# OCR Computer Science A Level

## 2.3.1 Path Finding Algorithms

Intermediate Notes

**Specification:**

- Dijkstra's shortest path algorithm
- A* algorithm

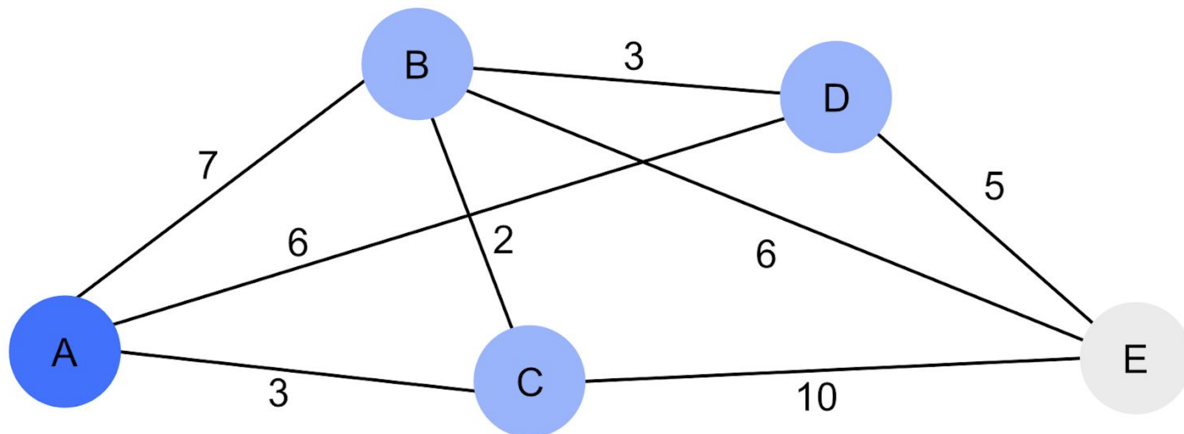
## Dijkstra's algorithm

Dijkstra's algorithm finds the shortest path between two nodes in a weighted graph. Remember that graphs are used as an abstraction for a variety of real life scenarios, which means nodes and edges can represent different entities. For example, graphs can model networks, with nodes representing devices and arc costs representing network traffic.

Dijkstra's algorithm is commonly implemented using a priority queue, with the smallest distances being stored at the front of the list.

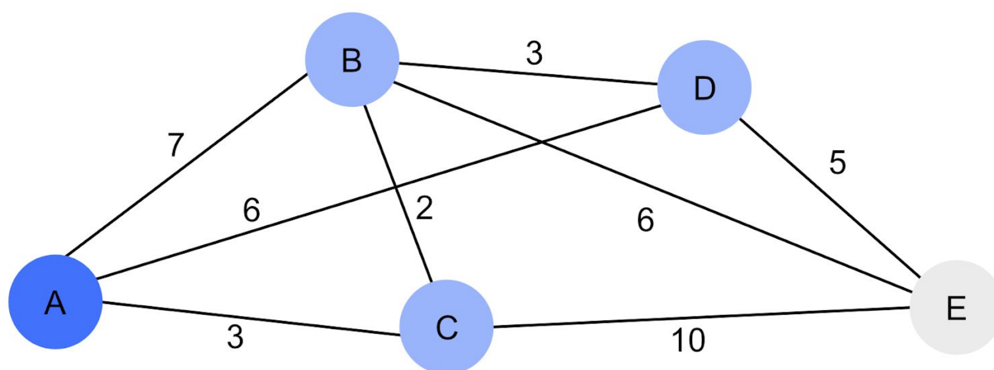
Below is a step-by-step example of using Dijkstra's algorithm to find the shortest path between A and E. You may be asked to demonstrate this in an exam.

-




Priority Queue

| C = 3 | D = 6 | B = 7 | E = ∞ | |
|---|---|---|---|---|




Priority Queue

| C = 3 | D = 6 | B = 7 | E = ∞ | |
|---|---|---|---|---|

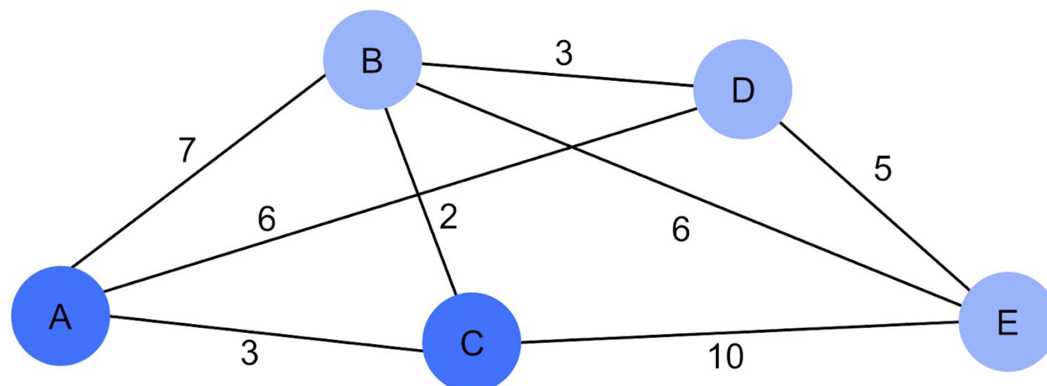
<u>Step 1</u>

Starting from the root node (A), add the distances to all of the immediately neighbouring nodes (B, C, D) to the priority queue. All of the remaining nodes that cannot be reached are shown in grey and their distance is denoted by ∞.

When asked to demonstrate using Dijkstra's algorithm to find the distance between two nodes, it is helpful to use the table below. This allows you to keep track of visited nodes and distances in an organised way, which is particularly important for solving more complex problems which can get very confusing!

Begin by filling in the 'Node' column with the nodes connected to the root node. The 'From' column should contain the node that you are travelling from, and the 'Distance' column contains the distance between these two nodes. The 'Total distance' is the sum of the distances from the root node to that particular node. The node which is the shortest distance away from the root node is highlighted and selected for the next stage.

| Node | From | Distance | Total distance |
|---|---|---|---|
| B | A | 7 | 7 |
| C | A | 3 | 3 |
| D | A | 6 | 6 |




AC = 3

Priority Queue

| B = 5 | D = 6 | E = 13 | | |
|---|---|---|---|---|

<u>Step 2</u>

Remove the node the shortest distance away, from the front of the queue. Now traverse all of the nodes connected to the removed node C.

If the total distance passing through the removed node to the neighbouring node is smaller than the distance currently stored with this node, update this value to the smaller distance. In this case, C has two neighbours: B and E. The shortest total cost of travelling to E so far is therefore 13, as it is less than the infinite value previously allocated to E. Travelling from A to C to B adds up to a total cost of 5, while travelling from A to B has a cost of 7. The cost value associated with B is therefore updated to 5.

We have visited B, so can ignore it. We can also remove routes that are not the shortest way to get to a node, so the top row can be ignored. The new shortest path is highlighted.

| Node | From | Distance | Total distance |
|---|---|---|---|
| ~~B~~ | ~~A~~ | ~~7~~ | ~~7~~ |
| ~~C~~ | ~~A~~ | ~~3~~ | ~~3~~ |
| D | A | 6 | 6 |
| B | C | 2 | 5 |
| E | C | 10 | 13 |

Step 3

Continue repeating Step 2 until the goal node has been reached. This time B is removed from the front of the queue and the distances in the queue are updated. From B, we can only travel to D. As the shortest path to B so far is 5, and the path from B to D is 3, the total cost of travelling to D is 8. This is greater than the cost recorded in the queue already, 6, so this route can be ruled out.

| Node | From | Distance | Total distance |
|---|---|---|---|
| ~~B~~ | ~~A~~ | ~~7~~ | ~~7~~ |
| ~~C~~ | ~~A~~ | ~~3~~ | ~~3~~ |
| D | A | 6 | 6 |
| ~~B~~ | ~~C~~ | ~~2~~ | ~~5~~ |
| E | C | 10 | 13 |
| ~~D~~ | ~~B~~ | ~~3~~ | ~~8~~ |

Once again, we repeat this same process for node D. The only node connected to D is E, and the total cost of travelling from A to E is 11. This is shorter than the previous route to E, which cost 14. The value of E can thus be updated to 11.

| Node | From | Distance | Total distance |
|---|---|---|---|
| ~~B~~ | ~~A~~ | ~~7~~ | ~~7~~ |
| ~~C~~ | ~~A~~ | ~~3~~ | ~~3~~ |
| D | A | 6 | 6 |
| ~~B~~ | ~~C~~ | ~~2~~ | ~~5~~ |
| ~~E~~ | ~~C~~ | ~~10~~ | ~~13~~ |
| ~~D~~ | ~~B~~ | ~~3~~ | ~~8~~ |
| E | D | 5 | 11 |

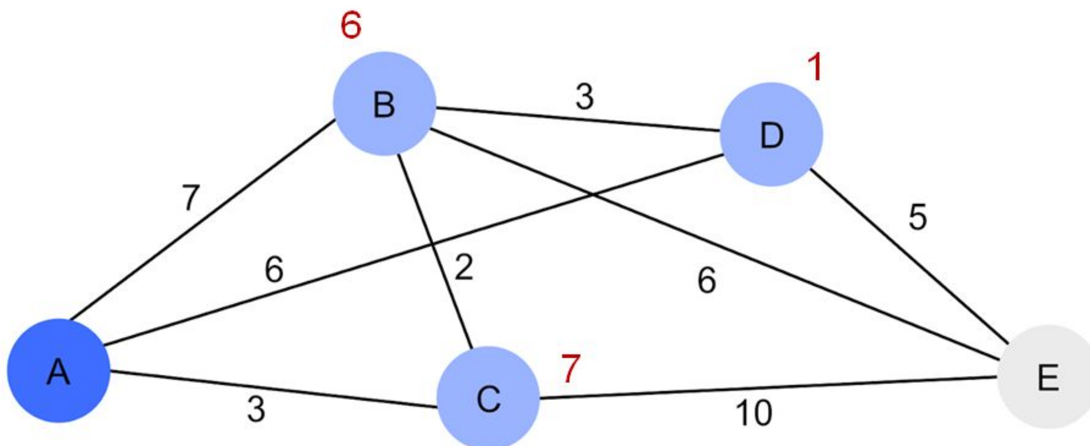
As we have now visited all of the nodes on the graph, we can confirm by tracing back through the table above that the shortest path is ADE.

## A* algorithm

The A* algorithm is a general path-finding algorithm which is an improvement of Dijkstra's algorithm and has two cost functions:

1) The first cost function is the actual cost between two nodes. This is the same cost as is measured in Dijkstra's algorithm.
2) The second cost function is an approximate cost from node x to the final node. This is called a heuristic, and aims to make the shortest path finding process more efficient. The approximate cost might be an estimate of the length between x and the final node, calculated using trigonometry.

When calculating the distance between two nodes using the A* algorithm, the approximate cost is added onto the actual cost. This is used to determine which node is visited next. The heuristic costs are labelled in red on the diagram.




Step 1

When working out the shortest distance using the A* algorithm, an extra column is required to store the heuristic. The method used here is very similar to the method used in Dijkstra's algorithm, with the exception that the heuristic cost is added onto the actual cost to calculate the total cost. Again, the route with the lowest total cost is selected to traverse further.

| Node | From | Distance | Heuristic | Total distance |
|---|---|---|---|---|
| B | A | 7 | 6 | 13 |
| C | A | 3 | 7 | 10 |
| D | A | 6 | 1 | 7 |

Step 2

The node D is then selected. Note that the heuristic cost of the previous node is not added on to the new distance, thus giving a total distance of 11 travelling from A to D to E. As 11 is the shortest total distance, the algorithm terminates. The shortest route is found to be ADE, at a cost of 11.

| Node | From | Distance | Heuristic | Total distance |
|---|---|---|---|---|
| B | A | 7 | 6 | 13 |
| C | A | 3 | 11 | 14 |
| D | A | 6 | 12 | 18 |
| E | D | 5 | - | 11 |

As you can see, the heuristics used here allow the shortest path to be found much quicker than when using Dijkstra's algorithm. How effective the A* algorithm is, however, depends largely on the accuracy of the heuristics used.