)$ starting at $s$. If $h$ is admissible, then it holds that
\[
C(p(\sigma)) = C^*(s, t) + C_{P(G)}(\sigma).
\]
Let $\sigma$ be the path $(s_2, s_4) \rightarrow (s_1, s_2) \rightarrow (s_3, s_2)$. We derive that $\text{seq}(\sigma) = \langle(s_3, s_2), (s_2, s_4)\rangle$. We construct the corresponding $s_0$–$s_4$ path. We start at $s_4$. The next sidetrack edge in $\text{seq}(\sigma)$ 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 $s_0 \ s_2 \ s_3 \ s_2 \ s_4$. 

---

**Example**

![Diagram of a path](image)

Let $\sigma$ be the path $(s_2, s_4) \rightarrow (s_1, s_2) \rightarrow (s_3, s_2)$. We derive that $\text{seq}(\sigma) = \langle(s_3, s_2), (s_2, s_4)\rangle$. We construct the corresponding $s_0$–$s_4$ path. We start at $s_4$. The next sidetrack edge in $\text{seq}(\sigma)$ 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 $s_0 \ s_2 \ s_3 \ s_2 \ s_4$. 

---

**Diagram**

- **Heap edges** are represented by solid lines.
- **Cross edges** are represented by dotted lines.
Algorithm 1: The K\(^*\) Algorithm

**Data:** A graph given by its start vertex \( s \in V \) and its successor function \( \text{succ} \) and a natural number \( k \)

**Result:** A list \( \mathcal{R} \) containing \( k \) sidetrack edge sequences representing \( k \) solution paths

1. \( \text{open}_D \leftarrow \) empty priority queue.
2. \( \text{closed}_D \leftarrow \) empty hash table.
3. \( \mathcal{R} \leftarrow \) empty list.
4. \( \mathcal{P}(G) \leftarrow \) 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 \( s \) into \( \text{open}_D \).
8. **while** A\(^*\) queue or \( \text{open}_D \) is not empty do
9.   **if** A\(^*\) queue is not empty then
10.     **if** \( \text{open}_D \) is not empty then
11.        Let \( u \) be the head of the search queue of A\(^*\) and \( n \) the head of \( \text{open}_D \).
12.        \( d \leftarrow \max\{ d(n) + \Delta(n, n') \mid n' \in \text{succ}(n) \} \).
13.        **if** \( g(t) + d \leq f(u) \) **then** Go to line 17.
14.        Resume A\(^*\) in order to explore a larger portion of \( G \).
15.        Refresh \( \mathcal{P}(G) \) and bring Dijkstra's search into a consistent status.
16.        Go to line 8.
17.     **if** \( \text{open}_D \) is empty then Go to line 8.
18.     Remove from \( \text{open}_D \) and place on \( \text{closed}_D \) the node \( n \) with the minimal \( d \)-value.
19.     **foreach** \( n' \) referred by \( n \) in \( \mathcal{P}(G) \) do
20.        \( d(n') := d(n) + \Delta(n, n') \)
21.        Attach to \( n' \) a parent link referring to \( n \).
22.        Insert \( n' \) into \( \text{open}_D \).
23.     Let \( \sigma \) be the path in \( \mathcal{P}(G) \) via which \( n \) was reached.
24.     Add \( \text{seq}(\sigma) \) at the end of \( \mathcal{R} \).
25.     **if** \( |\mathcal{R}| = k \) **then** Go to line 26.
26. **Return** \( \mathcal{R} \) and exit.
Example
The result of $K^*$ applied to the graph $G$ from Fig. 7.

<table>
<thead>
<tr>
<th>( \mathcal{P}(G) ) Path</th>
<th>Sidetrack Seq.</th>
<th>( s_0-s_6 ) Path (( \pi ))</th>
<th>( C(\pi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( s_0 )</td>
<td>( () )</td>
<td>( s_0 s_2 s_4 s_6 )</td>
<td>7</td>
</tr>
<tr>
<td>2. ( s_0, (s_4, s_2) )</td>
<td>( (s_4, s_2) )</td>
<td>( s_0 s_2 s_4 s_6 s_6 )</td>
<td>9</td>
</tr>
<tr>
<td>3. ( s_0, (s_4, s_2), (s_1, s_2) )</td>
<td>( (s_1, s_2) )</td>
<td>( s_0 s_1 s_2 s_4 s_6 )</td>
<td>9</td>
</tr>
<tr>
<td>4. ( s_0, (s_4, s_2), (s_1, s_6) )</td>
<td>( (s_1, s_6) )</td>
<td>( s_0 s_1 s_6 )</td>
<td>10</td>
</tr>
<tr>
<td>5. ( s_0, (s_4, s_2), (s_4, s_2) )</td>
<td>( (s_4, s_2), (s_4, s_2) )</td>
<td>( s_0 s_2 s_4 s_2 s_4 s_2 s_4 s_6 )</td>
<td>11</td>
</tr>
<tr>
<td>6. ( s_0, (s_4, s_2), (s_4, s_2), (s_1, s_2) )</td>
<td>( (s_1, s_2), (s_4, s_2) )</td>
<td>( s_0 s_1 s_2 s_4 s_6 )</td>
<td>11</td>
</tr>
<tr>
<td>7. ( s_0, (s_4, s_2), (s_1, s_2), (s_2, s_1) )</td>
<td>( (s_2, s_1), (s_1, s_2) )</td>
<td>( s_0 s_2 s_1 s_2 s_4 s_6 )</td>
<td>12</td>
</tr>
<tr>
<td>8. ( s_0, (s_4, s_2), (s_4, s_2), (s_4, s_2) )</td>
<td>( (s_4, s_2), (s_4, s_2), (s_4, s_2) )</td>
<td>( s_0 s_2 s_4 s_2 s_4 s_2 s_4 s_2 s_4 s_6 )</td>
<td>13</td>
</tr>
<tr>
<td>9. ( s_0, (s_4, s_2), (s_1, s_6), (s_2, s_1) )</td>
<td>( (s_2, s_1), (s_1, s_6) )</td>
<td>( s_0 s_2 s_1 s_6 )</td>
<td>13</td>
</tr>
</tbody>
</table>
Properties of K*

- correct and complete for locally finite graphs
- Always terminates on finite graphs (on infinite – only for bounded k)
- For admissible heuristic – s-t paths are found in non-decreasing order of length
- Time and space complexity $O(m + n \log n + k)$ (m edges, n nodes, k solutions)
Experiments: route planning (in New York city)

264,346 nodes and 733,846 edges
Experiments: route planning (in New York city)

(b) Mean runtime needed for finding the routes. The diagrams on the right-hand side zoom into the interesting segments of the curves.
Experiments: route planning (in New York city)

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

3,598,623 nodes and 8,778,114 edges.

(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 $K^*$. 

Experiments: route planning (Eastern USA)

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

Computing the counterexample for a stochastic model checking 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.
Computing the counterexample for a stochastic model checking problem

(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.
Computing the counterexample for a stochastic model checking problem

(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.