Dijkstra's algorithm

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1956 and published in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).

Dijkstra's original algorithm does not use a min-priority queue and runs in $O(|V|^2)$ (where $|V|$ is the number of vertices). The idea of this algorithm is also given in (Leyzorek et al. 1957). The implementation based on a min-priority queue implemented by a Fibonacci heap and running in $O(|E| + |V| \log |V|)$ (where $|E|$ is the number of edges) is due to (Fredman & Tarjan 1984). This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.

Contents

1 Algorithm
2 Description
3 Pseudocode
4 Running time
5 Related problems and algorithms
6 Dynamic programming perspective
7 See also
8 Notes
9 References
10 External links

Algorithm
Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the unvisited set consisting of all the nodes except the initial node.
3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be $6 + 2 = 8$. If this distance is less than the previously recorded tentative distance of B, then overwrite that distance. Even though a neighbor has been examined, it is not marked as "visited" at this time, and it remains in the unvisited set.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal), then stop. The algorithm has finished.
6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

Description

**Note:** For ease of understanding, this discussion uses the terms intersection, road and map — however, formally these terms are vertex, edge and graph, respectively.

Suppose you would like to find the shortest path between two intersections on a city map, a starting point and a destination. The order is conceptually simple: to start, mark the distance to every intersection on the map with infinity. This is done not to imply there is an infinite distance, but to note that that intersection has not yet been visited; some variants of this method simply leave the intersection unlabeled. Now, at each iteration, select a current intersection. For the first iteration the current intersection will be the starting point and the distance to it (the intersection's label) will be zero. For subsequent iterations (after the first) the current intersection will be the closest unvisited intersection to the starting point—this will be easy to find.

From the current intersection, update the distance to every unvisited intersection that is directly connected to it. This is done by determining the sum of the distance between an unvisited intersection and the value of the current intersection, and relabeling the unvisited intersection with this value if it is less than its current value. In effect, the intersection is relabeled if the path to it through the current intersection is shorter than the previously known paths. To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabeled intersection if you label/relabel it, and erase all others pointing to it. After you have updated the distances to each neighboring intersection, mark the current intersection as visited and select the unvisited intersection with lowest distance (from the starting point) — or lowest label—as the current intersection. Nodes marked as visited are labeled with the shortest path from the starting point to it and will
not be revisited or returned to.

Continue this process of updating the neighboring intersections with the shortest distances, then marking the current intersection as visited and moving onto the closest unvisited intersection until you have marked the destination as visited. Once you have marked the destination as visited (as is the case with any visited intersection) you have determined the shortest path to it, from the starting point, and can trace your way back, following the arrows in reverse.

Of note is the fact that this algorithm makes no attempt to direct "exploration" towards the destination as one might expect. Rather, the sole consideration in determining the next "current" intersection is its distance from the starting point. In some sense, this algorithm "expands outward" from the starting point, iteratively considering every node that is closer in terms of shortest path distance until it reaches the destination. When understood in this way, it is clear how the algorithm necessarily finds the shortest path, however it may also reveal one of the algorithm's weaknesses: its relative slowness in some topologies.

**Pseudocode**

In the following algorithm, the code $u := \text{vertex in } Q \text{ with smallest } \text{dist[}\cdot\text{]}$, searches for the vertex $u$ in the vertex set $Q$ that has the least $\text{dist[}\cdot\text{]}$ value. That vertex is removed from the set $Q$ and returned to the user. $\text{dist_between}(u, v)$ calculates the length between the two neighbor-nodes $u$ and $v$. The variable $\text{alt}$ on lines 20 & 22 is the length of the path from the root node to the neighbor node $v$ if it were to go through $u$. If this path is shorter than the current shortest path recorded for $v$, that current path is replaced with this $\text{alt}$ path. The $\text{previous}$ array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

```plaintext
1 function Dijkstra(Graph, source):
2     for each vertex v in Graph:
3         dist[v] := infinity ;
4         previous[v] := undefined ;
5     end for
6     dist[source] := 0 ;
7     Q := the set of all nodes in Graph ;
8     while Q is not empty:
9         u := vertex in Q with smallest distance in dist[] ;
10        remove u from Q ;
11        if dist[u] = infinity:
12            break ;
13        end if
14        for each neighbor v of u:
15            alt := dist[u] + dist_between(u, v) ;
16            if alt < dist[v]:
17                dist[v] := alt ;
18                previous[v] := u ;
19                decrease-key v in Q ;
20            end if
21        end for
22     end while
23     return dist;
24 endfunction
```

If we are only interested in a shortest path between vertices $source$ and $target$, we can terminate the search at line 13 if $u = target$. Now we can read the shortest path from $source$ to $target$ by reverse iteration:

```plaintext
1 S := empty sequence
2 u := target
3 while previous[u] is defined:
4     S := previous[u] in S
5     u := previous[u]
6 end while
7 return S;
```
Now sequence $S$ is the list of vertices constituting one of the shortest paths from source to target, or the empty sequence if no path exists.

A more general problem would be to find all the shortest paths between source and target (there might be several different ones of the same length). Then instead of storing only a single node in each entry of previous[] we would store all nodes satisfying the relaxation condition. For example, if both $r$ and source connect to target and both of them lie on different shortest paths through target (because the edge cost is the same in both cases), then we would add both $r$ and source to previous[target]. When the algorithm completes, previous[] data structure will actually describe a graph that is a subset of the original graph with some edges removed. Its key property will be that if the algorithm was run with some starting node, then every path from that node to any other node in the new graph will be the shortest path between those nodes in the original graph, and all paths of that length from the original graph will be present in the new graph. Then to actually find all these shortest paths between two given nodes we would use a path finding algorithm on the new graph, such as depth-first search.

**Running time**

An upper bound of the running time of Dijkstra's algorithm on a graph with edges $E$ and vertices $V$ can be expressed as a function of $|E|$ and $|V|$ using big-O notation.

For any implementation of vertex set $Q$ the running time is in $O(|E| \cdot dk_Q + |V| \cdot em_Q)$, where $dk_Q$ and $em_Q$ are times needed to perform decrease key and extract minimum operations in set $Q$, respectively.

The simplest implementation of the Dijkstra's algorithm stores vertices of set $Q$ in an ordinary linked list or array, and extract minimum from $Q$ is simply a linear search through all vertices in $Q$. In this case, the running time is $O(|E| + |V|^2) = O(|V|^2)$.

For sparse graphs, that is, graphs with far fewer than $O(|V|^2)$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. With a self-balancing binary search tree or binary heap, the algorithm requires $\Theta((|E| + |V|) \log |V|)$ time (which is dominated by $\Theta(|E| \log |V|)$ assuming the graph is connected). To avoid $O(|V|)$ look-up in decrease-key step on a vanilla binary heap, it is necessary to maintain a supplementary index mapping each vertex to the heap's index (and keep it up to date as priority queue $Q$ changes), making it take only $O(\log |V|)$ time instead. The Fibonacci heap improves this to $O(|E| + |V| \log |V|)$.

Note that for directed acyclic graphs, it is possible to find shortest paths from a given starting vertex in linear time, by processing the vertices in a topological order, and calculating the path length for each vertex to be the minimum length obtained via any of its incoming edges.\[3\]

**Related problems and algorithms**

The functionality of Dijkstra's original algorithm can be extended with a variety of modifications. For example, sometimes it is desirable to present solutions which are less than mathematically optimal. To obtain a ranked list of less-than-optimal solutions, the optimal solution is first calculated. A single edge appearing in the optimal solution is removed from the graph, and the optimum solution to this new graph is
calculated. Each edge of the original solution is suppressed in turn and a new shortest-path calculated. The secondary solutions are then ranked and presented after the first optimal solution.

Dijkstra's algorithm is usually the working principle behind link-state routing protocols, OSPF and IS-IS being the most common ones.

Unlike Dijkstra's algorithm, the Bellman–Ford algorithm can be used on graphs with negative edge weights, as long as the graph contains no negative cycle reachable from the source vertex \( s \). The presence of such cycles means there is no shortest path, since the total weight becomes lower each time the cycle is traversed.

The \( A^* \) algorithm is a generalization of Dijkstra's algorithm that cuts down on the size of the subgraph that must be explored, if additional information is available that provides a lower bound on the "distance" to the target. This approach can be viewed from the perspective of linear programming: there is a natural linear program for computing shortest paths, and solutions to its dual linear program are feasible if and only if they form a consistent heuristic (speaking roughly, since the sign conventions differ from place to place in the literature). This feasible dual / consistent heuristic defines a non-negative reduced cost and \( A^* \) is essentially running Dijkstra's algorithm with these reduced costs. If the dual satisfies the weaker condition of admissibility, then \( A^* \) is instead more akin to the Bellman–Ford algorithm.

The process that underlies Dijkstra's algorithm is similar to the greedy process used in Prim's algorithm. Prim's purpose is to find a minimum spanning tree that connects all nodes in the graph; Dijkstra is concerned with only two nodes. Prim's does not evaluate the total weight of the path from the starting node, only the individual path.

Breadth-first search can be viewed as a special-case of Dijkstra's algorithm on unweighted graphs, where the priority queue degenerates into a FIFO queue.

**Dynamic programming perspective**

From a dynamic programming point of view, Dijkstra's algorithm is a successive approximation scheme that solves the dynamic programming functional equation for the shortest path problem by the Reaching method.\(^4\)\(^5\)\(^6\)

In fact, Dijkstra's explanation of the logic behind the algorithm,\(^7\) namely

**Problem 2.** Find the path of minimum total length between two given nodes \( P \) and \( Q \).

We use the fact that, if \( R \) is a node on the minimal path from \( P \) to \( Q \), knowledge of the latter implies the knowledge of the minimal path from \( P \) to \( R \).

is a paraphrasing of Bellman's famous Principle of Optimality in the context of the shortest path problem.

**See also**

- \( A^* \) search algorithm
- Euclidean shortest path
- Flood fill
- Longest path problem

**Notes**

1. ^ Dijkstra, Edsger; Thomas J. Misa, Editor (2010-08). "An Interview with Edsger W. Dijkstra". Communications
"What is the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path which I designed in about 20 minutes. One morning I was shopping with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path."

2. ^ Dijkstra 1959
7. ^ Dijkstra 1959, p. 270

References


External links

- Dijkstra's Algorithm in C++ (https://github.com/xtaci/algorithm/blob/master/include/dijkstra.h)
- Dijkstra's Algorithm with 'c' implementation (http://scanftree.com/Data_Structure/dijkstra's-algorithm)
Dijkstra's Algorithm in C# (http://www.codeproject.com/KB/recipes/ShortestPathCalculation.aspx)
Fast Priority Queue Implementation of Dijkstra's Algorithm in C# (http://www.codeproject.com /KB/recipes/FastHeapDijkstra.aspx)
Applet by Carla Laffra of Pace University (http://www.dgp.toronto.edu/people/JamesStewart /270/9798s/Laffra/DijkstraApplet.html)
Animation of Dijkstra's algorithm (http://www.cs.sunysb.edu/~skiena/combinatorica/animations /dijkstra.html)
Visualization of Dijkstra's Algorithm (http://students.ceid.upatras.gr/~papagel/english/java_docs /minDijk.htm)
Shortest Path Problem: Dijkstra's Algorithm (http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra /Dijkstra.shtml)
Dijkstra's Algorithm Applet (http://www.unf.edu/~wkloster/foundations/DijkstraApplet /DijkstraApplet.htm)
Open Source Java Graph package with implementation of Dijkstra's Algorithm
(http://code.google.com/p/annas/)
Haskell implementation of Dijkstra's Algorithm (http://bonsaicode.wordpress.com/2011/01 /04/programming-praxis-dijkstra's-algorithm/) on Bonsai code
QuickGraph, Graph Data Structures and Algorithms for .NET (http://quickgraph.codeplex.com/)
Implementation in Boost C++ library (http://www.boost.org/doc/libs/1_43_0/libs/graph /doc/dijkstra_shortest_paths.html)
Implementation in T-SQL (http://hansolav.net/sql/graphs.html)
A Java library for path finding with Dijkstra's Algorithm and example applet
(http://www.stackframe.com/software/PathFinder)
Dijkstra's Algorithm in C Programming Language (http://www.rawbytes.com/dijkstras-algorithm-in-c/)
Step through Dijkstra's Algorithm in an online JavaScript Debugger (http://www.turb0js.com /a/Dijkstra%27s_Algorithm)

Categories: 1959 in computer science | Graph algorithms | Search algorithms | Routing algorithms | Combinatorial optimization | Dutch inventions

This page was last modified on 26 August 2013 at 20:36.
Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.