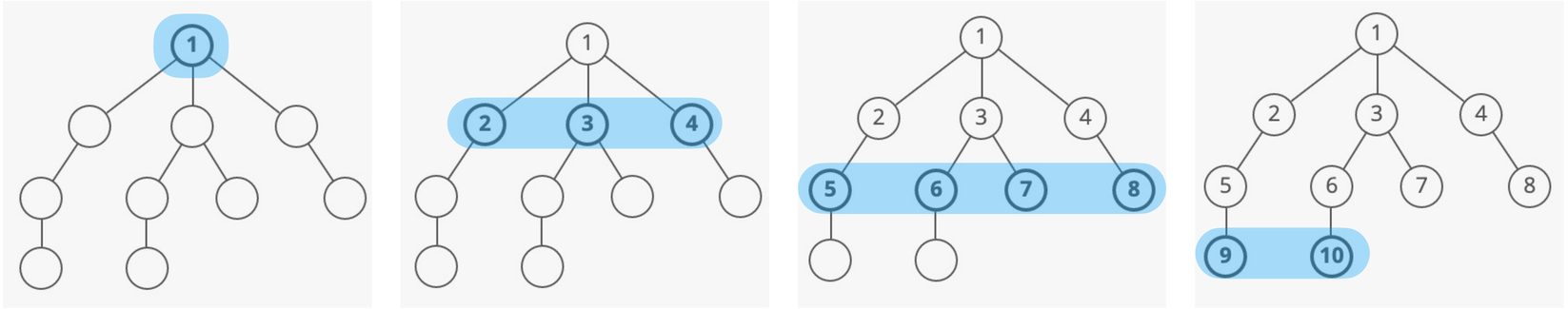
Algorithms and Data Structures

Graph

Part II: Graph Algorithms

Breadth-First Search

- The **Breadth-first Search** (BFS) is a method for exploring a tree or graph. In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc.
  - it is like throwing a stone in the center of a pond. The nodes you explore "ripple out" from the starting point.



Breadth-First Search

- Pseudocode and Example
- BFS(graph, start)
  - Let q be a queue.
  - Let v be a list with every node and mark them unvisited.
  - Put start in q
  - While (q is not empty)
    - Pop first element in q, let it be current, print it.
    - For all neighbors (one connection) w of current in Graph g,
      - If w is not visited
        - Put w in q and mark it as visited

Breadth-First Search Pseudocode

BFS(*graph*, *start*)

Let *q* be a queue.

Let *v* be a list with every node and mark them unvisited.

Put *start* in *q*

While (*q* is not empty)

Pop first element in *q* as *current*

For all neighbors (one connection) *w* of *current* in *graph*,

If *w* is not visited

Put *w* in *q* and mark it as visited

Breadth-First Search

BFS(*graph*, *start*)

Let *q* be a queue.

Let *v* be a list with every node and mark them unvisited.

Put *start* in *q*

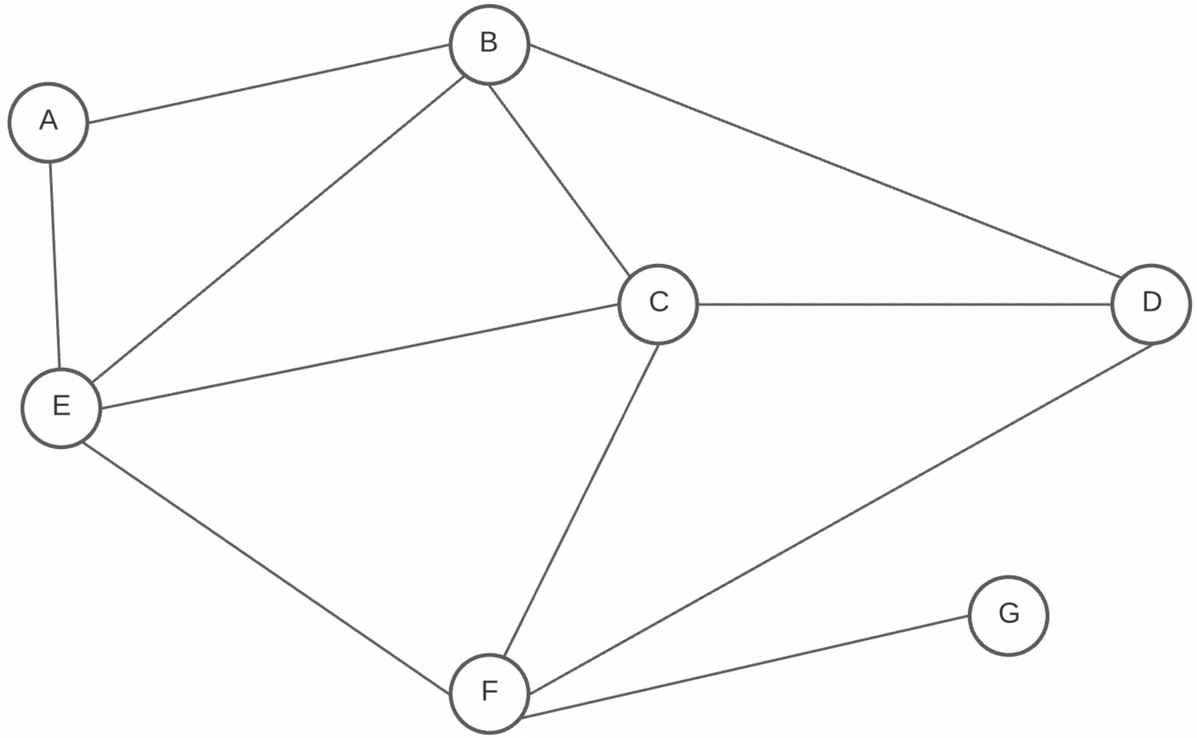
While (*q* is not empty)

Pop first element in *q* as *current*

For all neighbors (one connection) *w* of *current* in *graph*,

If *w* is not visited

Put *w* in *q* and mark it as visited



Breadth-First Search

BFS(*graph*, *start*)

Let *q* be a queue.

Let *v* be a list with every node and mark them unvisited.

Put *start* in *q*

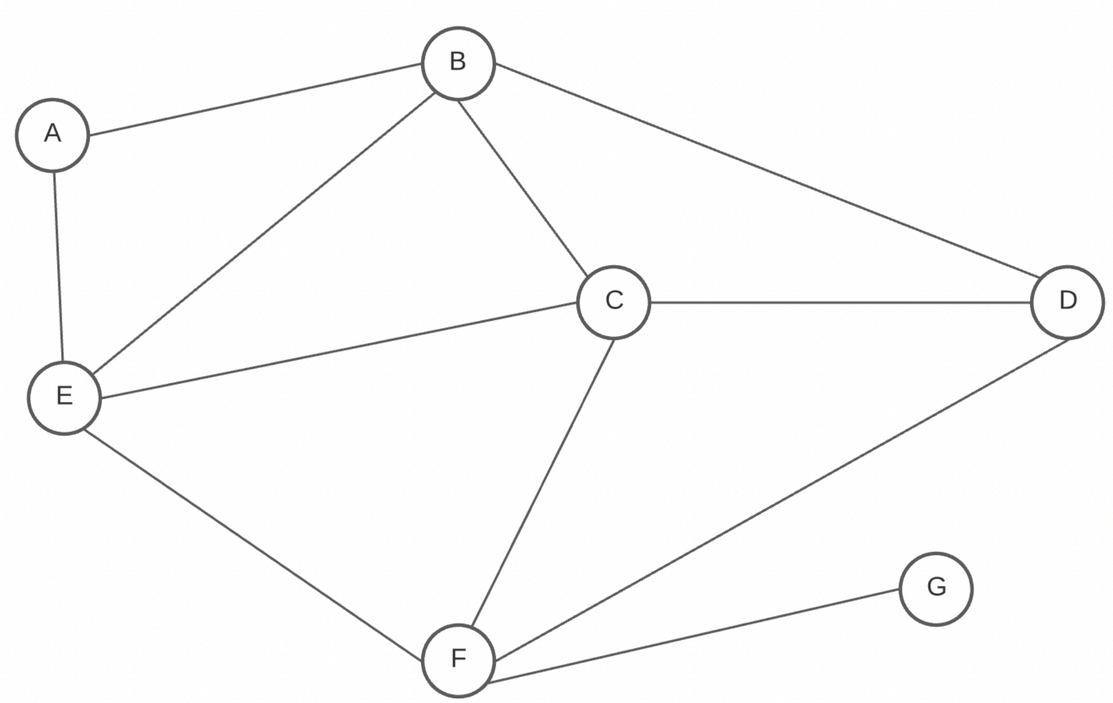
While (*q* is not empty)

Pop first element in *q* as *current*

For all neighbors (one connection) *w* of *current* in *graph*,

If *w* is not visited

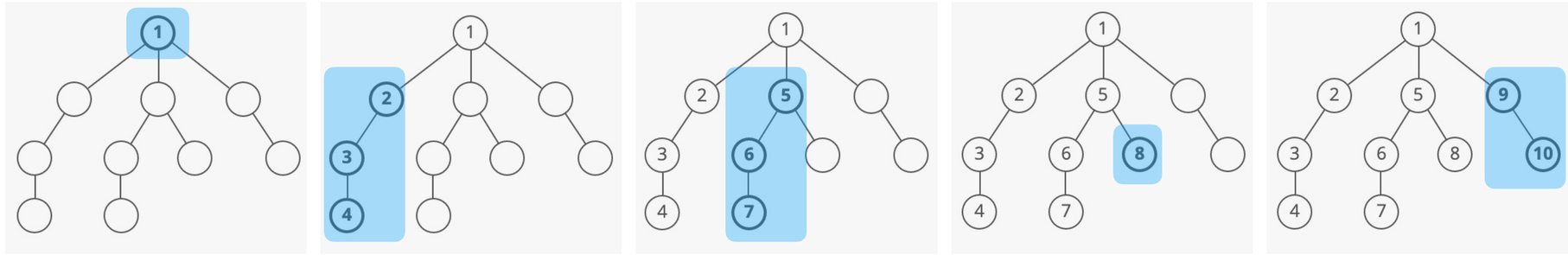
Put *w* in *q* and mark it as visited



q: ~~A B E C D F G~~
v: ~~A B C D E F G~~
output: A B E C D F G

Depth-First Search

- The **Depth-first search** (DFS) is a method for exploring a tree or graph. In a DFS, you go as deep as possible down one path before backing up and trying a different one.
  - it is like walking through a maze. You explore one path, hit a dead end, and go back and try a different one.



Depth-First Search Pseudocode

DFS(*graph*, *start*)

Let *s* be a stack.

Let *v* be a list with every node and mark them unvisited.

Put *start* in *s*

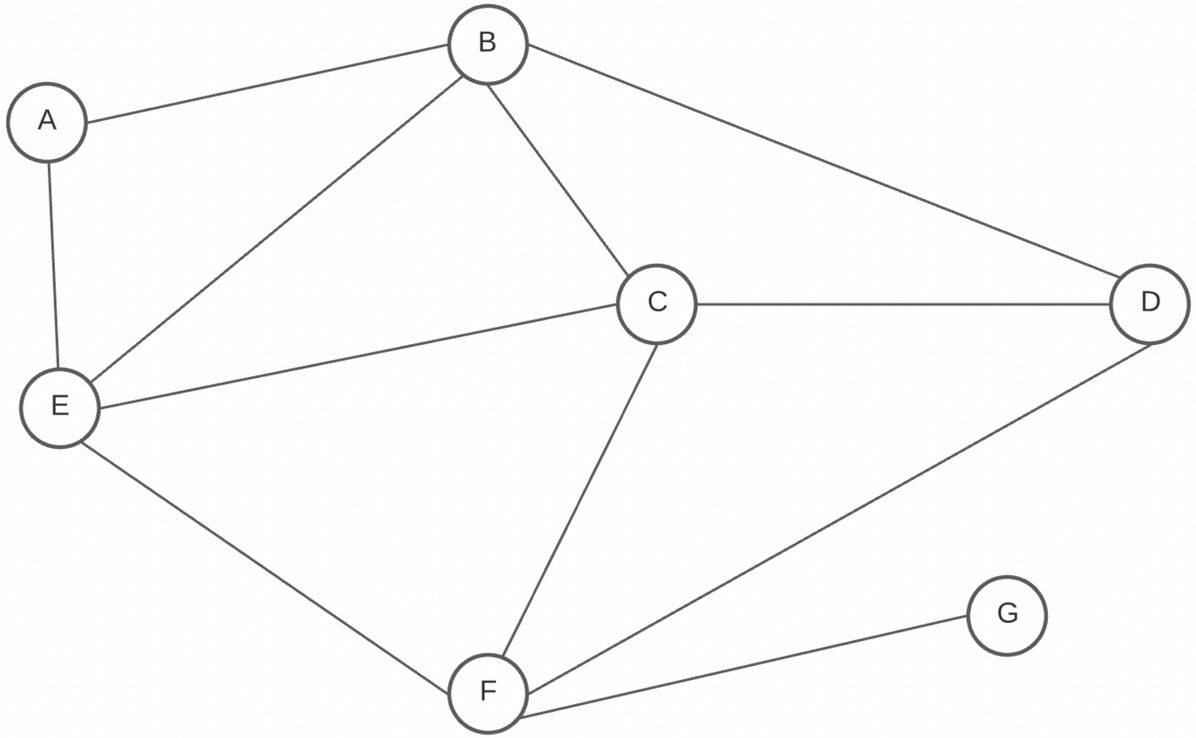
While (*s* is not empty)

Pop first element in *s* as *current*

For all neighbors (one connection) *w* of *current* in *graph*,

If *w* is not visited

Put *w* in *s* and mark it as visited



Depth-First Search Pseudocode

DFS(*graph*, *start*)

Let *s* be a stack.

Let *v* be a list with every node and mark them unvisited.

Put *start* in *s*

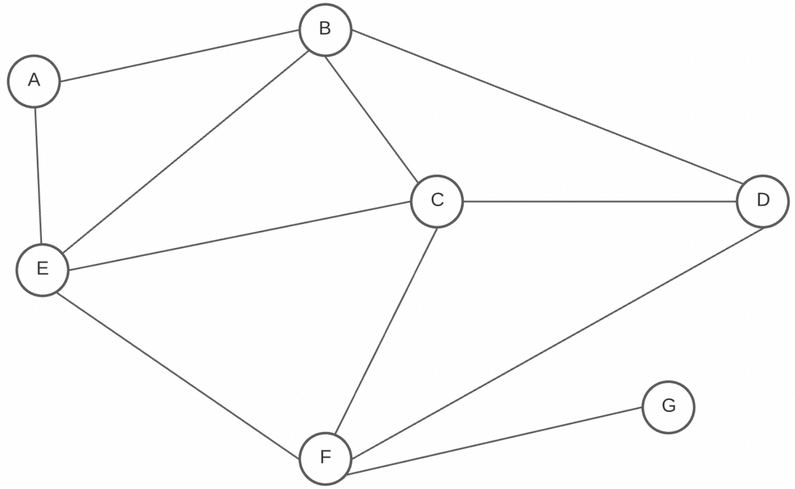
While (*s* is not empty)

Pop first element in *s* as *current*

For all neighbors (one connection) *w* of *current* in *graph*,

If *w* is not visited

Put *w* in *s* and mark it as visited



s: ~~A B E C F D G~~
v: ~~A B C D E F G~~
Output: A E F G D C B

Depth-First Search Pseudocode

DFS(*graph*, *start*)

Let *s* be a stack.

Let *v* be a list with every node and mark them unvisited.

Put *start* in *s*

While (*s* is not empty)

Pop first element in *s* as *current*

For all neighbors (one connection) *w* of *current* in *graph*,

If *w* is not visited

Put *w* in *s* and mark it as visited

BFS VS DFS

- API stands for Application Programmable Interface, and enables developers to:
  - access data (weather information, gps, etc)
  - avoid complexity (take advantage of the library)
  - extended functionality (prepare widgets etc)
  - security (gate keepers for private information)

	BFS	DFS
Advantages	A BFS will find the shortest path between the starting point and any other reachable node.	A DFS can be easily implemented with recursion.
Disadvantages	A BFS on a binary tree *generally* requires more memory than a DFS	A DFS will not necessarily find the shortest path

BFS VS DFS

BFS(*graph*, *start*)

Let *q* be a queue.

Let *v* be a list with every node and mark them unvisited.

Put *start* in *q*

While (*q* is not empty)

Pop first element in *q* as *current*

For all neighbors (one connection) *w* of *current* in *graph*,

If *w* is not visited

Put *w* in *q* and mark it as visited

DFS(*graph*, *start*)

Let *s* be a stack.

Let *v* be a list with every node and mark them unvisited.

Put *start* in *s*

While (*s* is not empty)

Pop first element in *s* as *current*

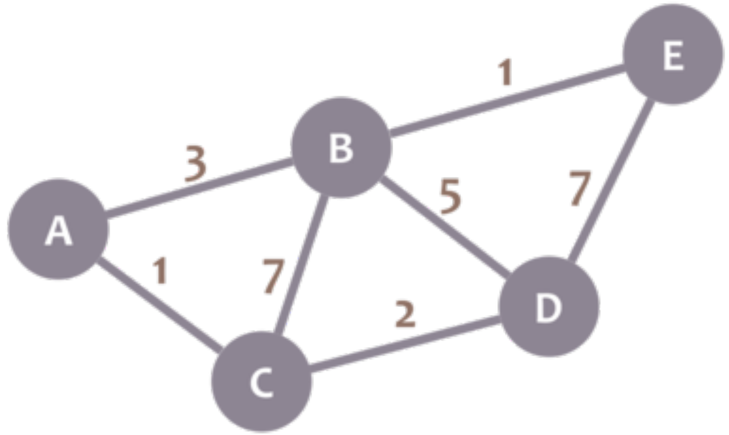
For all neighbors (one connection) *w* of *current* in *graph*,

If *w* is not visited

Put *w* in *s* and mark it as visited

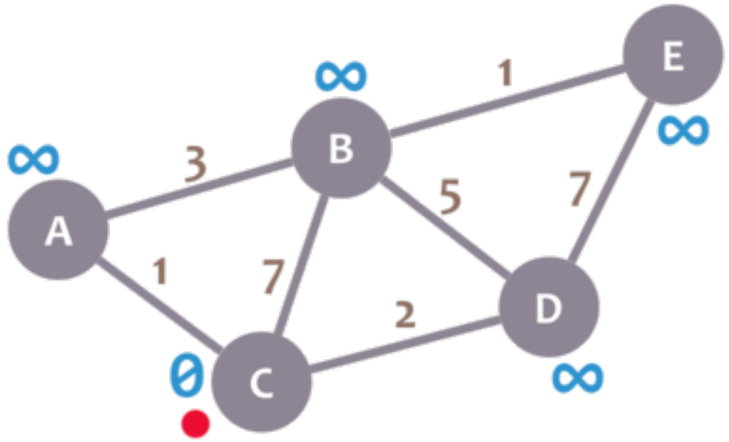
Dijkstra's Algorithm

- Dijkstra’s Algorithm finds the shortest path from one node to all other nodes in a *weighted* graph:
- Let's calculate the shortest path between node C and the other nodes in our graph:



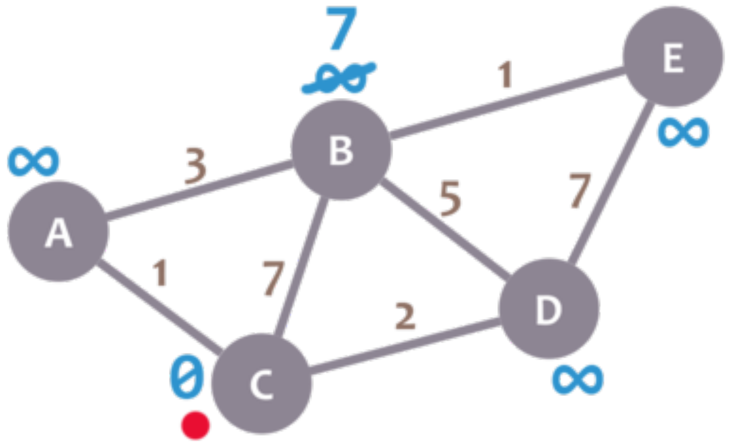
Dijkstra's Algorithm

- During the algorithm execution, we'll mark every node with its *minimum distance* to node C (our selected node). For node C, this distance is 0. For the rest of nodes, as we still don't know that minimum distance, it starts being infinity (∞):
- We'll also have a *current node*. Initially, we set it to C (our selected node). In the image, we mark the current node with a red dot



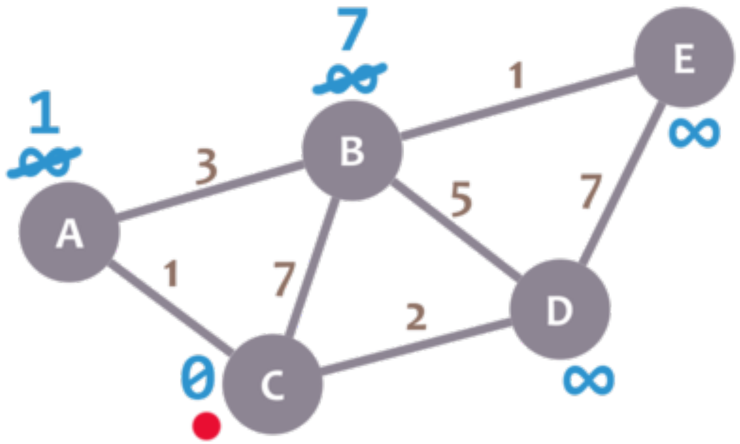
Dijkstra's Algorithm

- We check the neighbors of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7. We compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity):



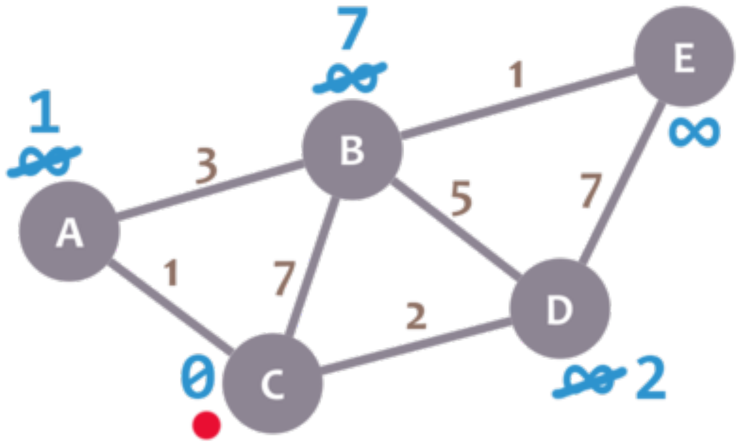
Dijkstra's Algorithm

- Let's check neighbor A. We add 0 (the minimum distance of C, our current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. We compare that 1 with the minimum distance of A (infinity), and leave the smallest value



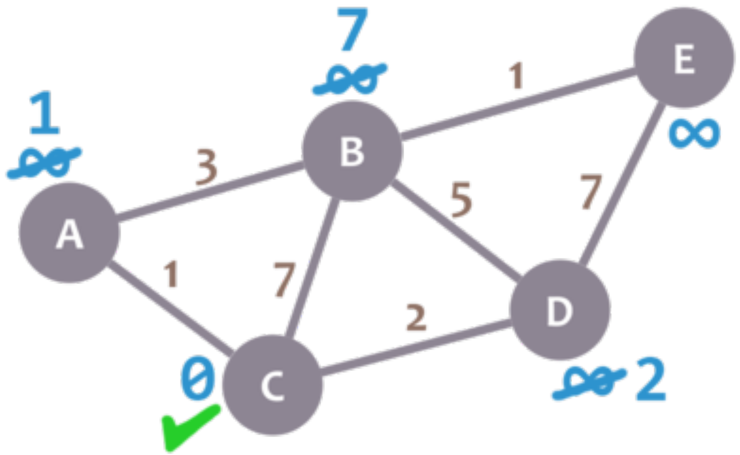
Dijkstra's Algorithm

- Repeat the same procedure for D:



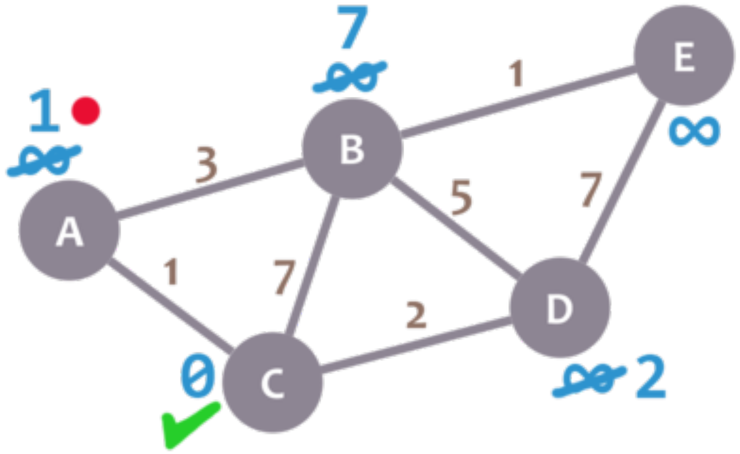
Dijkstra's Algorithm

- We have checked all the neighbors of C. Because of that, we mark it as *visited*. Let's represent visited nodes with a green check mark:



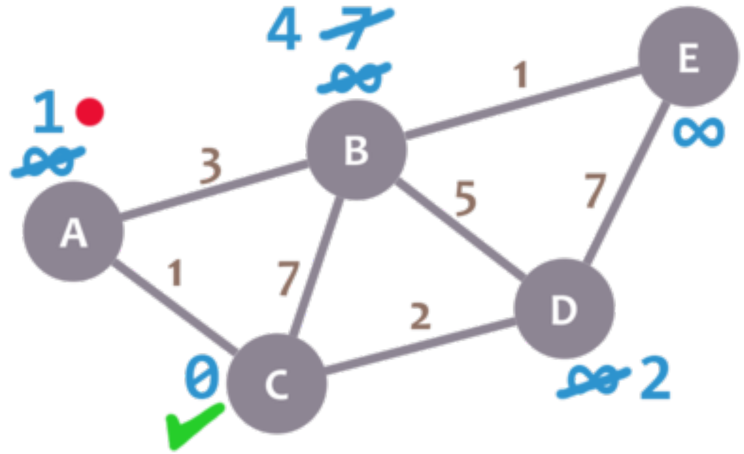
Dijkstra's Algorithm

- We now need to pick a new *current node*. That node must be the unvisited node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A. Let's mark it with the red dot:



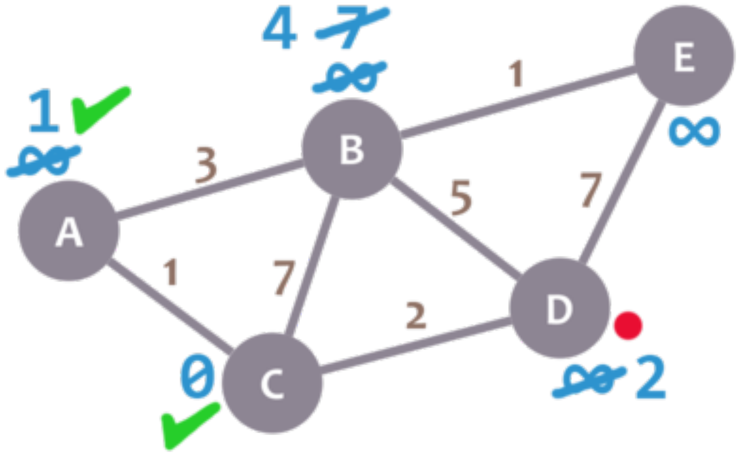
Dijkstra's Algorithm

- And now we repeat the algorithm. We check the neighbors of our current node, ignoring the visited nodes. This means we only check B.
- For B, we add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare that 4 with the minimum distance of B (7) and leave the smallest value: 4.



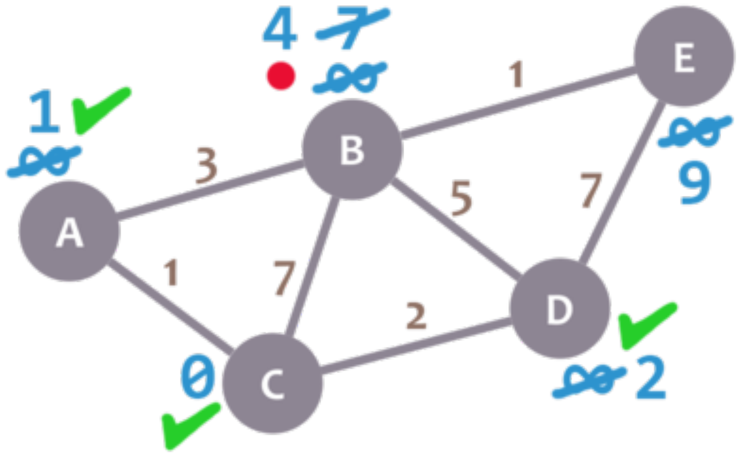
Dijkstra's Algorithm

- Afterwards, we mark A as visited and pick a new current node: D, which is the non-visited node with the smallest current distance.



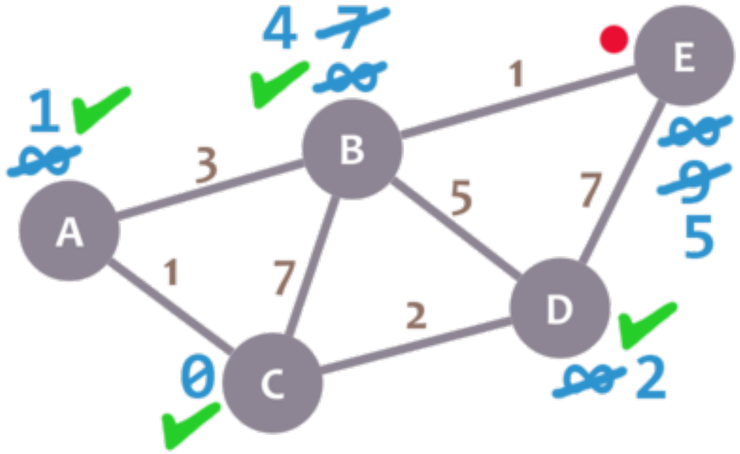
Dijkstra's Algorithm

- We repeat the algorithm. This time we check B and E.
- For B, we obtain 2 + 5 = 7. We compare that value with B's minimum distance (4) and leave the smallest value (4). For E, we obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one (9).
- We mark D as visited and set our current node to B.



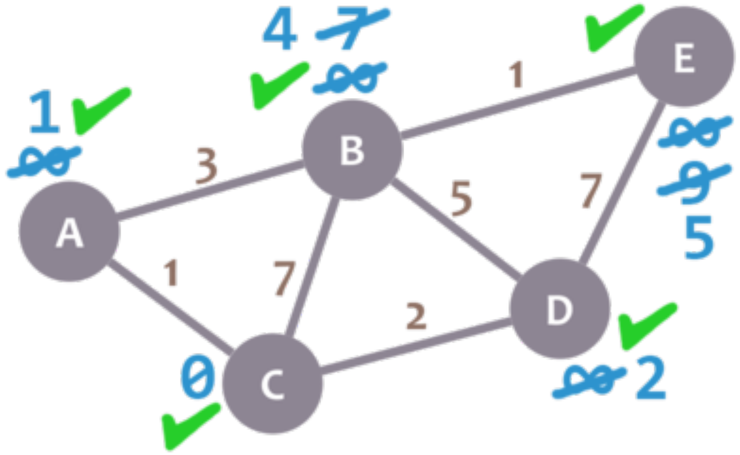
Dijkstra's Algorithm

- We only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, we mark B as visited and set E as the current node.



Dijkstra's Algorithm

- E doesn't have any non-visited neighbors, so we don't need to check anything. We mark it as visited.
- As there are not univisited nodes, we're done! The minimum distance of each node now actually represents the minimum distance from that node to node C (the node we picked as our initial node)!

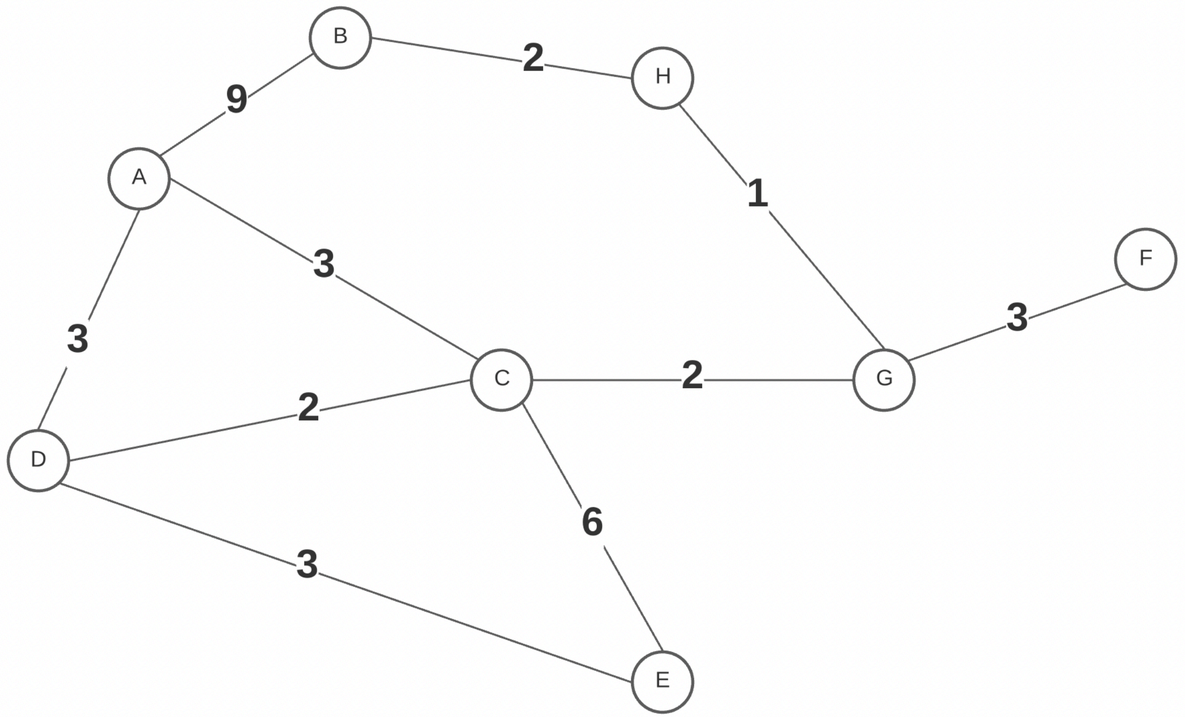


Dijkstra's Algorithm

- Here's Dijkstra's Algorithm again:
  - Mark your selected initial node with a current distance of 0 and the rest with infinity.
  - Set the non-visited node with the smallest current distance as the current node `C`.
  - For each neighbor `N` of your current node `C`: add the current distance of `C` with the weight of the edge connecting `C`-`N`. If it's smaller than the current distance of `N`, set it as the new current distance of `N`.
  - Mark the current node `C` as visited.
  - If there are non-visited nodes, go to step 2.

Exercise

- Given a weighted graph as:



- Find the shortest path starting from node A to other nodes.

What About Negative Weights?

- Dijkstra Algorithm cannot deal with edges with negative weights.
- We can use BFA(Bellman Ford Algorithm) - Single Source Shortest Path.
- We can also use WFI (Floyd Warshall Algorithms) - All Pairs Shortest Path.
- BFA/WFI algorithms have Dynamic Programming behind, we will learn it after next week.

Some Other Algorithms

- Spanning tree
  - Kruskal Algorithm (minimum spanning tree)
  - Prim Algorithm (minimum spanning tree)
- Path
  - Hamiltonian path/cycle
  - Euler path/cycle

Thank you!