A* search algorithm

isaaccomputerscience.org/concepts/dsa_search_a_star

The **A* search algorithm**, builds on the principles of Dijkstra's shortest path algorithm to provide a faster solution when faced with the problem of finding the shortest path between two nodes. It achieves this by introducing a heuristic element to help decide the next node to consider as it moves along the path. You can read more about heuristics in the topic on complexity.

Dijkstra's algorithm finds the shortest path between the start node and all other nodes. As well as being faster, the A* algorithm differs from Dijkstra's in that it seeks only the shortest path between the start node and the target node.

Heuristics, sometimes called heuristic functions, are used to provide 'good enough' solutions to very complex problems where finding a perfect solution would take too much time. When you use heuristics, you trade accuracy, correctness, and exactness for speedy processing.

One of the drawbacks with Dijkstra's algorithm is that it can (and will) evaluate paths that will never provide the shortest option. Imagine tring to find the shortest route on a map between London and Edinburgh. Anyone with a reasonable grasp of UK geography, would not bother to evaluate a route that went via Plymouth. The trade off between speed and accuracy is important. In some applications, accuracy is less important than computational time. For example, in a SatNav application a route that is calculated in seconds and is "short enough" is preferable to having to wait 10 minutes for the perfect route.

The A* algorithm uses a heuristic function to help decide which path to follow next. The heuristic function provides an **estimate** of the minimum cost between a given node and the target node. The algorithm will combine the **actual cost from the start node** - referred to as `g(n)` - with the **estimated cost to the target node** - referred to as `h(n)` - and uses the result to select the next node to evaluate. This is explained in more detail in the step-by-step method that follows.

Choosing a heuristic function

There is no single best heuristic to use in path finding, as every application is different. For example, if the cost relates to a distance, it could be estimated by calculating the "straight line" distance between the nodes, perhaps by using one of the following methods:
Euclidean distance, Manhattan distance, great circle distance
However, do remember that the weights on a graph do not always represent distance.

It is very important that the heuristic function does **not overestimate** costs. However, so long as the heuristic function provides an estimate that is less than or equal to the actual cost, A* will always find an optimal path and will generally find it much faster than

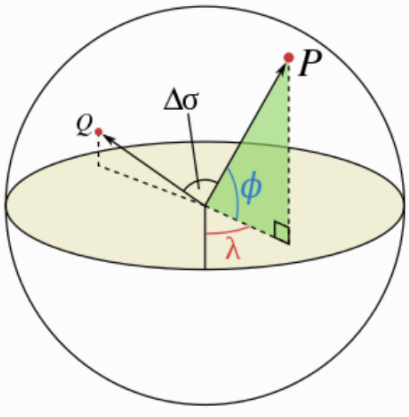
Euclidean distance

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Manhattan distance

$$d(p,q) = \sum_{i=1}^{n} |q_i - p_i|$$

Great circle distance



low precision, big angles

$$\Delta\sigma = \arccos\left(\sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \cos(\Delta\lambda)\right).$$

high precision, small angles

$$\begin{aligned}\Delta\sigma &= \operatorname{archav}(\operatorname{hav}(\Delta\phi) + (1 - \operatorname{hav}(\Delta\phi) - \operatorname{hav}(\phi_1 + \phi_2)) \cdot \operatorname{hav}(\Delta\lambda)) \\ &= 2\arcsin\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \left(1 - \sin^2\left(\frac{\Delta\phi}{2}\right) - \sin^2\left(\frac{\phi_1 + \phi_2}{2}\right)\right) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)}.\end{aligned}$$

Dijkstra's Algorithm.

In this example, you will consider a small graph and use the A* algorithm to find the shortest path from **A to F**.

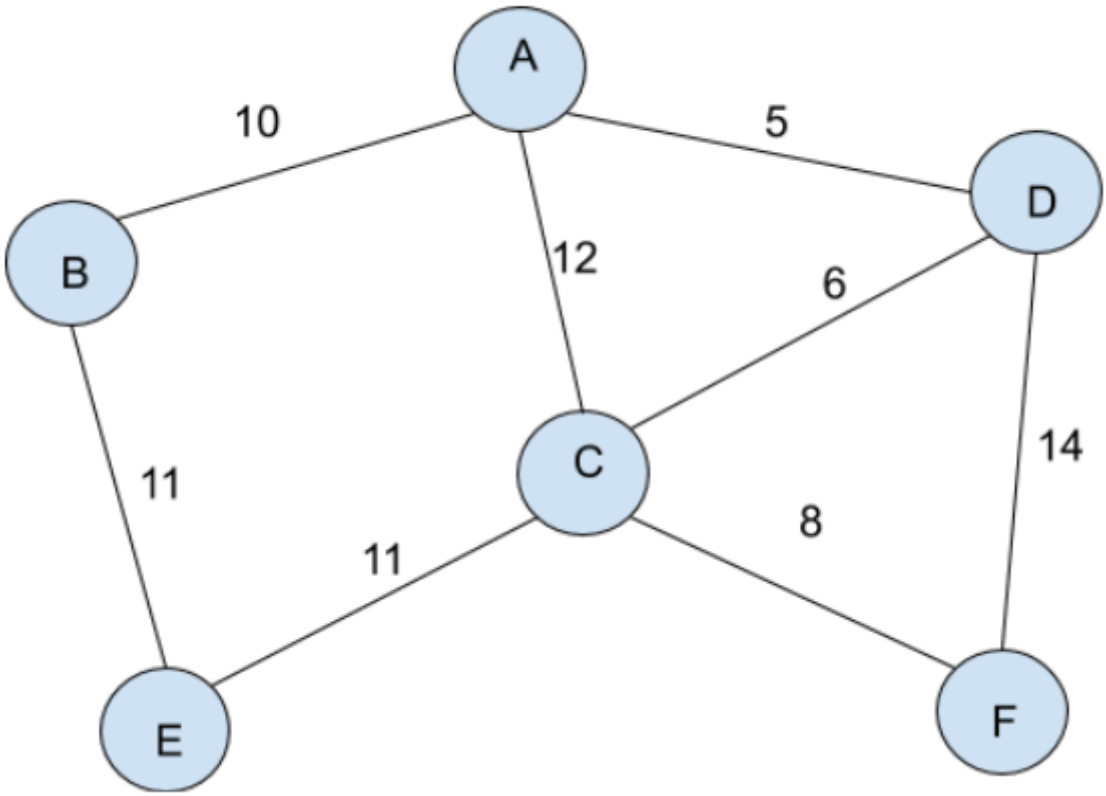


Figure 1: A weighted graph

If you have not yet already studied Dijkstra's shortest path algorithm, it is recommended that you start there, as much of the terminology is introduced and explained in that topic.

The heuristic function that is used here is a **black box**. This means that you do not see the algorithm (the detail is abstracted so that there is one less thing to think about). However, the function will return the estimated distance from the given node to the target node (F), as follows:

Node	Estimated cost (to target node)
A	10
B	15
C	5
D	5
E	10

F	0

On the graph, these heuristic values appear adjacent to the node so that you have all of the important information (that you need to follow the method) in one place.

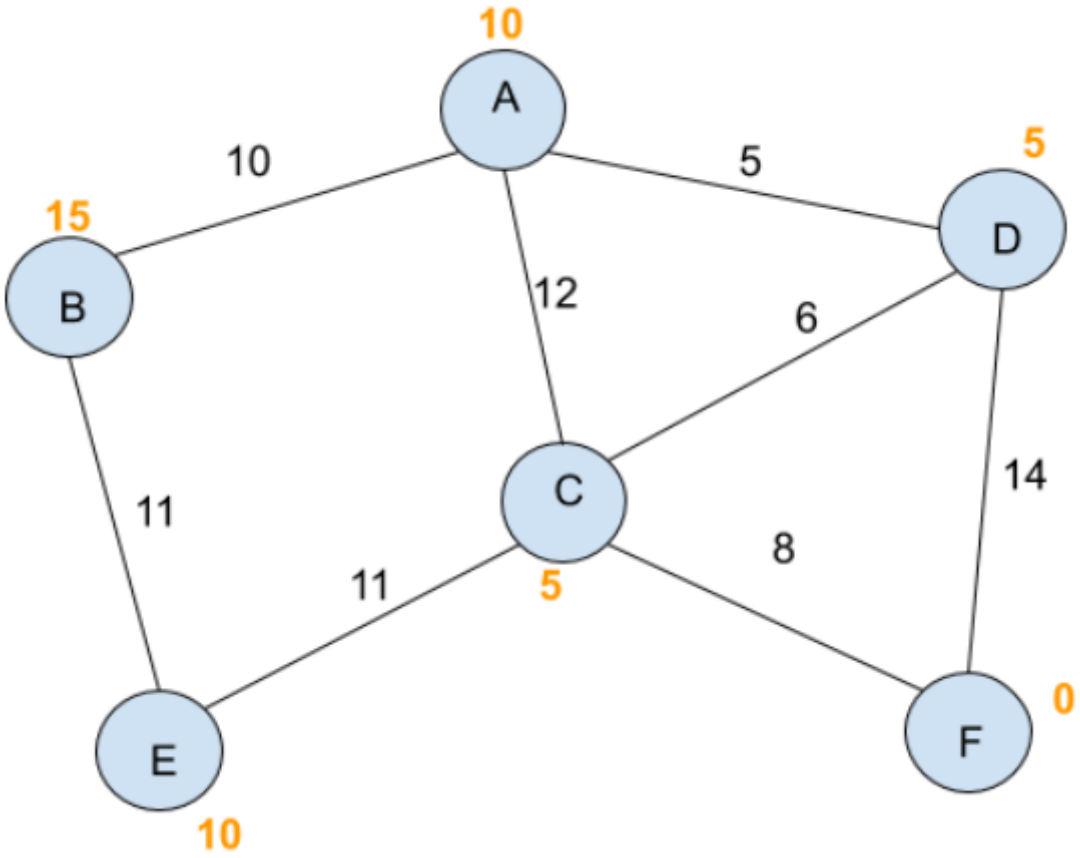


Figure 2: Graph annotated with heuristic values

`g-score` and `f-score`

In Dijkstra's algorithm, you kept track of the cost of each path as you evaluated all of the possible routes. Every time a shorter path was found, you updated the cost and the previous node.

In A*, the cost of the route from the start node is calculated, stored and updated in a similar way but is referred to as `g-score`. There is also an `f-score`, which is the cost of the path (so far) + the estimated cost (provided by the heuristic) of reaching the target node. **It is the f-score (rather than the g-score) that is used to select the next current node**.

Step 1

The algorithm starts by initialising the `g-score` of all of the nodes to infinity (or a very large number) to show that the score has not yet been calculated. The value for `f-score` can also be set to infinity.

- Create the **unvisited list** with the following headings
    - node
    - g-score (cost from start)
    - f-score (cost from start + heuristic)
    - previous
- In the node column list all of the nodes, starting with the start node (labelled node A in this example)
- set the `g-score` of the start node (A) to 0 and the `f-score` to 10. This is the value returned by the heuristic function for node A.
- for all other nodes, set the `g-score` and `f-score` to `infinity` and previous to `none`.

Unvisited list			
node	**g-score**	**f-score**	**previous**
A	0	10	none
B	∞	∞	none
C	∞	∞	none
D	∞	∞	none
E	∞	∞	none
F	∞	∞	none

open list

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

Visited list			
node	**g-score**	**f-score**	**previous**

closed list

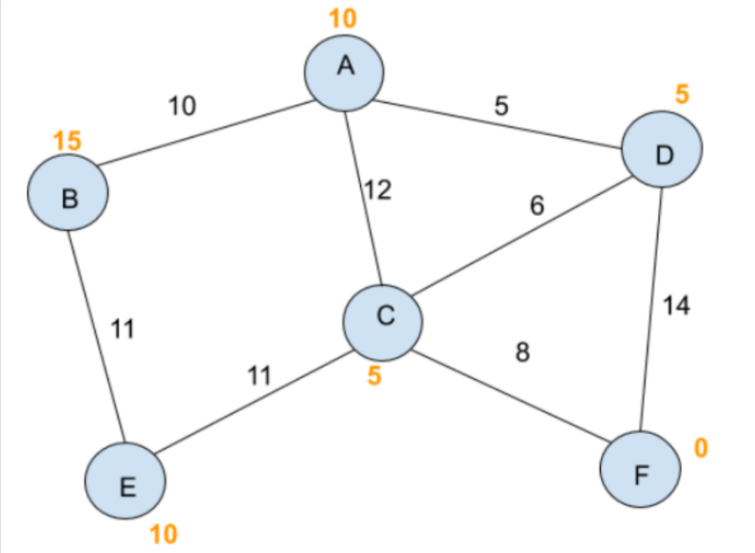
Step 2

- **Pick the node that has the lowest** `f-score` in the unvisited list. The `f-score` 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 weight of 10. The `g-score` currently recorded in your unvisited list is `infinity`. A value of 10 is less than infinity, so you erase that and write 10 instead, and record the previous node as A. The heuristic `h(B)` is 15. The `f-score` is calculated by adding the heuristic value to `g-score`; this gives you an `f-score` for node B of 25 (10 + 15 = 25) and this is recorded in the unvisited list.
    - The edge from **A to C** has a weight of 12. The `g-score` currently recorded in your unvisited list is `infinity`. A cost of 12 is less than infinity, so you erase that and write 12 instead, and record the previous node as A. The heuristic `h(C)` is 5. This gives you an `f-score` for node C of 17 (12 + 5 = 17) which you record.
    - The edge from **A to D** has a weight of 5. The `g-score` 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. The heuristic `h(D)` is 5. This gives you an `f-score` for node B of 10 (5 + 5 = 10) which you record.
- You have now evaluated all of the neighbours of the current node. Remove A from the unvisited list and add it to the visited list with its `g-score` (0), `f-score` (10) and previous node (A).

Visited list			
node	**g-score**	**f-score**	**previous**
A	0	10	none

Unvisited list			
node	**g-score**	**f-score**	**previous**
B	~~∞~~ 10	~~∞~~ 25	A
C	~~∞~~ 12	~~∞~~ 17	A
D	~~∞~~ 5	~~∞~~ 10	A
E	∞	∞	none
F	∞	∞	none



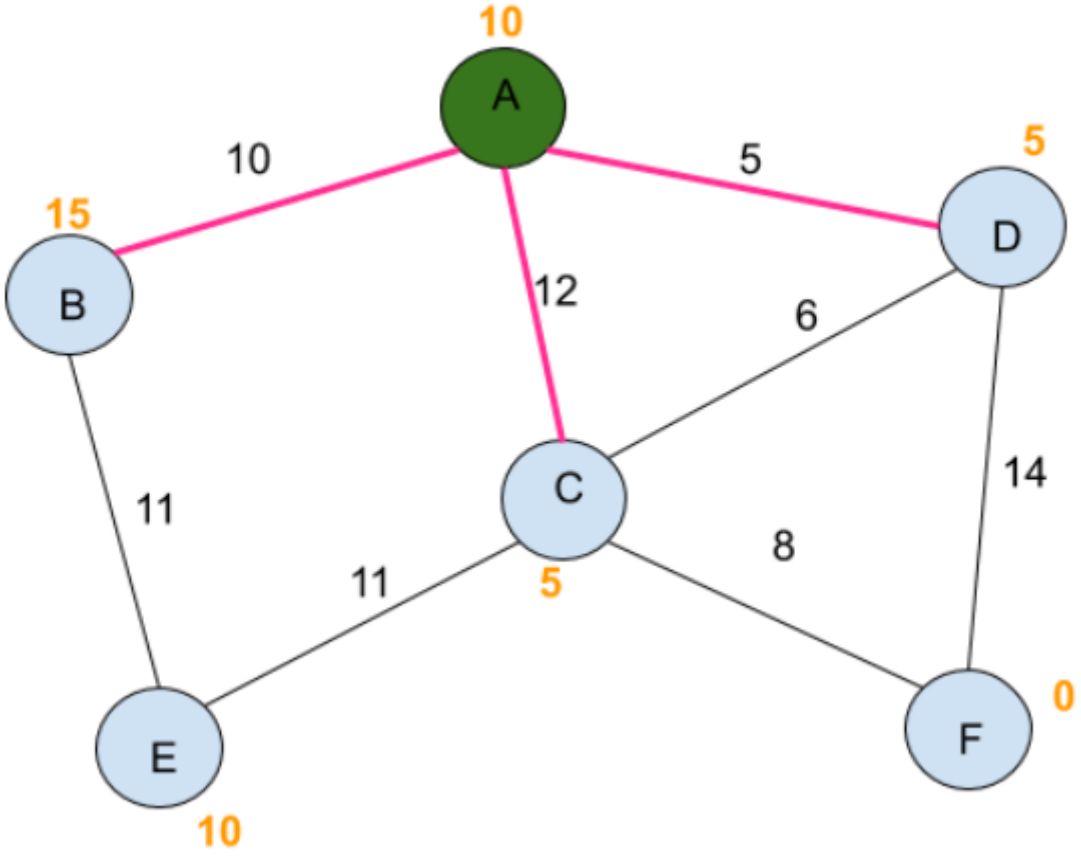


Figure 3: Node **A** has been fully evaluated

Step 3

- Pick the node that has the lowest `f-score` in the unvisited list. The `f-score` of D (10) is currently the lowest in the unvisited list. **D becomes the current node**.
- Examine the nodes that can be reached directly from D (D'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 **D to C** has a weight of 6. The `g-score` for D is 5, so you add 5 and 6 to produce the cost (`g-score`) of using that path (i.e. to get from A to C via D), which is 11. This is lower than the current `g-score` for C, **so you update the entry in the unvisited list** and record the previous node as **D**. The heuristic `h(C)` is 5. This gives you a new `f-score` for node C of 16 (11 + 5 = 16); you update the `f-score` for C in the unvisited list.
    - The edge from **D to F** has a weight of 14. The g-score for D is 5, so you add 5 and 14 to produce the cost (`g-score`) of using that path (i.e. to get from A to F via D), which is 19. The `g-score` currently recorded for F is `infinity` ; 19 is less than infinity, so you erase that and write 19 instead, and record the previous node as D. The heuristic `h(F)` is 0. This gives you a new `f-score` for node F of 19 (19 + 0 = 19); this is updated in the visited list.

Note that by following this method, **you always maintain the lowest values for each node's `g-score` and `f-score` in the unvisited list**, which will help you eventually find the shortest path.

> You have now evaluated all of the neighbours of the current node. Remove D from the unvisited list and add it to the visited list with its `g-score` (5), `f-score` (10) and previous node (A).

Visited list			
node	**g-score**	**f-score**	**previous**
A	0	10	None
D	5	10	A

Unvisited list			
node	**g-score**	**f-score**	**previous**
B	10	25	A
C	~~12~~ 11	~~17~~ 16	~~A~~ D
E	∞	∞	none
F	~~∞~~ 19	~~∞~~ 19	D

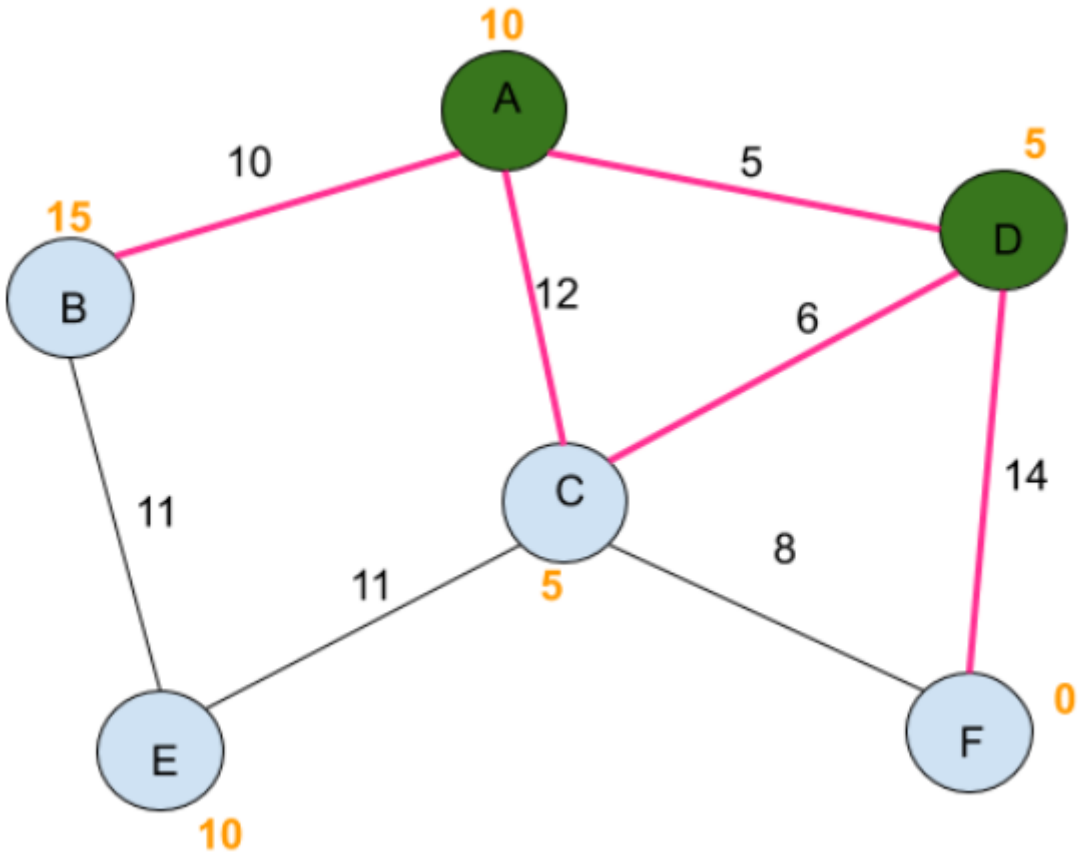


Figure 4: Node **D** has been fully evaluated

Step 4

- Pick the node that has the lowest `f-score` in the unvisited list. The `f-score` of C (16) is currently the lowest in the unvisited list. **C becomes the current node**.
- Examine the nodes that can be reached directly from C (C's neighbours) **that have not yet been visited**.
    - The edge from **C to E** has a weight of 11. The `g-score` for C is 11, so you add 11 and 11 to produce the cost (`g-score`) of using that path (i.e. to get from A to E via C), which is 22; 22 is less than infinity, so you erase that and write 22 instead, and record the previous node as C. The heuristic `h(E)` is 10. This gives you a new `f-score` for node E of 32 (22 + 10 = 32) which is recorded in the unvisited list.
    - The edge from **C to F** has a weight of 8. The `g-score` for C is 11, so you add 11 and 8 to produce the cost (`g-score`) of using that path (i.e. to get from A to F via C), which is 19. The 'g-score for F is already 19, so **no changes are made**.
- You have now evaluated all of the neighbours of the current node. Remove C from the unvisited list and add it to the visited list with its `g-score` (11), `f-score` (16) and previous node (D).

Visited list			
node	**g-score**	**f-score**	**previous**
A	0	10	None
D	5	10	A
C	11	16	D

Unvisited list			
node	**g-score**	**f-score**	**previous**
B	10	25	A
E	22	32	C
F	19	19	D

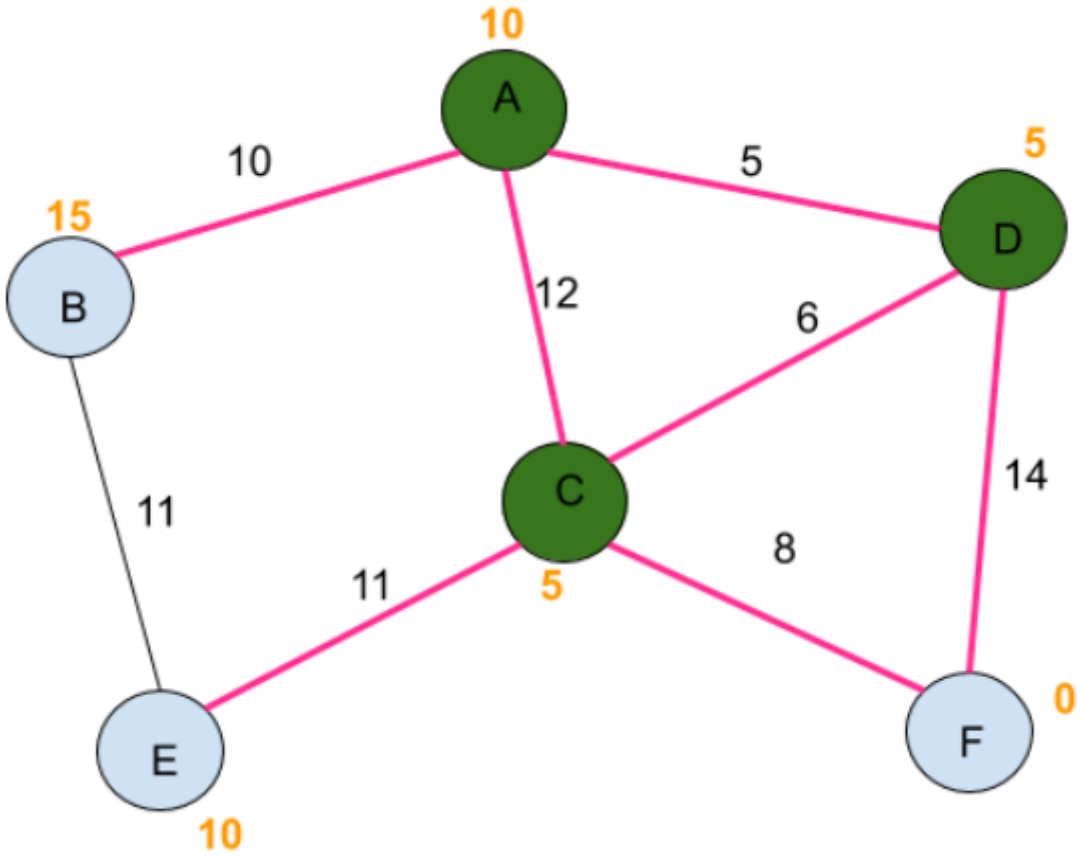


Figure 5: Node **C** has been fully evaluated

Step 5

- Pick the node that has the lowest f-score in the unvisited list. The f-score of F (19) is currently the lowest in the unvisited list. **F becomes the current node**.

- **F is the target node so the search is complete**. Remove F from the unvisited list and add it to the visited list with its `g-score` (19), `f-score` (19) and previous node (D).
- **The `g-score` for node F provides the cost of the shortest path** from A to F which is 19.

Visited list			
node	**g-score**	**f-score**	**previous**
A	0	10	None
D	5	10	A
C	11	16	D
F	19	19	D

Finding the shortest path between two nodes

In the visited list you have a set of data for all of the nodes that you examined. Observe that, unlike Dijkstra's algorithm, the A* algorithm does not necessarily produce the information for all of the nodes in the graph as it uses the heuristic function to direct the path more efficiently towards the target node.

To find the route of the shortest path from A to F:

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

Therefore, the shortest path from A to F is A → D → F.

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 `f-score` 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 represent a null value.

A* algorithm in pseudocode

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

- the graph
- visited list
- unvisited list

Dictionaries are a good choice here, as the label 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 g-score, f-score and the value of the previous node. So, if `unvisited` looks like this:

```
{'B': [10, 25,'A'], 'C': [11, 16,'D'], 'E': [22, 32, 'C'], F: [19, 19, D}
```

the syntax `visited[key]` will return a list containing the g-score, f-score and the previous node for the specified key value. For example `visited['E']` will return `[22, 32, 'C']`. The cost and the previous node can be accessed individually by position, and here constants are used (`G-SCORE`, `F-SCORE` and `PREVIOUS`) to help make the code self documenting. As an example, `visited['E'][F-SCORE]` will return the integer value 32 and `visited['E'][PREVIOUS]` will return the node C.

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

```
graph = {
            'A': {'B':10, 'C':12, 'D':5},
            'B': {'A':10, 'E':11},
            'C': {'A':12, 'D':6, 'E':11},
            'D': {'A':5, 'C':6, 'F':14},
            'E': {'B':11, 'C':11},
            'F': {'C':8, 'D':14}
          }
```

In the pseudocode that follows:

- if `current_node` is `A`, the statement `graph[current_node]`, will return `{'B':10, 'C':12, 'D':5}`.
- if the value of `neighbour` is `B`, the statement `graph[current_node][neighbour]` will return the integer value 10.

The pseudocode calls two functions where the implementation details are abstracted:

- `heuristic(n)` returns the estimated cost of the path from node n to the target node
- `get_minimum(unvisited)` returns the node with the lowest f-score in the unvisited list

```
G_SCORE = 0  // Used to index g-score
F_SCORE = 1  // Used to index f-score
PREVIOUS = 2  // Used to index previous

FUNCTION a_star(graph, start_node, target_node)

    visited = {}  // Declare visited list as empty dictionary
    unvisited = {}  // Declare unvisited list as empty dictionary

    // Add every node to the unvisited list
    FOREACH key IN graph
        unvisited[key] = [∞ , ∞ , Null]
    NEXT key

    // Update values for start node in unvisited list
    h_score = heuristic(start_node)
    unvisited[start_node] = [0, h_score, NULL]

    // Repeat until there are no nodes in the unvisited list
    finished = False
    WHILE finished == False
        IF LEN(unvisited) == 0 THEN  // No nodes left to evaluate
            finished = True
        ELSE
            // Get node with lowest f-score from open list
            current_node = get_minimum(unvisited)
            IF current_node == target_node THEN
                finished = True
                // Copy data to visited list
                visited[current_node] = unvisited[current_node]
            ELSE
                // Examine neighbours
                FOREACH neighbour IN graph[current_node]
                    // Only check unvisited neighbours
                    IF neighbour NOT IN visited THEN
                        // Calculate new g-score
                        new_g_score = unvisited[current_node][G_SCORE] +
graph[current_node][neighbour]
                        // Check if new g-score is less
                        IF new_g_score < unvisited[neighbour][G_SCORE] THEN
                            unvisited[neighbour][G_SCORE] = new_g_score
                            unvisited[neighbour][F_SCORE] = new_g_score +
heuristic(neighbour)
                            unvisited[neighbour][PREVIOUS] = current_node
                        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
```

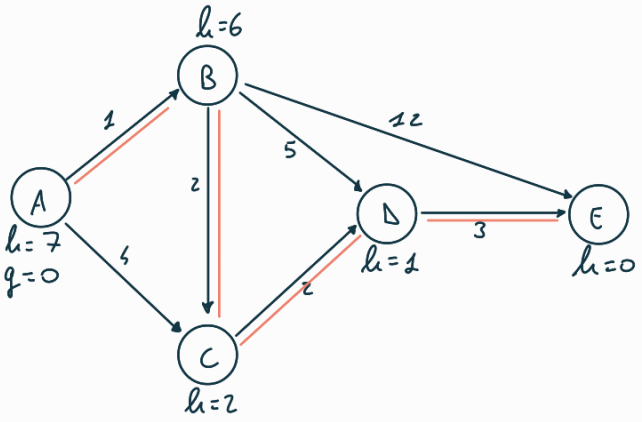
```
        ENDIF
    ENDWHILE

    // Return final visited list
    RETURN visited
ENDFUNCTION
```

Dijkstra's algorithm will always find the shortest path between the start node and a target node. In addition, Dijkstra's algorithm will find the shortest path between the start node and every other node in the graph. However, the algorithm can be inefficient as it will 'waste time' evaluating routes that could be ignored.

The A* algorithm is more efficient than Dikstra's algorithm. By using a heuristic function to provide an estimate of the cost of the path between each node and the target node, it can make better choices about the next path to take and will find the shortest path faster.

The greatest challenge in selecting A* is the need for a good heuristic function. The time it takes to provide the heuristic must not cancel out any time savings in the process of pathfinding. In addition, the heuristic must not overestimate the cost of the path. If h(n) is always lower than (or equal to) the cost of moving from n to the target node, then A* is guaranteed to find the shortest path.



open_list

node	g	h	f	prev.
A	0	7	7	none
B	∞	6	∞	none
C	∞	2	∞	none
D	∞	1	∞	none
E	∞	0	∞	none

closed_list

node	g	h	f	prev.

---

open_list

node	g	h	f	prev.
B	~~∞~~ 1	6	~~∞~~ 7	~~none~~ A
C	~~∞~~ 4	2	~~∞~~ 6	~~none~~ A
D	∞	1	∞	none
E	∞	0	∞	none

closed_list

node	g	h	f	prev.
A	0	7	7	none

---

open_list

node	g	h	f	prev.
B	1	6	7	A
D	~~∞~~ 6	1	~~∞~~ 7	~~none~~ C
E	∞	0	∞	none

closed_list

node	g	h	f	prev.
A	0	7	7	none
C	4	2	6	A

---

open_list

node	g	h	f	prev.
D	6	1	7	A
E	~~∞~~ 13	0	~~∞~~ 13	~~none~~ B
C	3	2	5	B

closed_list

node	g	h	f	prev.
A	0	7	7	none
C	~~4~~ 3	2	~~6~~ 5	~~A~~ B
B	1	6	7	A

REWRITE THE LAST STEP FOR MORE CLARITY

node	g	h	f	prev.
D	6	1	7	A
E	13	0	13	B
C	3	2	5	B

node	g	h	f	prev.
A	0	7	7	none
B	1	6	7	A

---

node	g	h	f	prev.
D	~~6~~ 5	1	~~7~~ 6	~~A~~ C
E	13	0	13	B

node	g	h	f	prev.
A	0	7	7	none
B	1	6	7	A
C	3	2	5	B

---

node	g	h	f	prev.
E	~~13~~ 5+3=8	0	~~13~~ 8+0=8	~~B~~ D

node	g	h	f	prev.
A	0	7	7	none
B	1	6	7	A
C	3	2	5	B
D	5	1	6	C

---

node	g	h	f	prev.

node	g	h	f	prev.
A	0	7	7	none
B	1	6	7	A
C	3	2	5	B
D	5	1	6	C
E	8	0	8	D

Path:

A $\xrightarrow{1}$ B $\xrightarrow{2}$ C $\xrightarrow{2}$ D $\xrightarrow{3}$ E

cost = 8