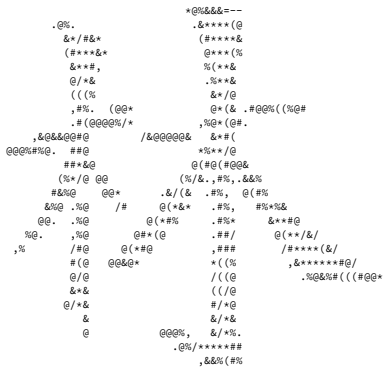
Tim Lum (twhlum@gmail.com)
2019.01.31
CSS 502 - Data Structures and Algorithms II
Djikstra's Algorithm



DJIKSTRA'S ALGORITHM

CSS 502 - Data Structures and Algorithms

2019.01.31 (Winter, Q2)

1 - Overview:

Given a graph (a set of connected nodes) with a start node and edge costs between nodes, Djikstra's algorithm guarantees a method to determine the lowest travel cost needed to reach all other nodes.

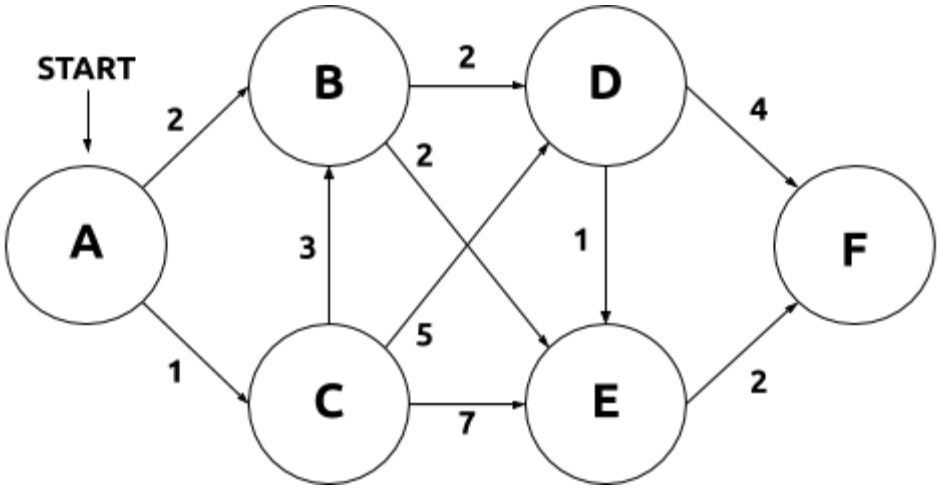


Fig. 1.0 - A directional graph of six nodes with associated edge distances

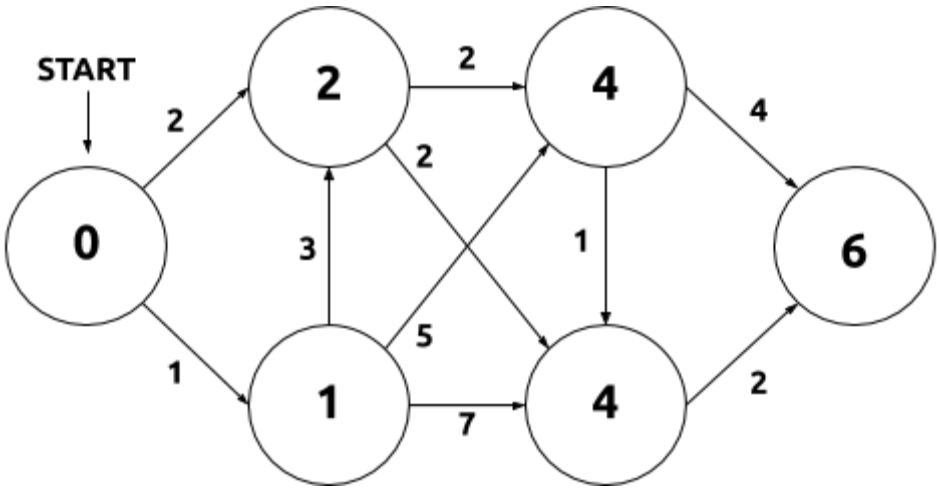


Fig. 1.1 - A directional graph of six nodes with shortest distances solved by Djikstra's algorithm

If a destination node is included, the algorithm will halt when it solves the shortest possible travel distance to the destination node.

Analysis on the graph state after the algorithm halts is often performed to determine the shortest available path in addition to the total travel cost derived previously.

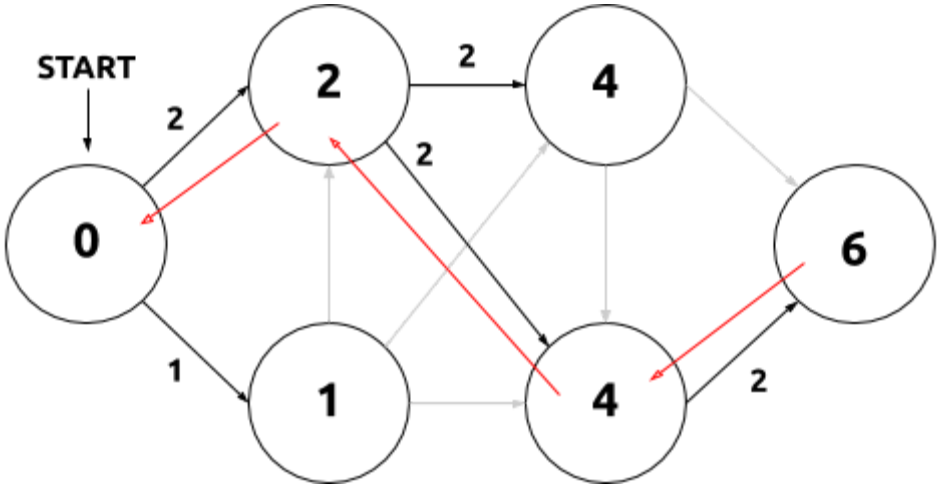


Fig. 1.2 - Back-analysis of a solved graph yielding the shortest path

In summary, Djikstra's algorithm is an effective method for determining the shortest path between nodes.

2 - Components:

These represent the minimum required set of information needed to execute Djikstra's algorithm:
0) Start node
1) Adjacencies and distances
2) Visited status - That is, have we confirmed that we found the shortest distance to this node?
3) Reachable - A **set** of nodes which Djikstra's can generate a path to

3 - Description:

From the start node (distance 0), gauge the total distance needed to reach all adjacent nodes.
Select the lowest total-distance-to-reach node ('NextNode').
Mark NextNode as visited.
Update the total distance to reach all of its adjacent, unvisited nodes.
Repeat this process of selection, mark-as-visited, update-total-distances until the destination is reached.

4 - Adjacency Representation:

While a drawing of the graph may represent adjacency distances easily, this information must be contained in a machine-readable format. Three common approaches are detailed as follows:

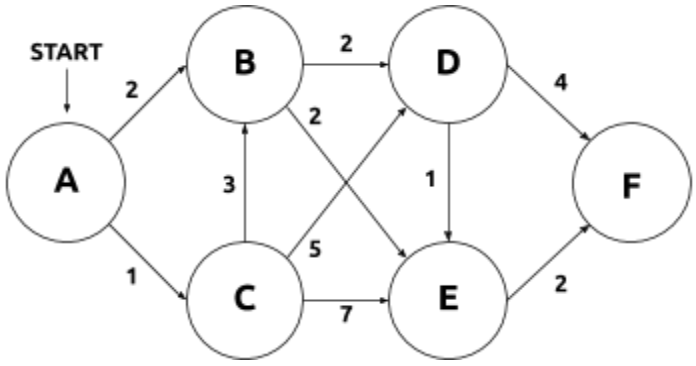


Fig. 4.1 - The above graph shall be utilized in the following examples

1) Adjacencies as a vector/queue/stack/array - If nodes are represented as class objects, the simplest way to store their adjacencies is as an array, internal to each node object:

```
Node 'A's adjacencies only

   A     B     C     D     E     F
-------------------------------------
|  0  |  2  |  1  |  -  |  -  |  -  |
-------------------------------------
```

Fig. 4.2 - Adjacencies stored as an array: 'This' node (A) is adjacent to nodes B (distance 1) and C (distance 2).
It is also technically adjacent to itself (distance 0), but the representation of this is left as an implementation detail.

While this works for small graphs, large graphs where nodes are loosely connected will result in significant amounts of unused memory space being allocated to documenting non-adjacencies. Optimization strategies are myriad, selection and design of which is left as an exercise to the reader and/or standard library writer.

2) Adjacencies as an array of lists - Adjacencies may also be stored as an array of lists:

```
[ ]
 A -> B:2 -> C:1
 B -> D:2 -> E:2
 C -> B:3 -> D:5 -> E:7
 D -> E:1 -> F:4
 E -> F:2
 F ->
```

Fig. 4.3 - Adjacencies stored as a linked list of tuples

This strategy is space-efficient, though somewhat more complicated requiring as it does a Node:Distance tuple and associated logic for linked list traversal.

3) Adjacencies as matrix - Lastly the adjacencies may be stored as a matrix. In terms of memory space, this is identical to storing a nodes' own adjacencies as an array, but moves the storage from individual nodes to a graph-wide view of the adjacency situation.

FROM	TO A	TO B	TO C	TO D	TO E	TO F
A	0	2	1	-	-	-
B	-	0	-	2	2	-
C	-	3	0	5	7	-
D	-	-	-	0	1	4
E	-	-	-	-	0	2
F	-	-	-	-	-	0

Fig. 4.4 - Adjacencies stored as an adjacency matrix. Note that in uni-directional / one-way graphs, roughly half the allocated memory space is unused. Also note the prominent diagonal feature of 0's denoting nodes adjacent to themselves.

5 - Djikstra's Execution and Shortest Path Representation:

In a similar fashion to the adjacency matrix, the current state of Djikstra's algorithm may also be represented as a matrix. Note that these two bodies of data are separate, and considerable complexity is introduced if you attempt to solve the path in the same memory space as the adjacency matrix. Using the adjacency matrix above, we may derive that the total cost to reach 'A' is zero.

FROM \ TO	A	B	C	D	E	F
A	0	2	1	-	-	-
B	-	-	-	-	-	-
C	-	-	-	-	-	-
D	-	-	-	-	-	-
E	-	-	-	-	-	-
F	-	-	-	-	-	-

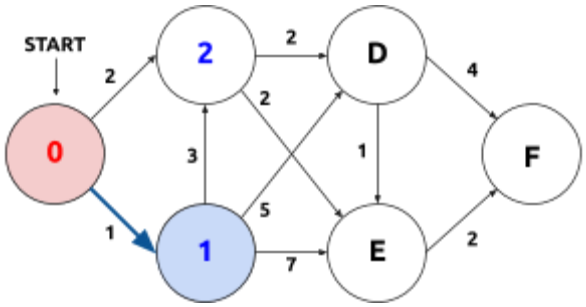


Fig. 5.0 - Start state of Djikstra's algorithm.

Since we are sure no shorter path to 'A' exists, node 'A' is flagged as 'visited' and excluded from future distance updating (after all, any changes to the distance estimate can only get worse if we've already figured out what the shortest total distance to 'A' is).

Next, the distances to all of A's adjacencies are loaded, and the next node to explore may be determined as the lowest value, unvisited node. In this case, 'C'. We then:

- Move to C
- Update our tentative distances to all nodes adjacent C, which is equal to whichever is less:
  - <the current node's total distance to reach + relevant adjacency distance>
    OR
  - <the adjacent node's pre-existing tentative distance>
- Mark C as visited
- Select the next lowest total-travel-distance node to explore
- And repeat

We should pause here a moment to consider the logical implications of this algorithm. If we had proven that the distance to the initial node was as short as possible, then we may by extension also assert that the distance to the next nearest node is also as short as possible. By successively moving to the next total-distance nearest node, we are in effect demonstrating that no shorter path could exist for any node we move to which is selected by this methodology.

		TO A	B	C	D	E	F
	A	0	2	1	-	-	-
	B	-	-	-	-	-	-
F	C	-	1+3	-	1+5	1+7	-
R	D	-	-	-	-	-	-
O	E	-	-	-	-	-	-
M	F	-	-	-	-	-	-

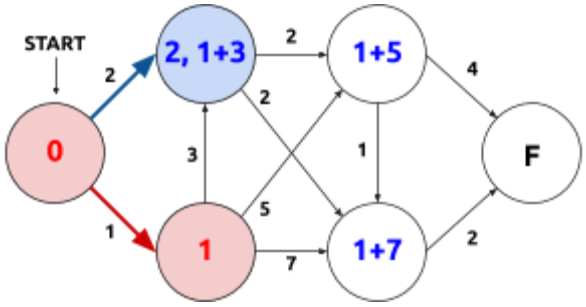


Fig. 5.1 - Second iteration of Djikstra's algorithm. 'B' is selected as the next node to explore after C, since it holds the next lowest, unvisited total travel distance (2).

We then repeat the process until the destination is visited.

		TO A	B	C	D	E	F
	A	0	2	1	-	-	-
	B	-	-	-	2+2	2+2	-
F	C	-	4	-	6	8	-
R	D	-	-	-	-	-	-
O	E	-	-	-	-	-	-
M	F	-	-	-	-	-	-

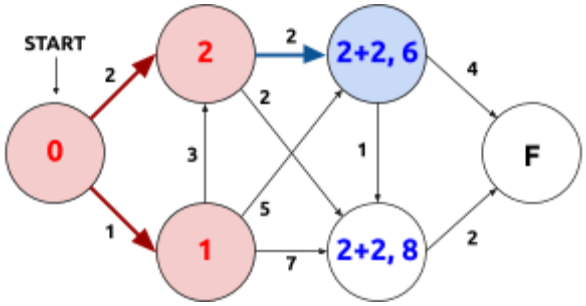


Fig. 5.2 - Third iteration of Djikstra's algorithm. Either D or E may be selected for equivalent results

FROM \ TO	A	B	C	D	E	F
A	0	2	1	–	–	–
B	–	–	–	4	4	–
C	–	4	–	6	8	–
D	–	–	–	–	4+1	4+4
E	–	–	–	–	–	–
F	–	–	–	–	–	–

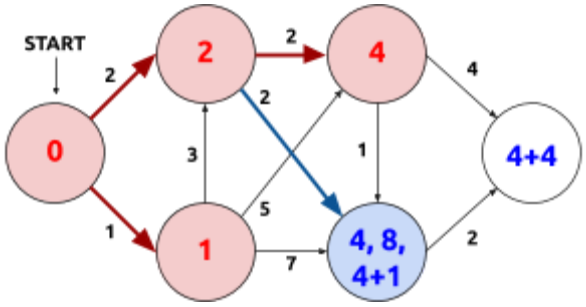


Fig. 5.3 - Fourth iteration of Djikstra's algorithm. 'E' must be explored before 'F'

FROM \ TO	A	B	C	D	E	F
A	0	2	1	–	–	–
B	–	–	–	4	4	–
C	–	4	–	6	8	–
D	–	–	–	–	5	8
E	–	–	–	–	–	4+2
F	–	–	–	–	–	–

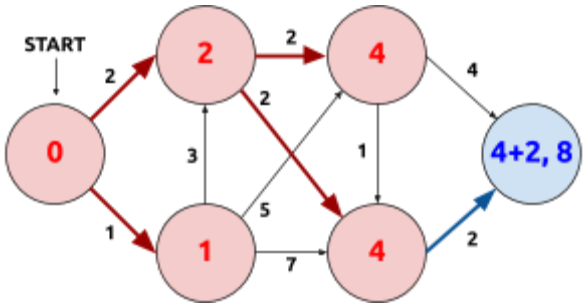


Fig. 5.4 - Fifth iteration of Djikstra's algorithm.

FROM \ TO	A	B	C	D	E	F
A	0	2	1	–	–	–
B	–	–	–	4	4	–
C	–	4	–	6	8	–
D	–	–	–	–	5	8
E	–	–	–	–	–	6
F	–	–	–	–	–	–

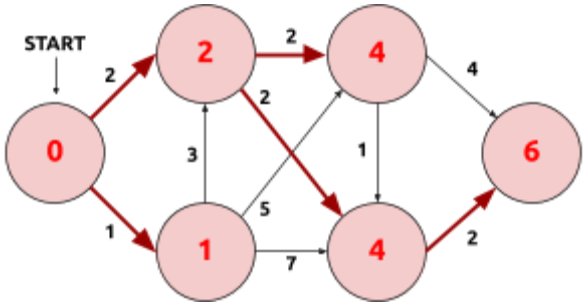


Fig. 5.5 - Halting of Djikstra's algorithm - destination reached.

6 - Path backtracking (breadcrumbing):

The resulting table may now be analyzed to determine the route. We begin at the destination (node 'F') and ask the question, "What's the best way to arrive **here**?" (Note: When you ask, "Here?", go ahead and queue whatever 'Here' is to a stream, stack, array, or other storage method - you'll want that later).

To answer that, find the lowest value in the "To F" column, as this represents the cheapest way to reach node F. We can now read across to find that to reach node 'F' for the lowest cost, we must approach from node 'E'.

```
                    TO
       A     B     C     D     E     F
    -------------------------------------
  A | 0 | 2 | 1 | - | - |  -  |
    -------------------------------------
  B | - | - | - | 4 | 4 |  -  |
    -------------------------------------
F C | - | 4 | - | 6 | 8 |  -  |
R   -------------------------------------
O D | - | - | - | - | 5 |  8  |
M   -------------------------------------
  E | - | - | - | - | - |  6  | <- Lowest value in this column. From 'E' (enque)
    -------------------------------------    But how should we get to 'E'?
  F | - | - | - | - | - |  -  |
    -------------------------------------
```

Fig. 6.0 - First backtrack: How did we arrive at the destination node?

This of course begs the question, what's the best way to arrive at node 'E'? Again, just read the 'E' column for the lowest value (4), then read across to find the approach vector ('B').

```
                    TO
       A     B     C     D     E     F
    -------------------------------------
  A | 0 | 2 | 1 | - |  -  |  -  |
    -------------------------------------
  B | - | - | - | 4 |  4  |  -  | <- Lowest value in this column. From 'B' (enque)
    -------------------------------------    But how should we get to 'B'?
F C | - | 4 | - | 6 |  8  |  -  |
R   -------------------------------------
O D | - | - | - | - |  5  |  8  |
M   -------------------------------------
  E | - | - | - | - |  -  |  6  |
    -------------------------------------
  F | - | - | - | - |  -  |  -  |
    -------------------------------------
```

Fig. 6.1 - Second backtrack: How did we arrive at node 'E'?

And so on and so forth:

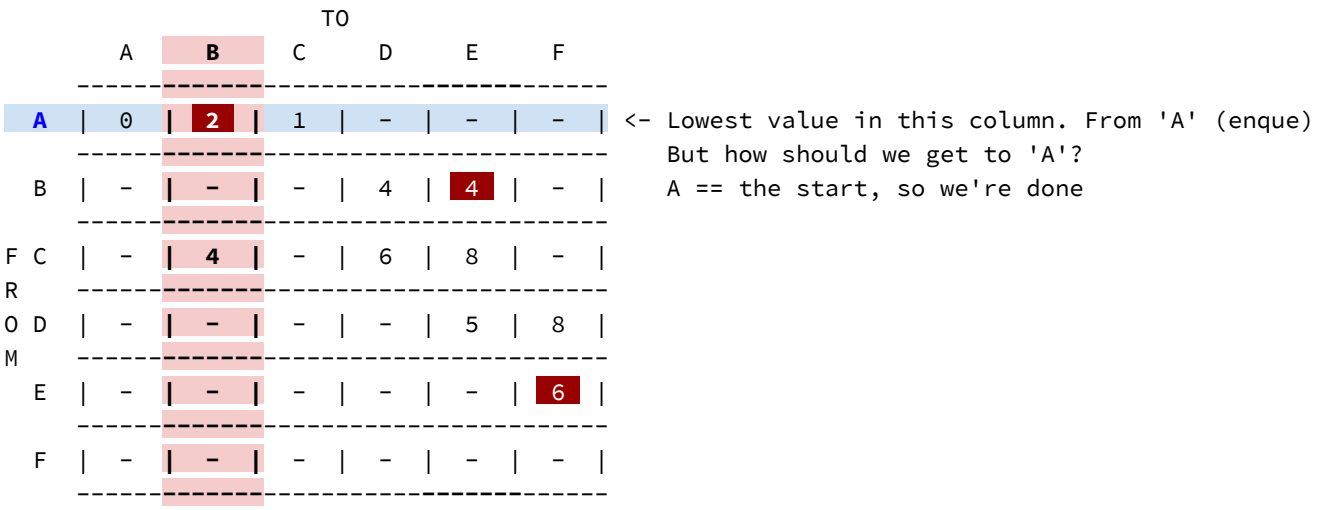


Fig. 6.2 - Third backtrack: How did we arrive at node 'B'?

Finally, read out the queue to get your path result: F <- E <- B <- A
Or stack, if you want the reverse order: A -> B -> E -> F

If handling nodes as objects, the task is somewhat simplified. Merely read off the parent pointer from every node starting from the destination, similar to a linked list.

7 - Pseudocode (nodes as objects):

```
struct tuple {
   node* adjacentNode;
   int   distance;
}

struct node {
   tuple[] adjacencies;
   bool    isVisited;
   int     costToReach; // This value is tentative until isVisited is set to 'true'
   node*   fromWhere;   //
}

djikstrasAlgorithm( node* startNodePtr, node* endNodePtr ) {
   bool  reachedDestination = false;
   node* currNodePtr        = startNodePtr;
   node* nextNodePtr        = nullptr;

   // While we have not reached the destination node...
   while ( !reachedDestination ) {

      // If this is the destination...
      if (currNodePtr == endNodePtr) {
         // Halt
         reachedDestination = true;
      }

      // Otherwise...
      else {
         // Update the lowest total distance for all valid adjacencies
         updateDistances( currNodePtr );
         // Flag this node as visited (confirmed shortest path)
         currNodePtr -> isVisited = true;
         // And find the next lowest, unvisited total distance node
         nextNodePtr = selectNextShortest( );
         // Note the 'parent' node from which we arrive
         nextNodePtr->fromWhere = currNodePtr;
         // Move to the target node
         currNodePtr = nextNodePtr;
      }

   }
   reportPath( endNodePtr );
} // Closing djikstrasAlgorithm( node*, node* )
```

`updateDistance( node* )` - Adds the internal `costToReach` and `adjacencies[]` tuples to identify and update adjacent nodes' `costToReach`. Only updates if the node is unvisited, adjacent, and the sum calculated is less than the adjacent nodes' current `costToReach`.

`selectNextShortest( )` - Examines all reachable, unvisited nodes and returns a pointer to the one with the lowest `costToReach` value.

`reportPath( node* )` - From the argument node, traces the `fromWhere` field back to the start node. May optionally also report the node->costToReach value to report both the route and distance.

8 - Pseudocode (as adjacency and distance matrices):

```
int from = <Number of Nodes>;
int to   = from;
int[from][to] adjacencyMatrix;
int[from][to] totalDistanceToReach;
bool[from]    isVisited;

// Initialize everything we need
for( each in from ) {
   for( each in to ) {
      adjacencyMatrix[from][to] = <No adjacency>;
      totalDistanceToReach      = <Unreachable>;
   }
   isVisited[from] = false;
}

djikstrasAlgorithm( int startNodeID, int endNodeID ) {
   bool reachedDestination = false;
   int  currNodeID         = startNodeID;
   int  nextNodeID         = NULL;

   // While we have not reached the destination node...
   while ( !reachedDestination ) {

      // If this is the destination...
      if (currNodeID == endNodeID) {
         // Halt
         reachedDestination = true;
      }

      // Otherwise...
      else {
         // Update the lowest total distance for all valid adjacencies
         updateDistances( currNodeID );
         // Flag this node as visited (confirmed shortest path)
         isVisited[currNodeID] = true;
         // And find the next lowest, unvisited total distance node
         nextNodeID = selectNextShortest( );
         // Move to the target node
         currNodeID = nextNodeID;
      }

   }
   reportPath( endNodeID );
} // Closing djikstrasAlgorithm( int, int )
```

`updateDistance( int )` - Uses the `totalDistanceToReach[]` and `adjacencyMatrix[]` to update adjacent nodes' entries on the `totalDistanceToReach[]` matrix. Only updates if the node is unvisited, adjacent, and the sum calculated is less than the adjacent nodes' current `costToReach`.

`selectNextShortest( )` - Examines all reachable, unvisited nodes in the `totalDistanceToReach[]` matrix and returns the int ID to the one with the lowest `costToReach` value.

`reportPath( int )` - From the argument node ID, traces the `totalDistanceToReach[]` matrix back to the start node. May optionally also report the costToReach value of the target node to report both the route and distance.

9 - FAQ:

Q: How do we represent an infinitely distant (unreachable) node? Computer number representation can't go that high.

A: You can use special values like (unsigned)INT_MAX (which is 2^32 - 1) or -1, then just have your code logic handle that special value as if it were unreachable (that is, you are not allowed to visit that node). Do -not- leave the integer value uninitialized! This is important.

Q: What if the edge to the destination has a very high cost? Does Djikstra's have to explore everything else first?

A: If the edge cost is high enough, yes, Djikstra's algorithm will explore all other options before checking the destination. The algorithm does not guarantee a speedy decision in finding the destination node, only that the path selected will be the shortest.

Q: Does Djikstra's function on bi-directional graphs? That is, if the algorithm can move both directions along an edge, does it still work?

A: Yup! In fact, it will still work even if the costs to go in the two directions differ from one another.

Q: Are there conditions in which Djikstra's will fail?

A: Classically, Djikstra's algorithm only fails in principle when edges have a negative cost to traverse (the algorithm can get stuck looping over the negative cost edges over and over again). Depending on your implementation, Djikstra's may also come to an erroneous conclusion if the destination node is unreachable or integers are used to represent the adjacent distances and overflow becomes an issue.

Practically, however, there are a lot of ways to break the implementation.

Q: Everything runs fine until I stress test with larger graphs and longer edges. My results become erroneous, but not consistently so. How come Djikstra's doesn't run the same every time?

A: If you're representing adjacency distances as a matrix of unsigned ints, the most likely cause is improper initialization of the matrix. If you only allocated memory, the number representation sitting in those 32 bits is still a valid number, it's just a random one. With most test cases where your edges are short (single- or double-digit), the odds of this random edge length being small enough to interfere with the algorithm is quite low, but the chances of interference increase as the edge sizes and total cost to reach distances add up.