Dijkstra's algorithm

isaaccomputerscience.org/concepts/dsa_search_dijkstra

There is a set of problems in computer science that require us to find the **shortest path** between a set of points. The applications are numerous and commonplace — from satellite navigation to internet packet routing; even finding the shortest length of wire needed to connect pins on a circuit board is an example of a shortest path problem.

Seeing as this type of problem arises so frequently, there are some standard algorithms that can be used to find a solution. One of these is known as **Dijkstra's algorithm**. It was designed by Dutch physicist Edsger Dijkstra in 1956, when he thought about how he might calculate the shortest route from Rotterdam to Groningen.

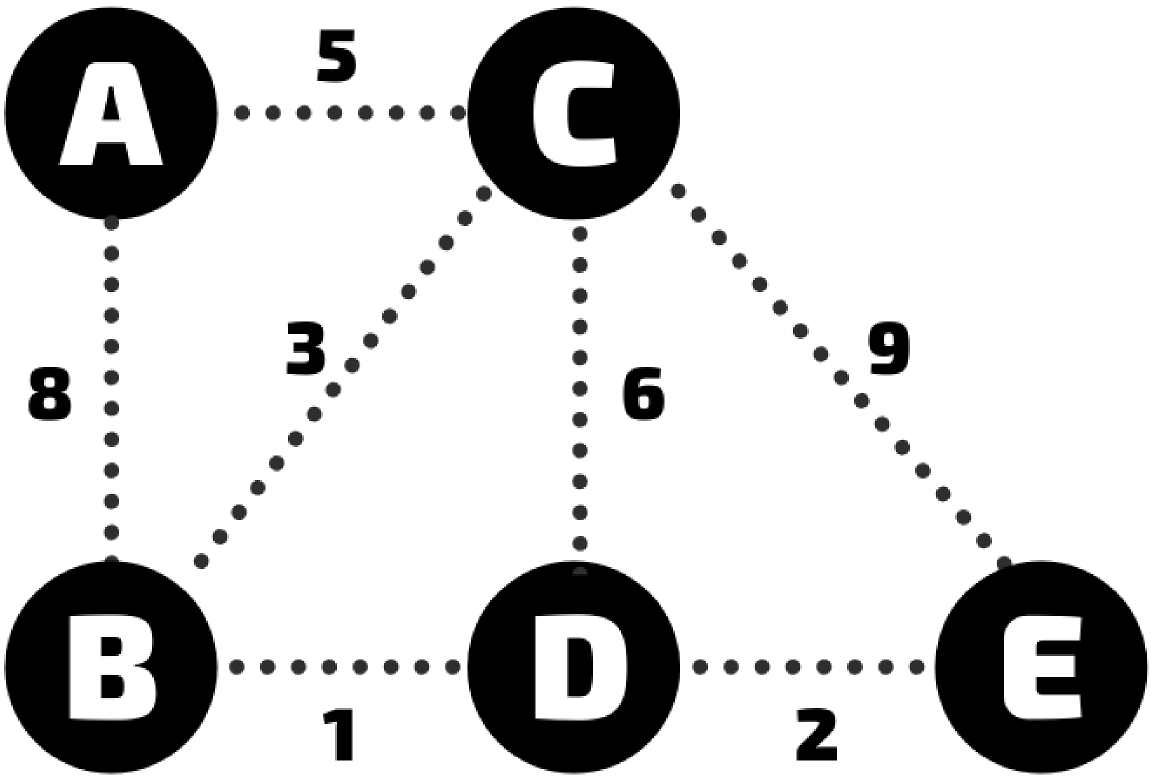


Figure 1: A weighted graph

Dijkstra’s algorithm works on a weighted graph, such as the one shown above. This graph, like all examples encountered for shortest path problems at A level, is undirected.

- **Nodes** (also called vertices) are represented in the diagram by labelled circles. Each node will store data relevant to the scenario. For example, if you use a graph to store data for a map, each node might represent a city and could store the city name. If you use a graph to store data about a local area network, each node might represent a network device, and could store the IP address and physical location of the device.

- The connections between nodes are called **edges** (sometimes called arcs). In a weighted graph these edges have a weight (or cost) associated with them. In a graph representing a map, the edges could represent a road or rail connection between cities and the weight may represent the time to travel between the cities. In a network graph the edges usually represent the connections between devices such as Ethernet or fiber optic cables. In this case the weight could represent the delay introduced by the connection.

In shortest path problems you are interested in the ‘best’ way to get from one node to another node. This is called the ‘least cost path’. The cost of a path is calculated by adding up all the individual weights for the edges that make up the path.

Dijkstra's algorithm can be used to solve a wide variety of complex problems. The algorithm is written in the context of weighted graphs, so all of the language reflects that (nodes, costs, etc). However, this solution will work on anything that can be abstracted to a series of nodes that are connected and have values attached to those connections. It can be used to manage networks, to control the movement of adversaries in video games, or to guide cars along their route.

Dijkstra’s algorithm has one motivation: to find the shortest paths from a start node to all other nodes on the graph.

- The **cost** of a path that connects two nodes is calculated by adding the weights of all the edges that belong to the path.

- The **shortest path** is the sequence of nodes, in the order they are visited, which results in the **minimum cost** to travel between the start and end node.

When the algorithm has finished running, it produces a list that holds the following information for each node:

- The **node** label

- The **cost** of the shortest path to that node (**from the start node**)

- The label of the **previous** node in the path

Using the information in this list you can backtrack through the previous nodes back to the start node. This will give you the shortest path (sequence of visited nodes) from the start node to each node and the cost of each path. This can be seen in the worked example that follows.

The algorithm step-by-step

Imagine that the nodes in the graph shown below represents 5 locations (A, B, C, D and E) and the weight of the edges represents the time required in minutes to move between the locations.

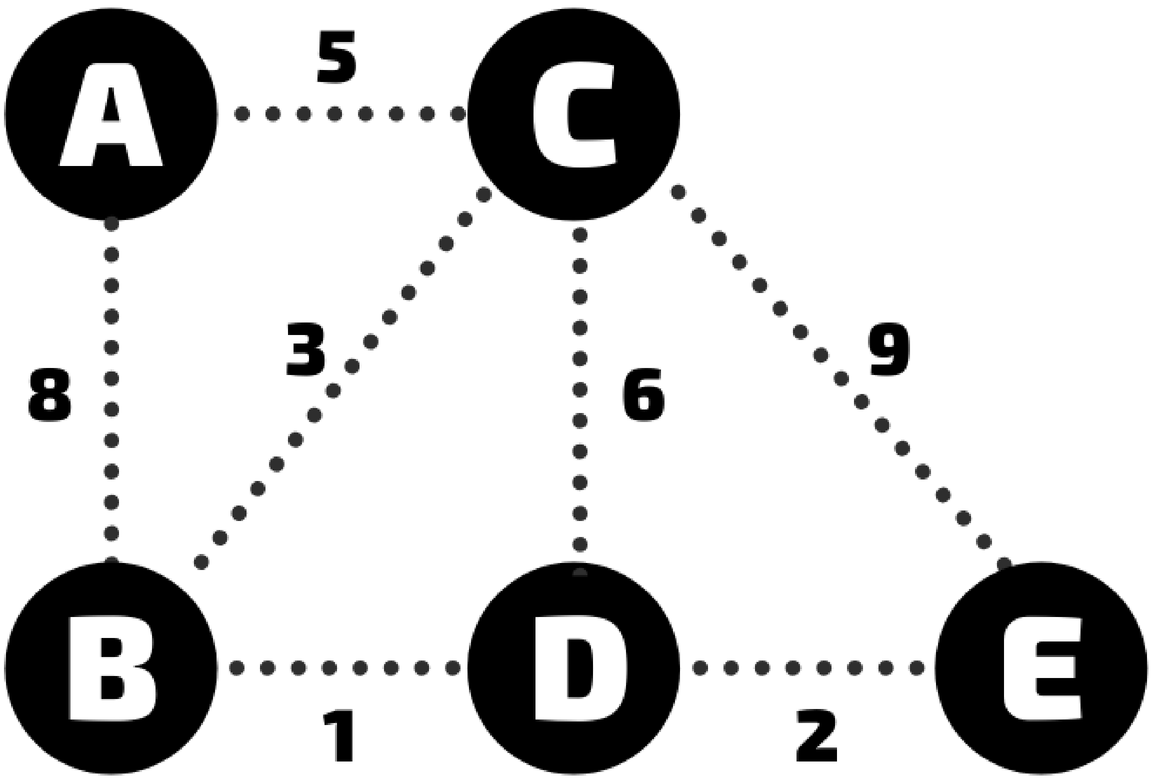


Figure 1: The weighted graph

Step 1

The algorithm starts by initialising the costs of all of the nodes to infinity (or a very large number) to show that the cost has not yet been calculated. The costs are updated as shorter paths are found.

- First create the **unvisited list** with the following headings
    - Node
    - Cost (from start)
    - Previous
- In the node column list all of the nodes, starting with the start node (labelled node A in this example)
- In the cost column write an infinity symbol (for each node).
- In the previous column write `none` (for each node).

  set the cost of the start node (A) to 0; there is no cost for A, as this is where you start from.

Unvisited list		
Node	**Cost (from start)**	**Previous**
A	0	none
B	∞	none
C	∞	none
D	∞	none
E	∞	none

Now create the **visited list**. This list is empty to begin with but it will be updated as the algorithm progresses. Write the same three headings:

- Node
- Cost (from start)
- Previous

Visited list		
Node	**Cost (from start)**	**Previous**

Step 2

- Pick the node that has the lowest cost (so far) in the unvisited list. The cost of A is currently the lowest in the unvisited list; all of the other nodes have a cost of infinity. **A becomes the current node**.
- Examine the nodes that can be reached directly from A (A's neighbours) **that have not yet been visited**:
    - The edge from A to B has a cost of 8. The cost currently recorded in your unvisited list is `infinity`. A cost of 8 is less than infinity, so you erase that and write 8 instead, and record the previous node as A.
    - The edge from A to C has a cost of 5. The cost currently recorded in your unvisited list is `infinity`. A cost of 5 is less than infinity, so you erase that and write 5 instead, and record the previous node as A.
- You have now evaluated the cost for all of the neighbours of the current node. Remove A from the unvisited list and add it to the visited list with its cost (0) and previous node (`none`).

Visited list		
Node	**Cost (from start)**	**Previous**
A	0	none

Unvisited list		
Node	**Cost (from start)**	**Previous**
A	0	none
B	~~∞~~ 8	~~none~~ A
C	~~∞~~ 5	~~none~~ A
D	∞	none
E	∞	none

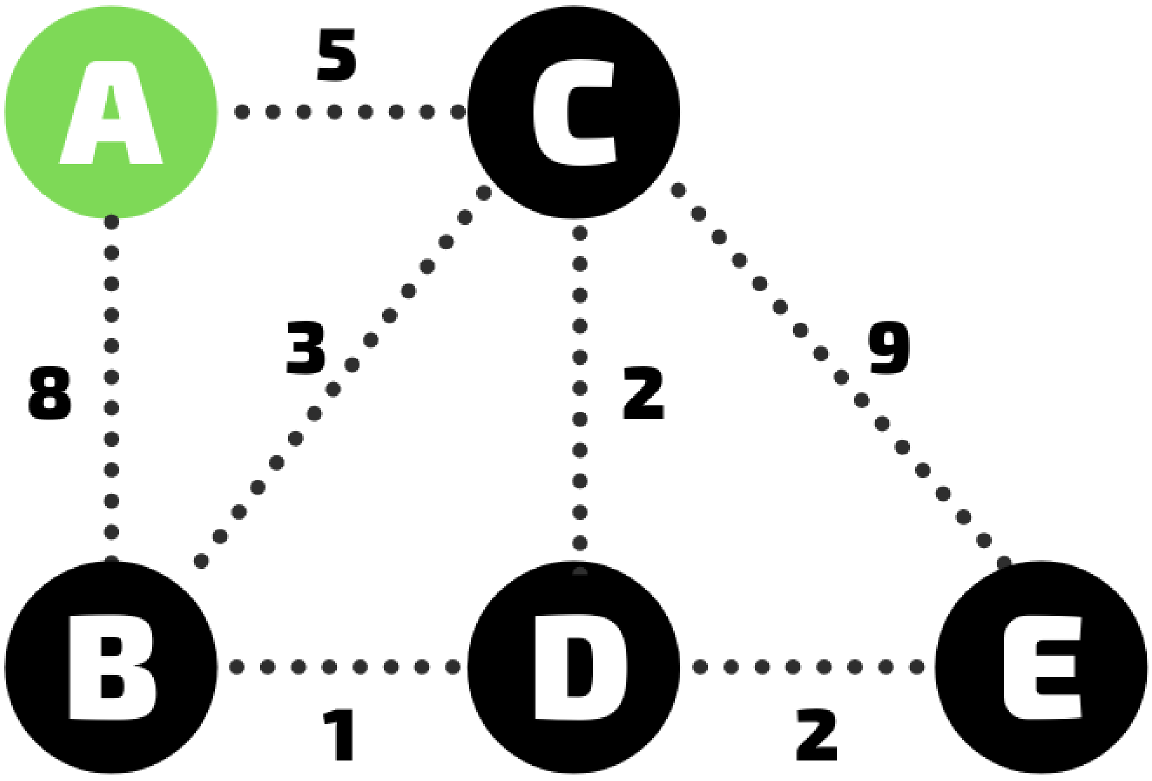


Figure 2: Node A has been visited.

Step 3

- Next you pick the node that has the shortest path (so far) from the start node as the next node to examine. The cost of the path from A to C is currently the lowest in the unvisited list; therefore, **C becomes the current node**.

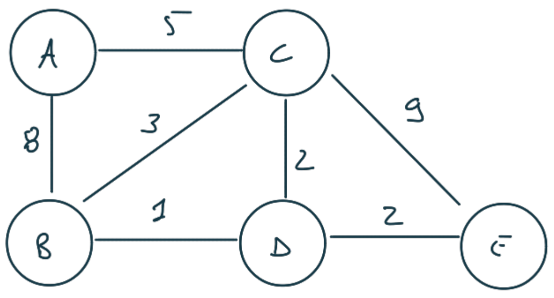
- Examine the nodes that can be reached directly from C (C's neighbours) **that have not yet been visited**. Note that you don't go back to nodes that you have already placed on the visited list.
    - The edge from C to B has a cost of 3. The cost from A to C is 5, so you add 3 and 5 to produce the total cost of using that path (i.e. to get from A to B via C), which is 8. Note that this is not lower than the current cost from A to B, **so you leave the entry in the unvisited list as it is**.
    - The edge from C to D has a cost of 6. The cost from A to C is 5, so you add 6 and 5 to produce the total cost of using that path (i.e. to get from A to D via C), which is 11. The cost currently recorded in your unvisited list for D is `infinity`, so you erase that and write 11 instead, with a previous node of C.
    - The edge from C to E has a cost of 9. The cost from A to C is 5, so you add 9 and 5 to produce the total cost of using that path (i.e. to get from A to E via C), which is 14. The cost currently recorded in your unvisited list for E is `infinity`, so you erase that and write 14 instead, with a previous node of C.

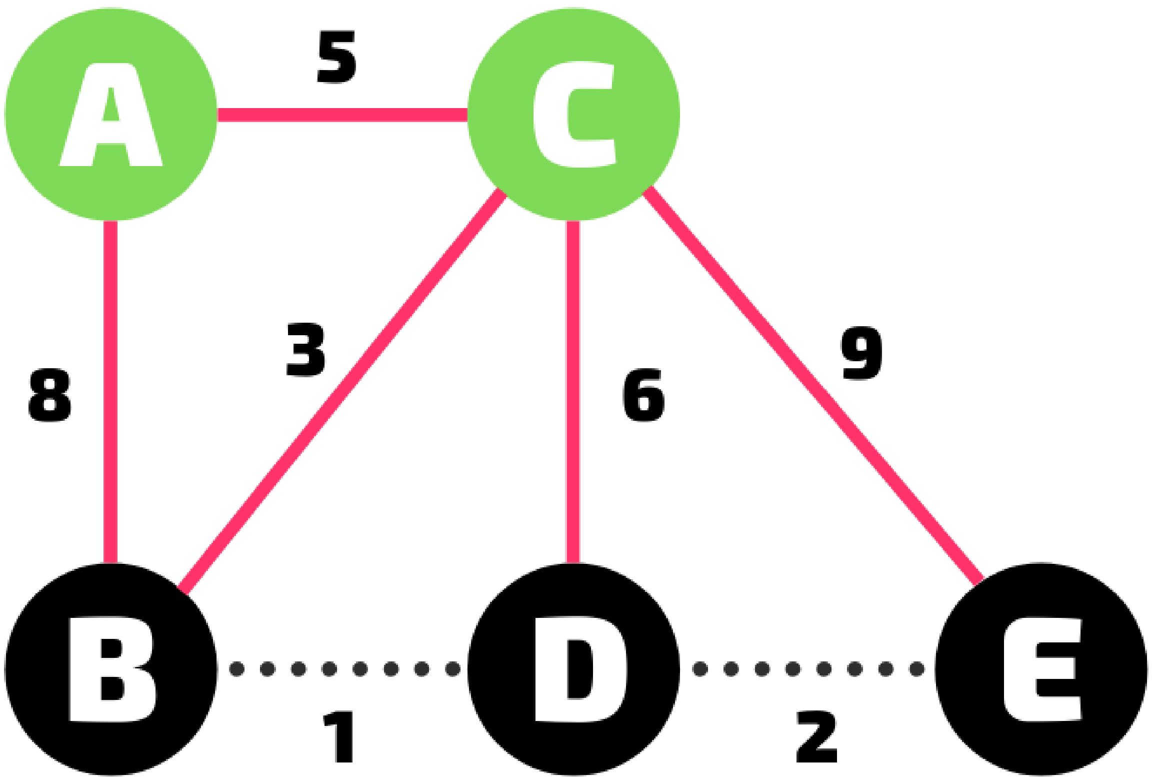
Note that by following this method, **you always keep the lowest cost in the unvisited list, which will help you eventually find the shortest path**.

You have now evaluated the cost for all of the neighbours of the current node. Remove C from the unvisited list and add it to the visited list with its cost (5) and previous node (A).

Visited list		
Node	**Cost (from start)**	**Previous**
A	0	none
C	5	A

Unvisited list		
Node	**Cost (from start)**	**Previous**
C	5	A
B	8	A
D	~~∞~~ 11	~~none~~ C
E	~~∞~~ 14	~~none~~ C





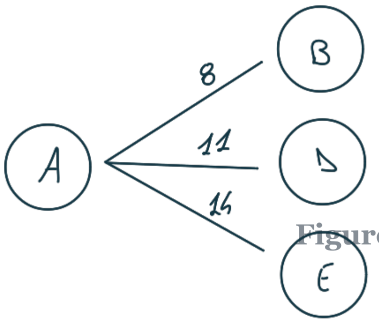


Figure 3: Node C has been visited.

Step 4

- Next you pick the node that has the shortest path (so far) from the start node as the next node to examine. The cost of the path from A to B is currently the lowest in the unvisited list; therefore, **B becomes the current node**.
- Examine the nodes that can be reached directly from B (B's neighbours) that have not yet been visited.

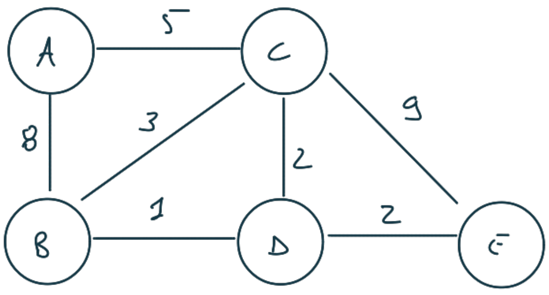
  The edge from B to D has a cost of 1. The cost for B from A is 8, so you add 1 and 8 to produce the total cost of using that path (i.e. to get from A to D via B), which is 9. The cost currently recorded in your unvisited list for D is 11, so you erase that and write 9 instead, with a previous node of B.

Note that by following this method, **you always keep the lowest cost in the unvisited list, which will help you eventually find the shortest path**.

You have now evaluated all of the connections from the current node. Remove B from the unvisited list and add B to the visited list with its cost (8) and previous node (A).

Visited list		
Node	**Cost (from start)**	**Previous**
A	0	none
C	5	A
B	8	A

Unvisited list		
Node	**Cost (from start)**	**Previous**
B	8	A
D	~~11~~ 9	~~C~~ B
E	14	C



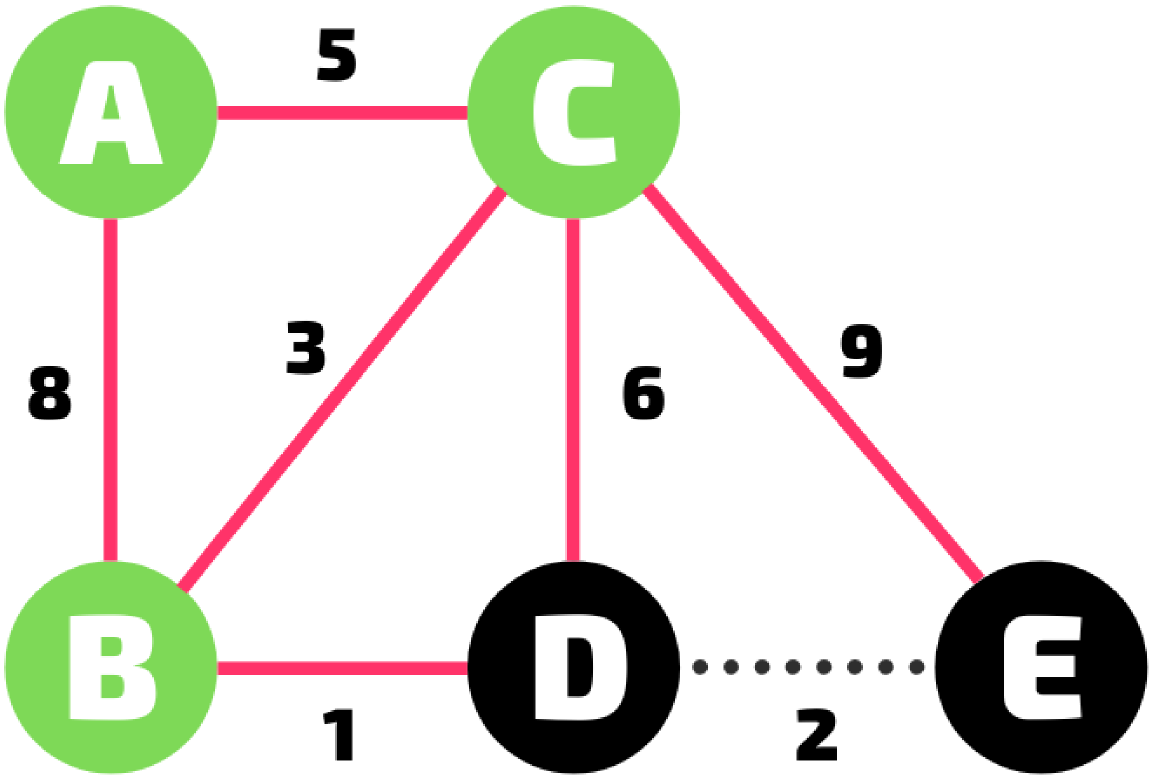


Figure 4: Node B has been visited.

Step 5

- Next you pick the node that has the shortest path (so far) from the start node as the next node to examine. The cost of the path from A to D is currently the lowest in the unvisited list; therefore, D becomes the current node.
- Examine the nodes that can be reached directly from D (D's neighbours) that have not yet been visited.

  The node from D to E has a cost of 2. The cost from A to D is 9, so you add 2 and 9 to produce the total cost of using that path (i.e. to get from A to E via D), which is 11. The cost currently recorded in your unvisited list for E is 14, so you erase that and write 11 instead, with a previous node of D.

- You have now evaluated all of the connections from the current node. Remove D from the unvisited list and add it to the visited list with its cost (9) and previous node (B).

Visited list		
Node	**Cost (from start)**	**Previous**
A	0	none
C	5	A
B	8	A
D	9	B

Unvisited list		
Node	**Cost (from start)**	**Previous**
D	9	B
E	~~14~~ 11	~~C~~ D

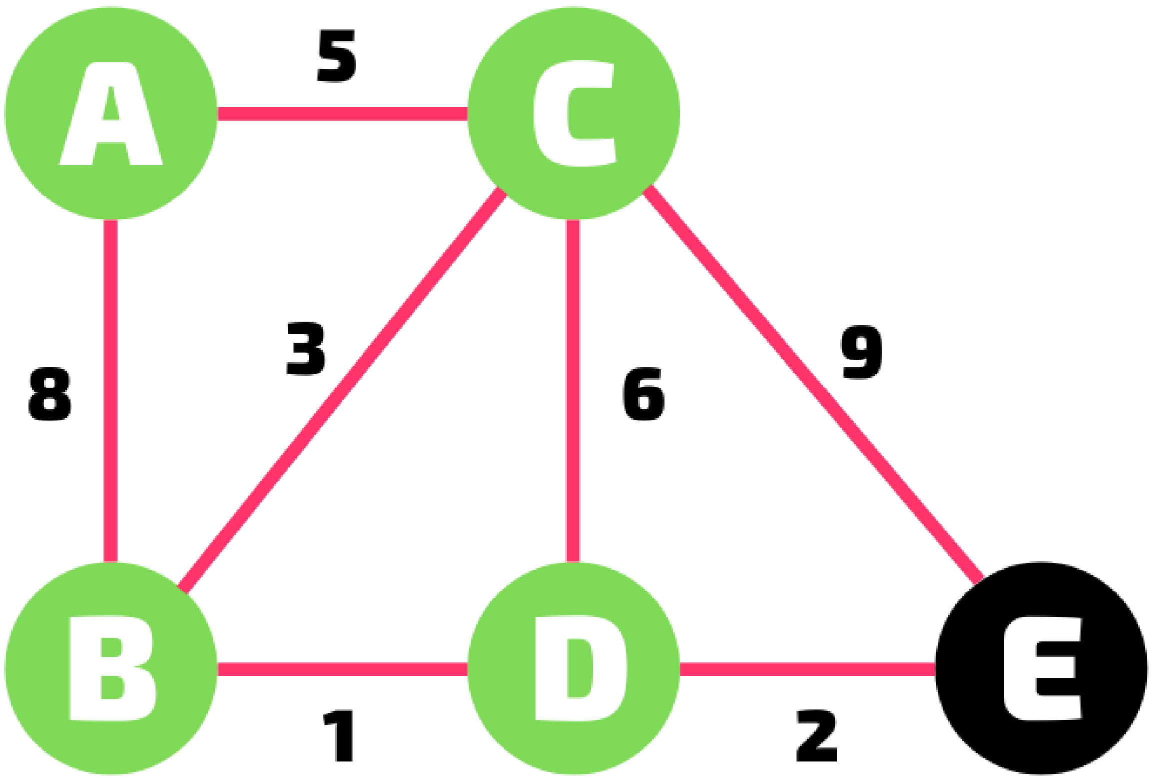


Figure 5: Node D has been visited.

Step 6

Now, the only remaining node in the unvisited list is E which has no more unvisited neighbours to examine. Therefore, you add it to the visited list with its cost (11) and previous node (D), and your work is done.

Visited list		
Node	**Cost (from start)**	**Previous**
A	0	none
C	5	A
B	8	A

Visited list		
D	9	B
E	11	D

Unvisited list		
Node	Cost (from start)	Previous
E	11	D

Finding the shortest path between two nodes

In the visited list you have a complete list of nodes, together with the cost of the shortest path to that node from the start node. You can recreate the shortest path from A to all other nodes by following the route back to the start node.

For example, to find the shortest path from A to E:

- first, you look at E and see that the previous node is D
- from D, the previous node is B
- from B, the previous node is A

Therefore, the shortest path from A to E is A → B → D → E and the cost is 11.

Similarly, to find the shortest path from A to D:

- first, you look at D and see that the previous node is B
- from B, the previous node is A

Therefore, the shortest path from A to D is A → B → D and the cost is 9.

Useful tip

- Note that not all nodes have to be included in the shortest path. In this example, C is not included in the shortest path from A to E.
- The shortest path is not necessarily the path with the fewest nodes. For example the path A → C → E has fewer nodes than A → B → D → E but a cost of 14 and therefore it is not the shortest path from A to E.

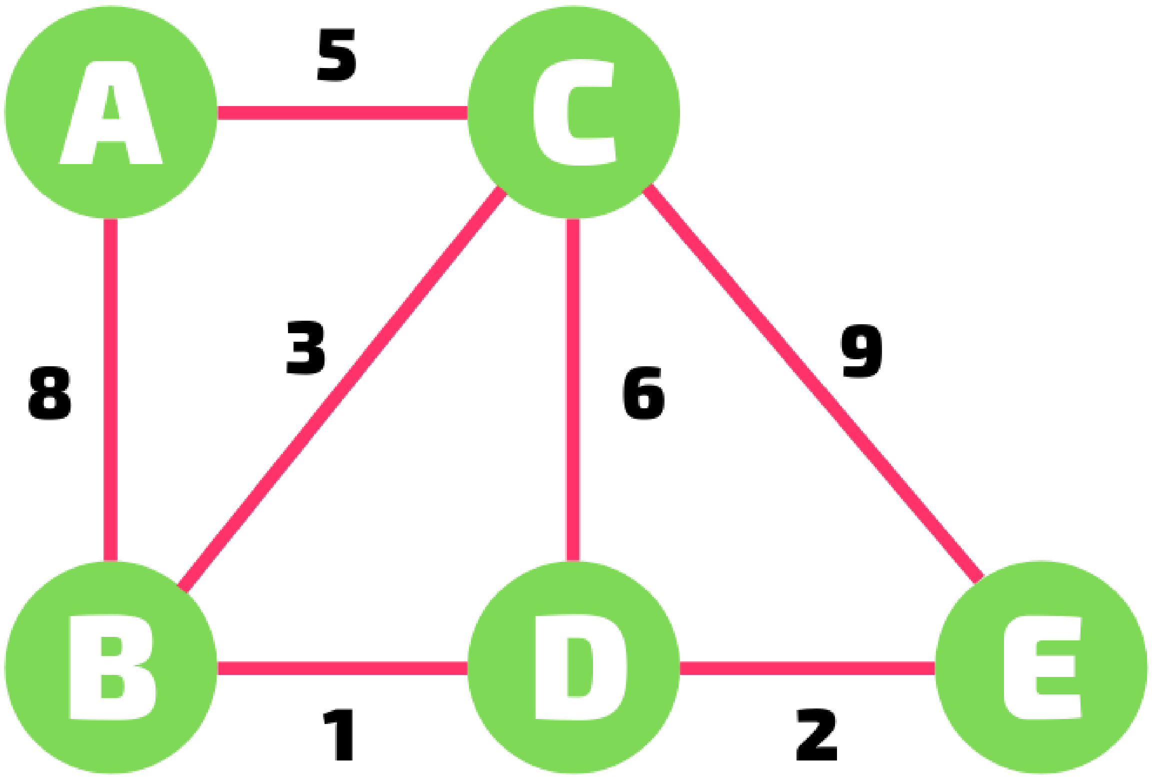


Figure 6: All nodes have been visited.

Try it yourself!

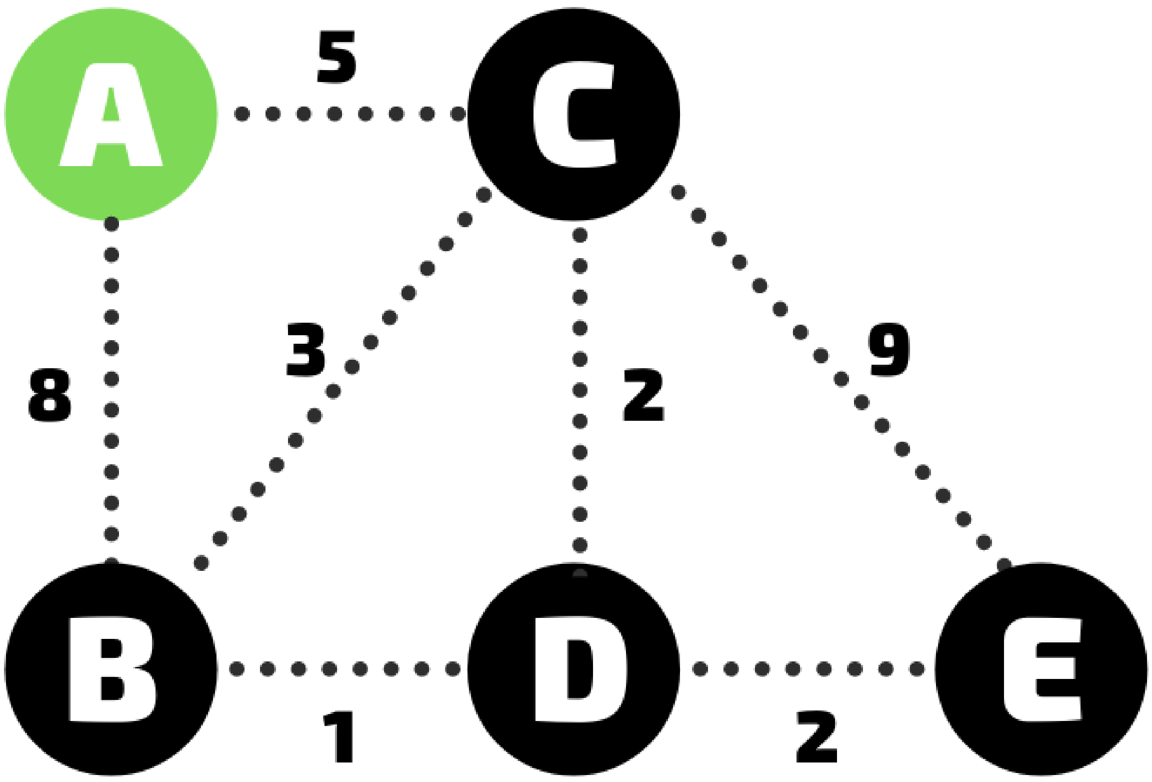


Figure 2: Graph for quick question

The graph above illustrates the connections between five nodes. Using Dijkstra's algorithm, find the shortest path from the start node **A** to all other nodes. Write down your answer as a list showing:

- the node
- the cost of the shortest path (from the start node)
- the previous node

Try it yourself!

Given the following set of information produced by running Dijkstra's algorithm, in the form of a visited list, what is the route of the shortest path from A to G?

Visited list

Visited list		
Node	**Cost (from start)**	**Previous**
A	0	none
C	3	A
F	9	C
E	10	F
B	12	A
D	13	E
G	15	E

$A \to C \to F \to E \to G$

Implementation considerations

There will be several complexities to consider before you try to implement the algorithm. Here are a few of them:

- how to store the graph
- how to store the unvisited list
- how to store the visited list
- what value to use to represent infinity
- what value to use to represent the absence of a previous node

Data structures

You have aready learned that a graph can be stored as an adjacency matrix or adjacency list. The underlying structure here could be an array or a dictionary as you will need direct access to the nodes.

The unvisited and visted lists could also be stored as arrays or dictionaries. However, it is common to see a priority queue used as a data structure to hold the unvisited list. In this case, the queue would be prioritised so that the node with the lowest cost (so far) was at the front of the queue. This makes it easier to access the next node to examine, as you do not need to sort or search the list. You can simply take the item from the head of the priority queue.

Representing values

In your choice of programming language, you will find ways of representing very high values. For example, in Python, `sys.maxsize` will return the largest positive integer supported by the platform. Similarly in Python, `None` can be used to repesent a null value.

Dijkstra's algorithm in pseudocode

In the pseudo code version of the algorithm that follows, a dictionary has been chosen to store each of:

- the graph
- visted list
- unvisited list

A dictionary is a good choice here, as the value of the node can be used as the key. This allows the data for the node to be accessed directly (by key), rather than having to search for it.

For `visited` and `unvisited`, the data stored for each node is a list containing the cost and the value of the previous node. So, if `visited` looks like this:

`{'A': [0, NULL], 'B': [5,'A'], 'C': [8,'A'], 'D': [9,'C'], 'E': [11, 'D']}`

the syntax `visited[key]` will return a list containing the cost and the previous node for the specified key value. For example `visited['E']` will return `[11, 'D']`. The cost and the previous node can be accessed indivdually by position, and here constants are used (`COST` and `PREVIOUS`) to help make the code self documenting. As an example, `visited['E'][COST]` will return the integer value 11 and `visited['E'][PREVIOUS]` will return the node D.

For the graph, the data stored for each node is another dictionary. Here is an example of some graph data:

```
graph = {'A': {'B':8, 'C':5},
         'C': {'A':5, 'D':6, 'E':9},
         'B': {'A':8, 'D':1},
         'D': {'C':6, 'B':1, 'E':2},
         'E': {'C':9, 'D':2}
         }
```

In the pseudocode that follows:

- if `current_node` is `'A'`, the statement `graph[current_node]`, will return `{'B':8, 'C':5}`.
- if the value of `neighbour` is `'B'`, the statement `graph[current_node][neighbour]` will return the integer value 8.

```
COST = 0  // Used to index cost
PREVIOUS = 1  // Used to index previous node

FUNCTION dijkstras_shortest_path(graph, start_node)

    // Initialise visited and unvisited lists
    unvisited = {}  // Declare unvisited list as empty dictionary
    visited = {}  // Declare visited list as empty dictionary

    // Add every node to the unvisited list
    FOREACH key IN graph
        // Set distance to infinity and previous to Null value
        unvisited[key] = [∞, NULL]
    NEXT key
    // Set the cost of the start node to 0
    unvisited[start_node][COST] = 0

    // Repeat the following steps until unvisited list is empty
    finished = False
    WHILE finished == False
        IF LEN(unvisited) == 0 THEN
            finished = True  // No nodes left to evaluate
        ELSE
            // Get unvisited node with lowest cost as current node
            current_node = get_minimum(unvisited)
            // Examine neighbours
            FOREACH neighbour IN graph[current_node]
                // Only check unvisited neighbours
                IF neighbour NOT IN visited THEN
                    // Calculate new cost
                    cost = unvisited[current_node][COST] + graph[current_node]
[neighbour]
                    // Check if new cost is less
                    IF cost < unvisited[neighbour][COST] THEN
                        unvisited[neighbour][COST] = cost  // Update cost
                        unvisited[neighbour][PREVIOUS] = current_node  // Update
previous
                    ENDIF
                ENDIF
            NEXT neighbour
            // Add current node to visited list
            visited[current_node] = unvisited[current_node]
            // Remove from unvisited list
            DEL(unvisited[current_node])
        ENDIF
    ENDWHILE
    RETURN visited
ENDFUNCTION
```

Displaying the shortest path between two nodes

The following pseudocode will display the actual route (as well as the cost) of the shortest path between the start node and all other nodes in the graph. The subroutine accepts two arguments: `start_node` and `visited`, as described in the pseudocode for the main algorithm (above).

```
FUNCTION display_shortest_paths(start, visited)
    FOREACH key IN visited
        IF key != start THEN  // Don't print path for start node
            current = key
            path = current
            WHILE current != start
                previous = visited[current][PREVIOUS]  // Get value of previous
node
                path = previous + path  // Add value to path
                current = visited[current][PREVIOUS]  // Backtrack
            ENDWHILE
            PRINT("Path for: " + key)
            PRINT(path)
            PRINT("Cost: " + visited[key][COST])
        ENDIF
    NEXT key
ENDFUNCTION
```

Dijkstra’s algorithm works well as a pathfinding algorithm for a whole set of nodes but, when using it to search for the shortest path to one specific target, it can be inefficient.

The algorithm's directive is to always follow the shortest path from its unvisted list, regardless of the direction or proximity to the target.

The algorithm can be tweaked to add an additional check each time it takes a node from the unvisited list: if this is the target node, then the algorithm can stop after processing the node, and can return the information needed. However, other pathfinding algorithms, such as A* (OCR only) can provide much more efficient ways of finding the shortest path to a specific target node.