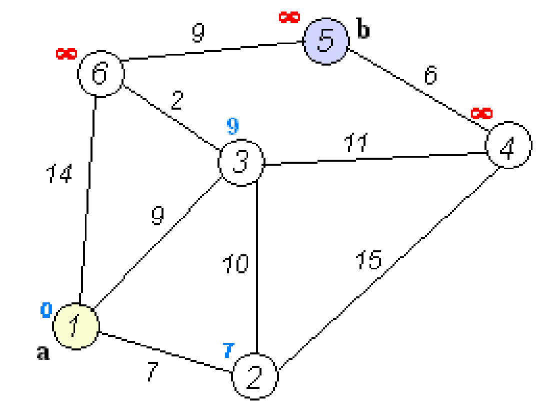
Dijkstra's algorithm

From Wikipedia, the free encyclopedia

Dijkstra's algorithm, conceived by computer scientist Edsger Dijkstra in 1956 and published in 1959,[1][2] is an algorithm for finding the shortest paths between nodes in graph (which may represent, for example, road networks). The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes,[2] but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest path tree.

Dijkstra's algorithm



Dijkstra's algorithm. It picks the unvisited vertex with the lowest-distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\|E\| + \|V\| \log \|V\|)$

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[3]:196–206 (although Dijkstra originally only considered the shortest path between a given pair of nodes[4]). 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 algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First). It is also employed as a subroutine in other algorithms such as Johnson's.

Dijkstra's original algorithm does not use a min-priority queue and runs in time $O(|V|^2)$ (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.

In some fields, in particular artificial intelligence, Dijkstra's algorithm or a variant of it is known as **uniform-cost search** and formulated as an instance of the more general idea of best-first search.[5]

Contents

History

Dijkstra thought about shortest path problem when working at Mathematical Center in Amsterdam in 1956 as a program to demonstrate capabilities of the new computer called ARMAC. His objective was to choose both a problem as well as answer (that will be produced by computer) that non-computing people can understand. He designed the shortest path algorithm in about 20 minutes without aid of paper and pen and later implemented it for ARMAC for slightly simplified transportation map of 64 cities in Netherland (ARMAC was 6-bit computer and hence could hold 64 cities comfortably).[1] A year later, he came across another problem from hardware engineers working on the institute's next computer: minimize the amount of wire needed to connect the pins on the back panel of the machine. As a solution, he re-discovered the algorithm known as Prim's minimal spanning tree algorithm in the same year that Prim re-discovered it. Dijkstra delayed publication of both algorithms until 1959.[6]

Algorithm

Let the node at which we are starting be called the **initial node**. Let the **distance of node *Y*** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

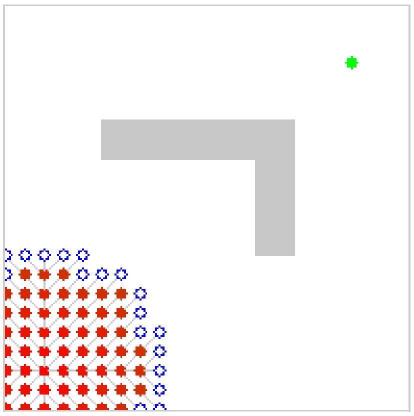


Illustration of Dijkstra's algorithm search for finding path from a start node (lower left, red) to a goal node (upper right, green) in a robot motion planning problem. Open nodes represent the "tentative" set. Filled nodes are visited ones, with color representing the distance: the greener, the farther. Nodes in all the different directions are explored uniformly, appearing as a more-or-less circular wavefront as Dijkstra's algorithm uses a heuristic identically equal to 0.

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. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. 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 B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
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; occurs when there is no connection between the initial node and remaining unvisited nodes), 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, in formal notation 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 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 the 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. This algorithm, therefore "expands outward" from the starting point, interactively 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 ← vertex in Q with min dist[u]`, searches for the vertex *u* in the vertex set *Q* that has the least `dist[u]` value. `length(u, v)` returns the length of the edge joining (i.e. the distance between) the two neighbor-nodes *u* and *v*. The variable *alt* on line 19 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 *alt* path. The `previous` array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

```
1  function Dijkstra(Graph, source):
2
3      dist[source] ← 0                      // Distance from source to source
4      prev[source] ← undefined              // Previous node in optimal path initialization
5
6      for each vertex v in Graph:  // Initialization
7          if v ≠ source            // Where v has not yet been removed from Q (unvisited nodes)
8              dist[v] ← infinity            // Unknown distance function from source to v
```

```
 9              prev[v] ← undefined             // Previous node in optimal path from source
10          end if
11          add v to Q                      // All nodes initially in Q (unvisited nodes)
12      end for
13
14      while Q is not empty:
15          u ← vertex in Q with min dist[u]  // Source node in first case
16          remove u from Q
17
18          for each neighbor v of u:           // where v is still in Q.
19              alt ← dist[u] + length(u, v)
20              if alt < dist[v]:               // A shorter path to v has been found
21                  dist[v] ← alt
22                  prev[v] ← u
23              end if
24          end for
25      end while
26
27      return dist[], prev[]
28
29  end function
```

If we are only interested in a shortest path between vertices *source* and *target*, we can terminate the search after line 15 if `u = target`. Now we can read the shortest path from *source* to *target* by reverse iteration:

```
S ← empty sequence
u ← target
while prev[u] is defined:                  // Construct the shortest path with a stack S
    insert u at the beginning of S          // Push the vertex onto the stack
    u ← prev[u]                             // Traverse from target to source
end while
```

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.

Using a priority queue

A min-priority queue is an abstract data structure that provides 3 basic operations : `add_with_priority()`, `decrease_priority()` and `extract_min()`. As mentioned earlier, using such a data structure can lead to faster computing times than using a basic queue. Notably, Fibonacci heap (Fredman & Tarjan 1984) or Brodal queue offer optimal implementations for those 3 operations. As the algorithm is slightly different, we mention it here, in pseudo-code as well :

```
function Dijkstra(Graph, source):
    dist[source] ← 0                       // Initialization
    for each vertex v in Graph:
        if v ≠ source
            dist[v] ← infinity             // Unknown distance from source to v
            prev[v] ← undefined            // Predecessor of v
        end if
        Q.add_with_priority(v, dist[v])
    end for

    while Q is not empty:                  // The main loop
        u ← Q.extract_min()               // Remove and return best vertex
        for each neighbor v of u:
            alt = dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)
            end if
        end for
    end while
    return prev[]
```

Instead of filling the priority queue with all nodes in the initialization phase, it is also possible to initialize it to contain only *source*; then, inside the `if alt < dist[v]` block, the node must be inserted if not already in the queue (instead of performing a `decrease_priority` operation).[3]:198

It should be noted that other data structures can be used to achieve even faster computing times in practice.[7]

Running time

Bounds of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of the number of edges, denoted $|E|$, and the number of vertices, denoted $|V|$, using big-O notation. How tight a bound is possible depends on the way the vertex set Q is implemented. In the following, note that upper bounds can be simplified because $|E| = O(|V|^2)$ for any graph, but that simplification disregards the fact that in some problems, other upper bounds on $|E|$ may hold.

For any implementation of the vertex set Q, the running time is in

$$O(|E| \cdot T_{\mathrm{dk}} + |V| \cdot T_{\mathrm{em}}),$$

where T_{dk} and T_{em} are the complexities of the *decrease-key* and *extract-minimum* operations in Q, respectively. The simplest implementation of the Dijkstra's algorithm stores the vertex set Q as an ordinary linked list or array, and extract-minimum 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 $|V|^2$ 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. To perform decrease-key steps in a binary heap efficiently, it is necessary to use an auxiliary data structure that maps each vertex to its position in the heap, and to keep this structure up to date as the priority queue Q changes. With a self-balancing binary search tree or binary heap, the algorithm requires

$$\Theta((|E| + |V|) \log |V|)$$

time in the worst case; for connected graphs this time bound can be simplified to $\Theta(|E| \log |V|)$. The Fibonacci heap improves this to

$$O(|E| + |V| \log |V|).$$

When using binary heaps, the average case time complexity is lower than the worst-case: assuming edge costs are drawn independently from a common probability distribution, the expected number of *decrease-key* operations is bounded by $O(|V| \log(|E|/|V|))$, giving a total running time of[3]:199–200

$$O\left(|E| + |V| \log \frac{|E|}{|V|} \log |V|\right).$$

Practical optimizations and infinite graphs

In common presentations of Dijkstra's algorithm, initially all nodes are entered into the priority queue. This is, however, not necessary: the algorithm can start with a priority queue that contains only one item, and insert new items as they are discovered (instead of doing a decrease-key, check whether the key is in the queue; if it is, decrease its key, otherwise insert it).[3]:198 This variant has the same worst-case bounds as the common variant, but maintains a smaller priority queue in practice, speeding up the queue operations.[5]

Moreover, not inserting all nodes in a graph makes it possible to extend the algorithm to find the shortest path from a single source to the closest of a set of target nodes on infinite graphs or those too large to represent in memory. The resulting algorithm is called *uniform-cost search* (UCS) in the artificial intelligence literature[5][8] and can be expressed in pseudocode as

```
procedure UniformCostSearch(Graph, start, goal)
  node ← start
  cost ← 0
  frontier ← priority queue containing node only
  explored ← empty set
  do
    if frontier is empty
      return failure
    node := frontier.pop()
    if node is goal
      return solution
    explored.add(node)
    for each of node's neighbors n
      if n is not in explored
        if n is not in frontier
          frontier.add(n)
        else if n is in frontier with higher cost
          replace existing node with n
```

The complexity of this algorithm can be expressed in an alternative way for very large graphs: when C^* is the length of the shortest path from the start node to any node satisfying the "goal" predicate, and each edge has cost at least ε, then the algorithm's worst-case time and space complexity are both in $O(b^{\lceil C^*/\varepsilon \rceil})$.[8]

Specialized variants

When arc weights are integers and bounded by a constant C, the usage of a special priority queue structure by Van Emde Boas etal.(1977) (Ahuja et al. 1990) brings the complexity to $O(|E| \log\log|C|)$. Another interesting implementation based on a combination of a new radix heap and the well-known Fibonacci heap runs in time $O(|E| + |V|\sqrt{\log|C|})$ (Ahuja et al. 1990). Finally, the best algorithms in this special case are as follows. The algorithm given by (Thorup 2000) runs in $O(|E| \log\log|V|)$ time and the algorithm given by (Raman 1997) runs in $O(|E| + |V| \min\{(\log|V|)^{1/3+\epsilon}, (\log|C|)^{1/4+\epsilon}\})$ time.

Also, for directed acyclic graphs, it is possible to find shortest paths from a given starting vertex in linear $O(|E| + |V|)$ 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.[9][10]

In the special case of integer weights and undirected graphs, the Dijkstra's algorithm can be completely countered with a linear $O(|V| + |E|)$ complexity algorithm, given by (Thorup 1999).

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. It is possible to adapt Dijkstra's algorithm to handle negative weight edges by combining it with the Bellman-Ford algorithm (to remove negative edges and detect negative cycles), such an algorithm is called Johnson's algorithm.

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.

Fast marching method can be viewed as a continuous version of Dijkstra's algorithm which computes the geodesic distance on a triangle mesh.

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.[11][12][13]

In fact, Dijkstra's explanation of the logic behind the algorithm,[14] 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
- Bellman–Ford algorithm
- Euclidean shortest path
- Flood fill

- Floyd–Warshall algorithm
- Johnson's algorithm
- Longest path problem

Notes

1. ^ ***a b*** Dijkstra, Edsger; Thomas J. Misa, Editor (August 2010). "An Interview with Edsger W. Dijkstra". *Communications of the ACM* **53** (8): 41–47. doi:10.1145/1787234.1787249 (https://dx.doi.org/10.1145%2F1787234.1787249). "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. ^ ***a b*** Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs" (http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf). *Numerische Mathematik* **1**: 269–271. doi:10.1007/BF01386390 (https://dx.doi.org/10.1007%2FBF01386390).
3. ^ ***a b c d*** Mehlhorn, Kurt; Sanders, Peter (2008). *Algorithms and Data Structures: The Basic Toolbox*. Springer.
4. **^** Dijkstra 1959
5. ^ ***a b c*** Felner, Ariel (2011). *Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm* (http://www.aaai.org/ocs/index.php/SOCS/SOCS11/paper/view/4017/4357). Proc. 4th Int'l Symp. on Combinatorial Search. In a route-finding problem, Felner finds that the queue can be a factor 500–600 smaller, taking some 40% of the running time.
6. **^** Dijkstra, Edward W., *Reflections on "A note on two problems in connexion with graphs* (https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD841a.PDF)
7. **^** Chen, M.; Chowdhury, R. A.; Ramachandran, V.; Roche, D. L.; Tong, L. (2007). *Priority Queues and Dijkstra's Algorithm — UTCS Technical Report TR-07-54 — 12 October 2007* (http://www.cs.sunysb.edu/~rezaul/papers/TR-07-54.pdf). Austin, Texas: The University of Texas at Austin, Department of Computer Sciences.
8. ^ ***a b*** Russell, Stuart; Norvig, Peter (2009) [1995]. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall. pp. 75, 81. ISBN 978-0-13-604259-4.
9. **^** http://www.boost.org/doc/libs/1_44_0/libs/graph/doc/dag_shortest_paths.html
10. **^** Cormen etal, Introduction to Algorithms & 3ed,chapter-24 2009
11. **^** Sniedovich, M. (2006). "Dijkstra's algorithm revisited: the dynamic programming connexion" (http://matwbn.icm.edu.pl/ksiazki/cc/cc35/cc3536.pdf) (PDF). *Journal of Control and Cybernetics* **35** (3): 599–620. Online version of the paper with interactive computational modules. (http://www.ifors.ms.unimelb.edu.au/tutorial/dijkstra_new/index.html)
12. **^** Denardo, E.V. (2003). *Dynamic Programming: Models and Applications*. Mineola, NY: Dover Publications. ISBN 978-0-486-42810-9.
13. **^** Sniedovich, M. (2010). *Dynamic Programming: Foundations and Principles*. Francis & Taylor. ISBN 978-0-8247-4099-3.
14. **^** Dijkstra 1959, p. 270

References

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". *Introduction to Algorithms* (Second ed.). MIT Press and McGraw–Hill. pp. 595–601. ISBN 0-262-03293-7.
- Fredman, Michael Lawrence; Tarjan, Robert E. (1984). *Fibonacci heaps and their uses in improved network optimization algorithms*. 25th Annual Symposium on Foundations of Computer Science. IEEE. pp. 338–346.

doi:10.1109/SFCS.1984.715934 (https://dx.doi.org/10.1109%2FSFCS.1984.715934).

- Fredman, Michael Lawrence; Tarjan, Robert E. (1987). "Fibonacci heaps and their uses in improved network optimization algorithms" (http://portal.acm.org/citation.cfm?id=28874). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi:10.1145/28869.28874 (https://dx.doi.org/10.1145%2F28869.28874).
- Zhan, F. Benjamin; Noon, Charles E. (February 1998). "Shortest Path Algorithms: An Evaluation Using Real Road Networks". *Transportation Science* **32** (1): 65–73. doi:10.1287/trsc.32.1.65 (https://dx.doi.org/10.1287%2Ftrsc.32.1.65).
- Leyzorek, M.; Gray, R. S.; Johnson, A. A.; Ladew, W. C.; Meaker, Jr., S. R.; Petry, R. M.; Seitz, R. N. (1957). *Investigation of Model Techniques — First Annual Report — 6 June 1956 — 1 July 1957 — A Study of Model Techniques for Communication Systems*. Cleveland, Ohio: Case Institute of Technology.
- Knuth, D.E. (1977). "A Generalization of Dijkstra's Algorithm". *Information Processing Letters* **6** (1): 1–5. doi:10.1016/0020-0190(77)90002-3 (https://dx.doi.org/10.1016%2F0020-0190%2877%2990002-3).
- Ahuja, Ravindra K.; Mehlhorn, Kurt; Orlin, James B.; Tarjan, Robert E. (April 1990). "Faster Algorithms for the Shortest Path Problem". *Journal of Association for Computing Machinery (ACM)* **37** (2): 213–223. doi:10.1145/77600.77615 (https://dx.doi.org/10.1145%2F77600.77615).
- Raman, Rajeev (1997). "Recent results on the single-source shortest paths problem". *SIGACT News* **28** (2): 81–87. doi:10.1145/261342.261352 (https://dx.doi.org/10.1145%2F261342.261352).
- Thorup, Mikkel (2000). "On RAM priority Queues". *SIAM Journal on Computing* **30** (1): 86–109. doi:10.1137/S0097539795288246 (https://dx.doi.org/10.1137%2FS0097539795288246).
- Thorup, Mikkel (1999). "Undirected single-source shortest paths with positive integer weights in linear time". *journal of the ACM* **46** (3): 362–394. doi:10.1145/316542.316548 (https://dx.doi.org/10.1145%2F316542.316548).

External links

Wikimedia Commons has media related to ***Dijkstra's algorithm***.

- Applet by Carla Laffra of Pace University (http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html)
- Dijkstra's Algorithm Applet (http://www.unf.edu/~wkloster/foundations/DijkstraApplet/DijkstraApplet.htm)
- Shortest Path Problem: Dijkstra's Algorithm (http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml)
- Dijkstra's Algorithm Simulation (http://optlab-server.sce.carleton.ca/POAnimations2007/DijkstrasAlgo.html)
- Oral history interview with Edsger W. Dijkstra (http://purl.umn.edu/107247), Charles Babbage Institute University of Minnesota, Minneapolis.
- Animation of Dijkstra's algorithm (http://www.cs.sunysb.edu/~skiena/combinatorica/animations/dijkstra.html)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=648542472"

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

- This page was last modified on 23 February 2015, at 22:20.