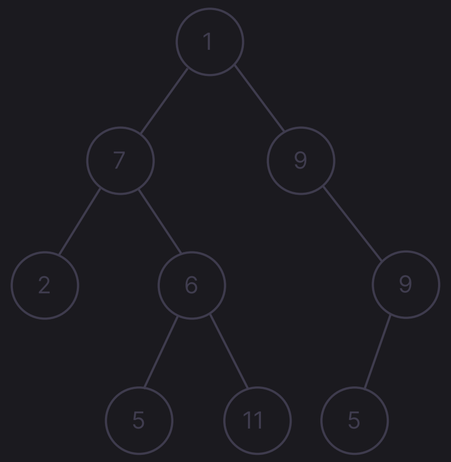
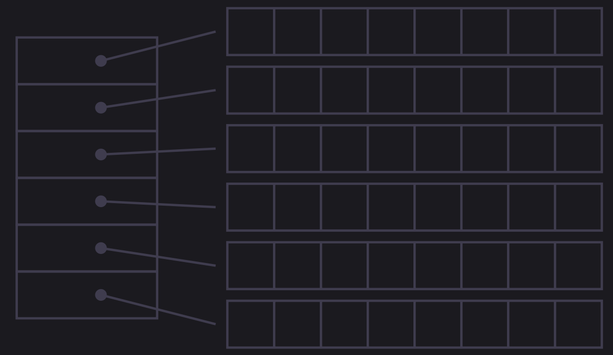
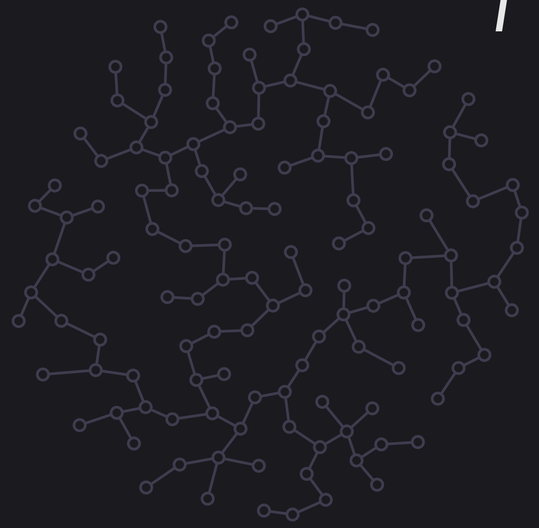


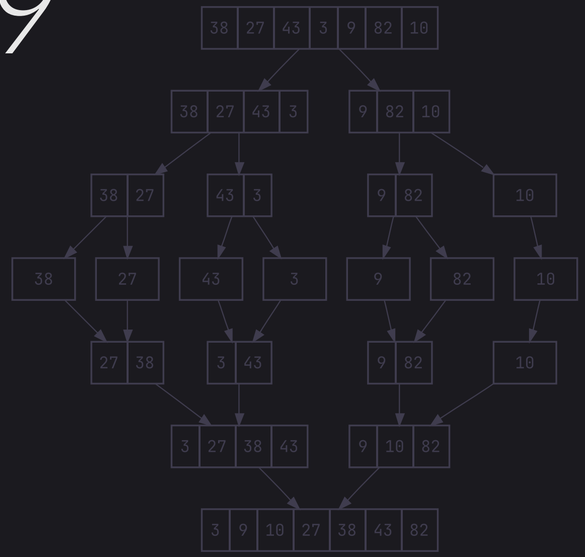


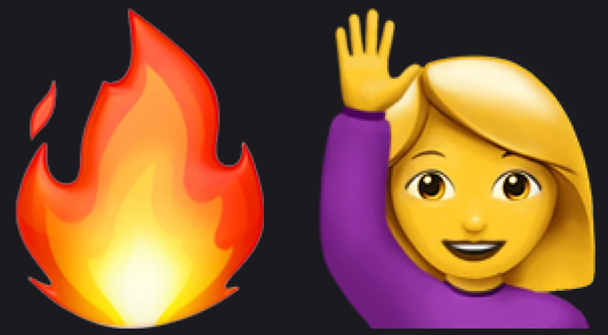
data structures & algorithms

Tutorial 8-9

(Week 9-10)







Burning questions from the previous tutorial?

This week’s lab

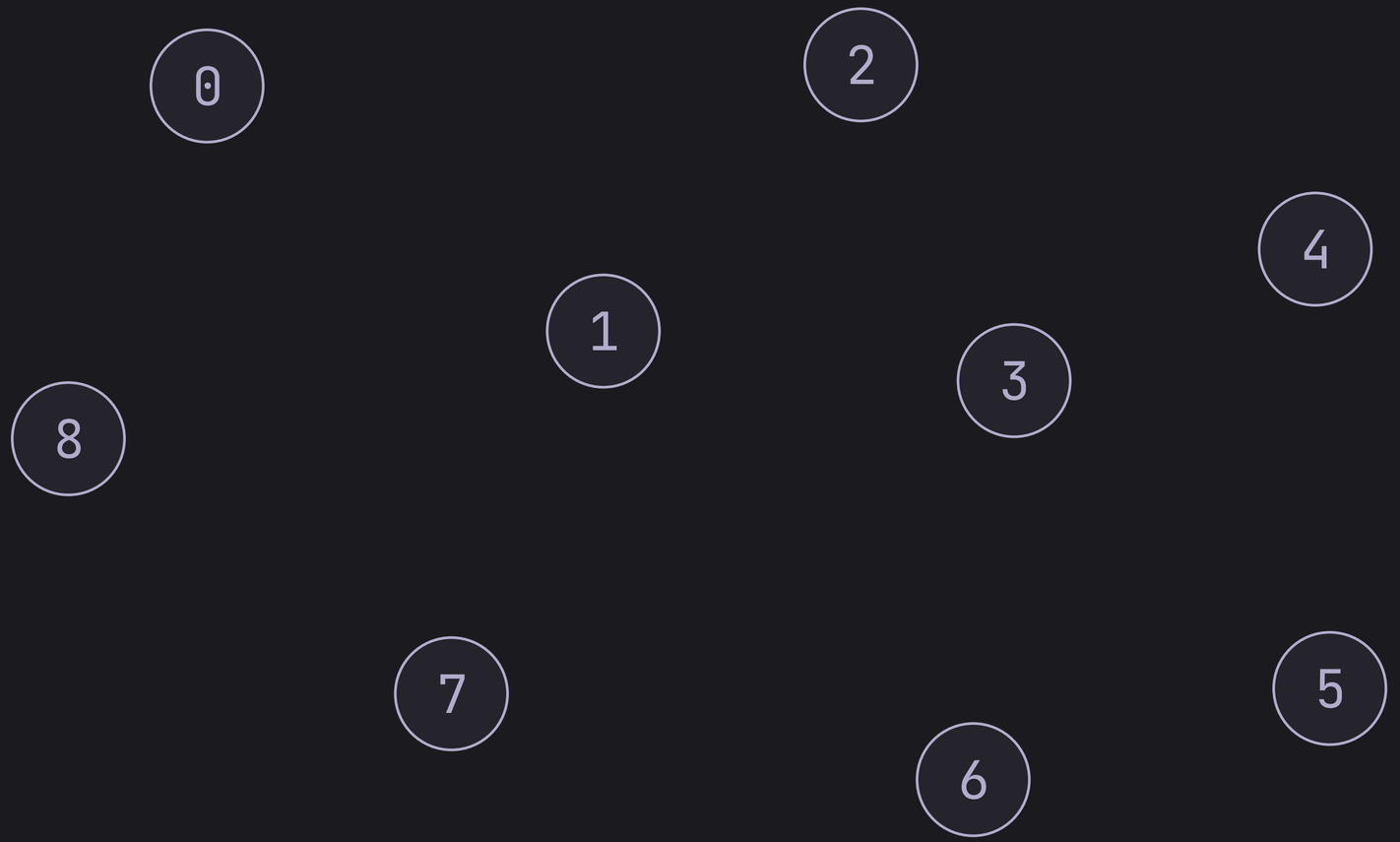
Minimum Knight Moves

This week we are learning about graphs and some very powerful graph algorithms

- Graphs
- Weighted Graphs
- Breadth First Search (BFS)
- Dijkstra’s Algorthm
- Minimum Knight Moves
- Shortest Path

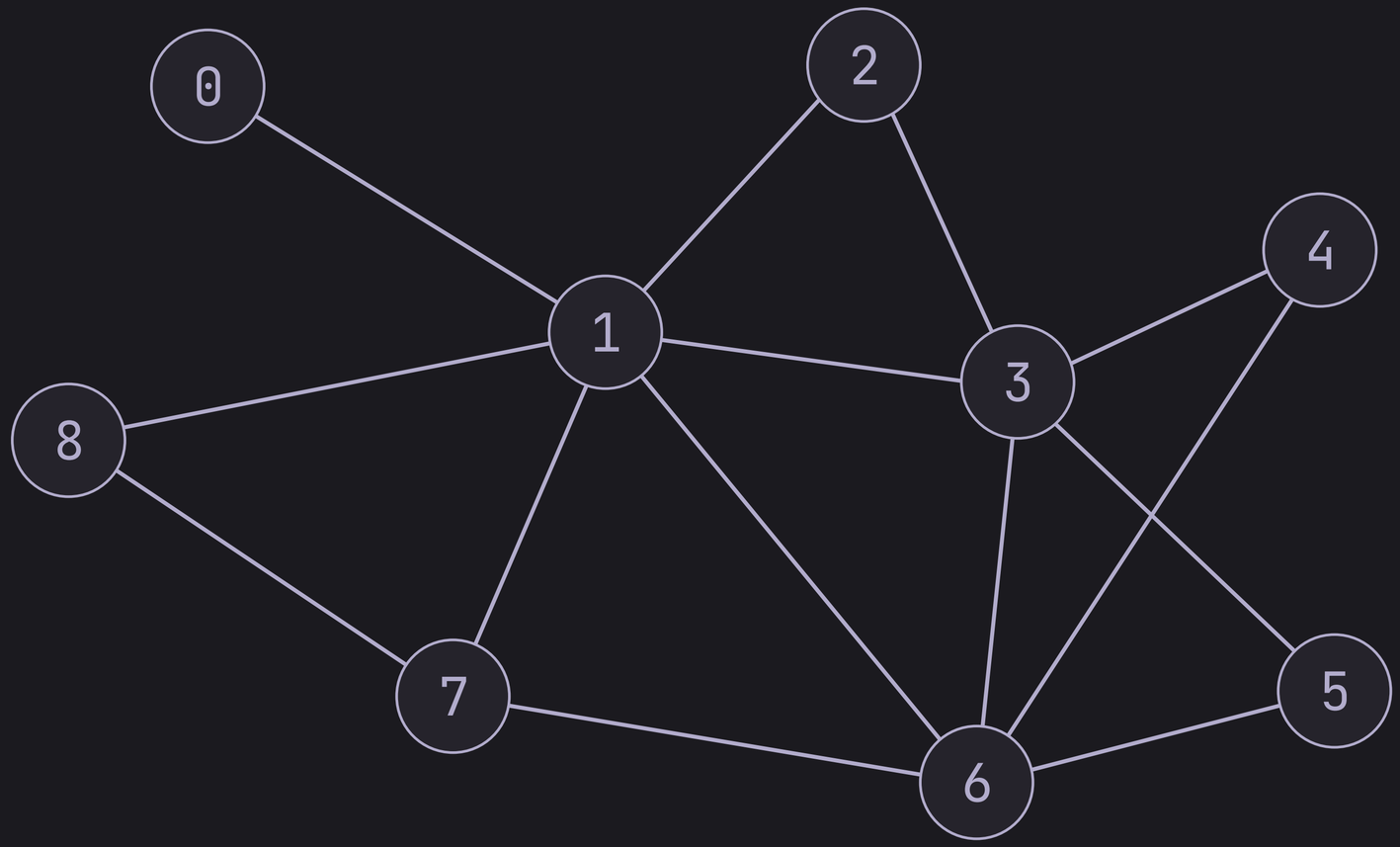
Graphs

Graphs



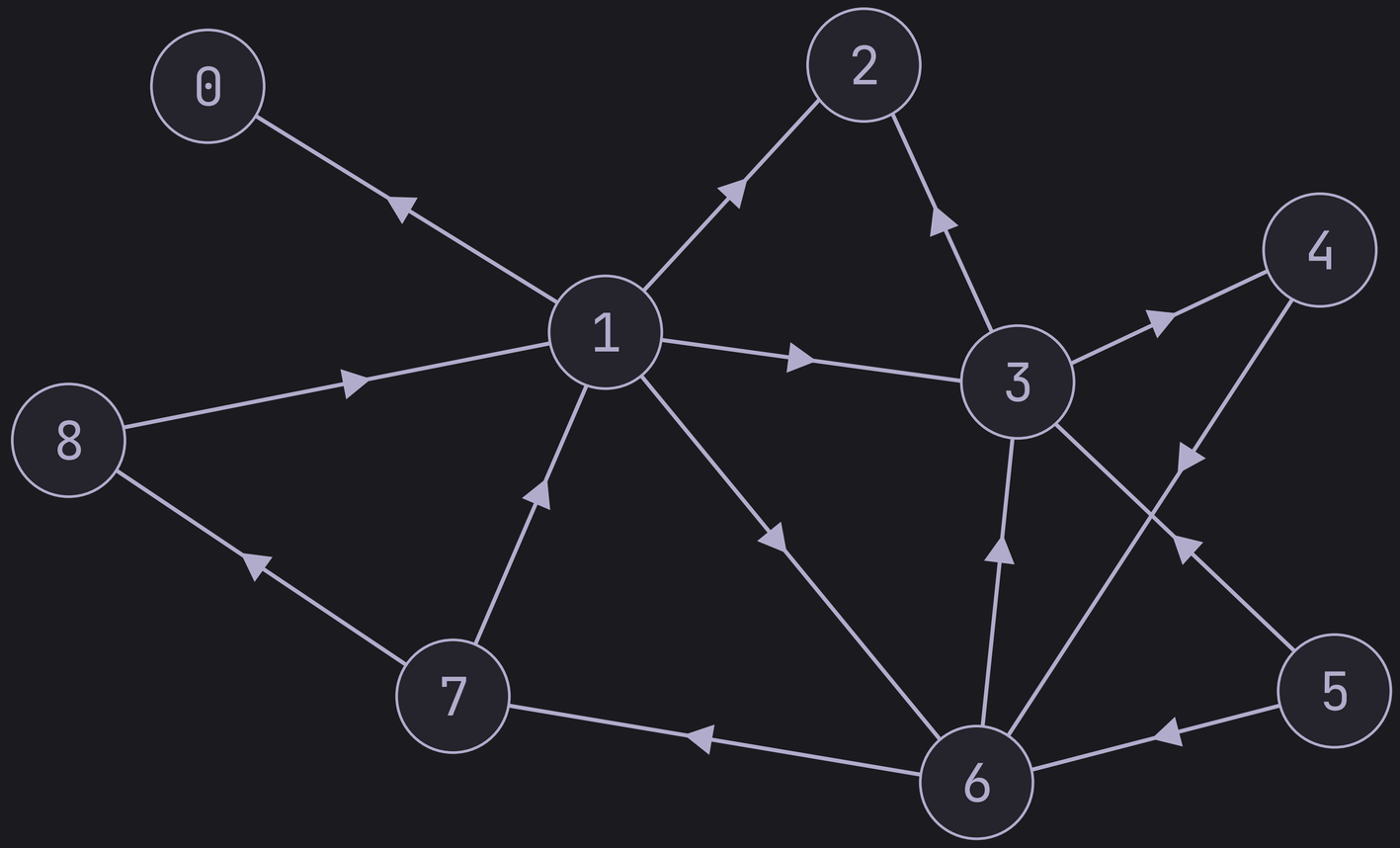
Suppose we have a collection of points that we call vertices

Graphs



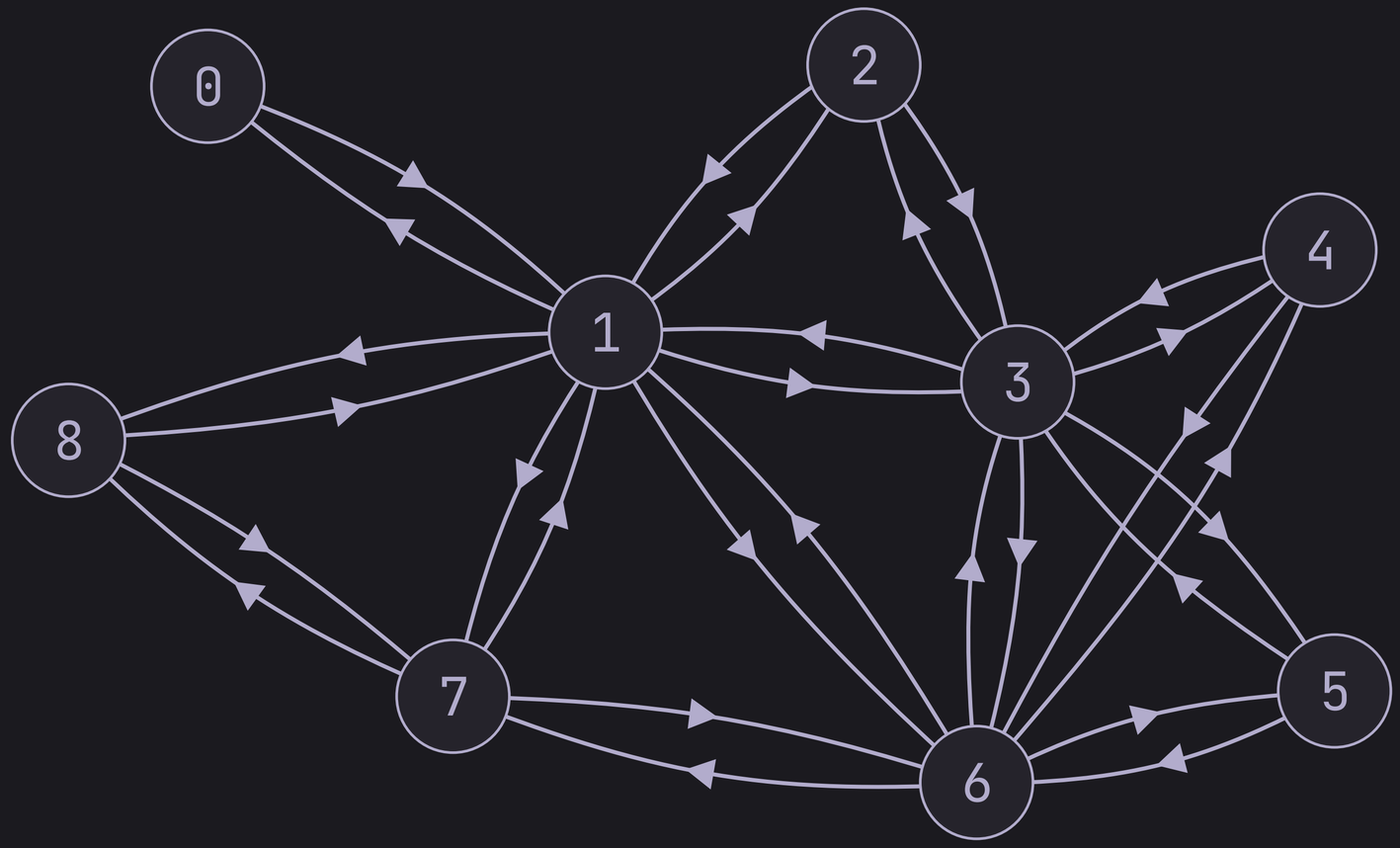
And we connect those vertices with edges

Graphs



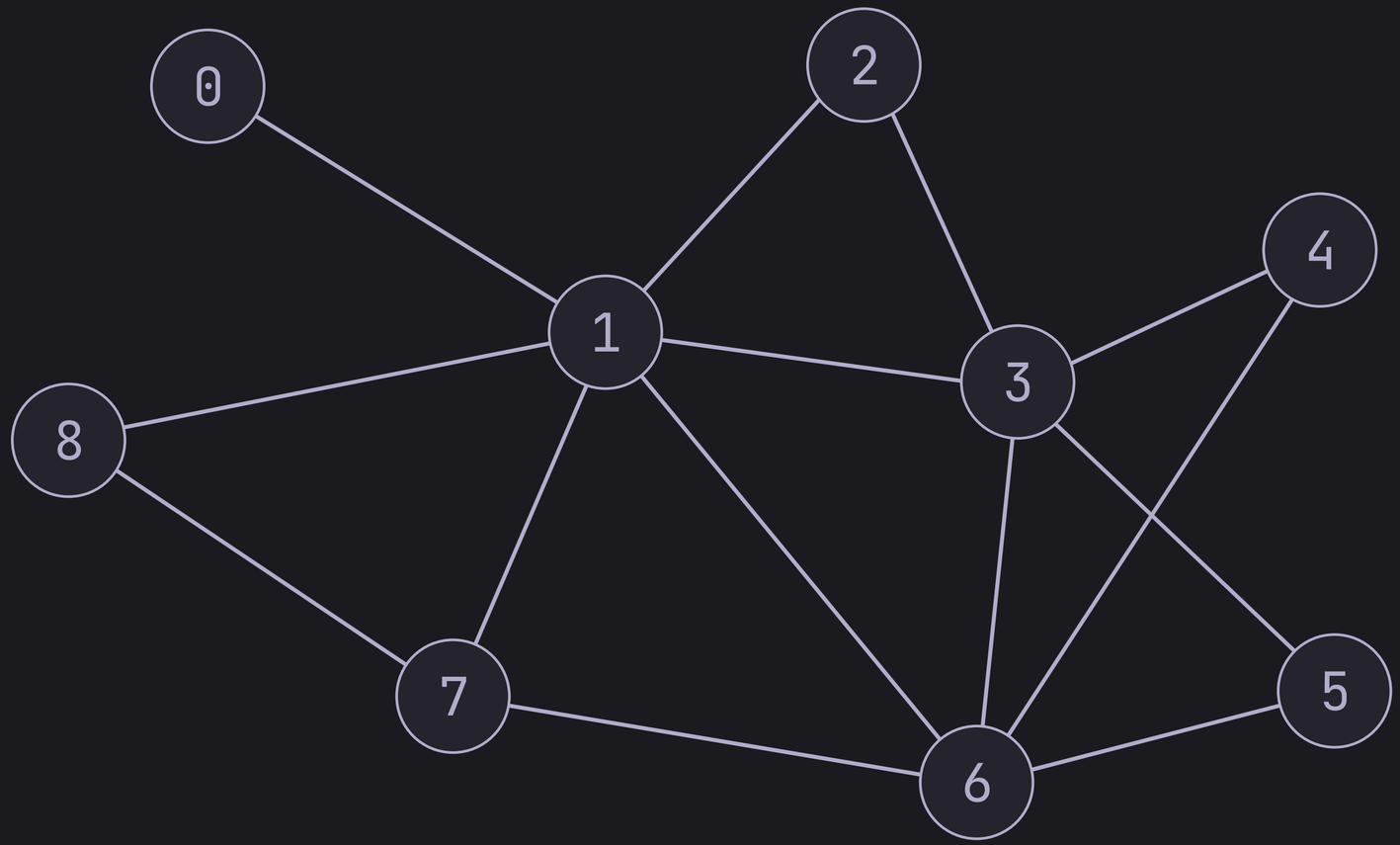
We can also have directed edges in a directed graph

Graphs

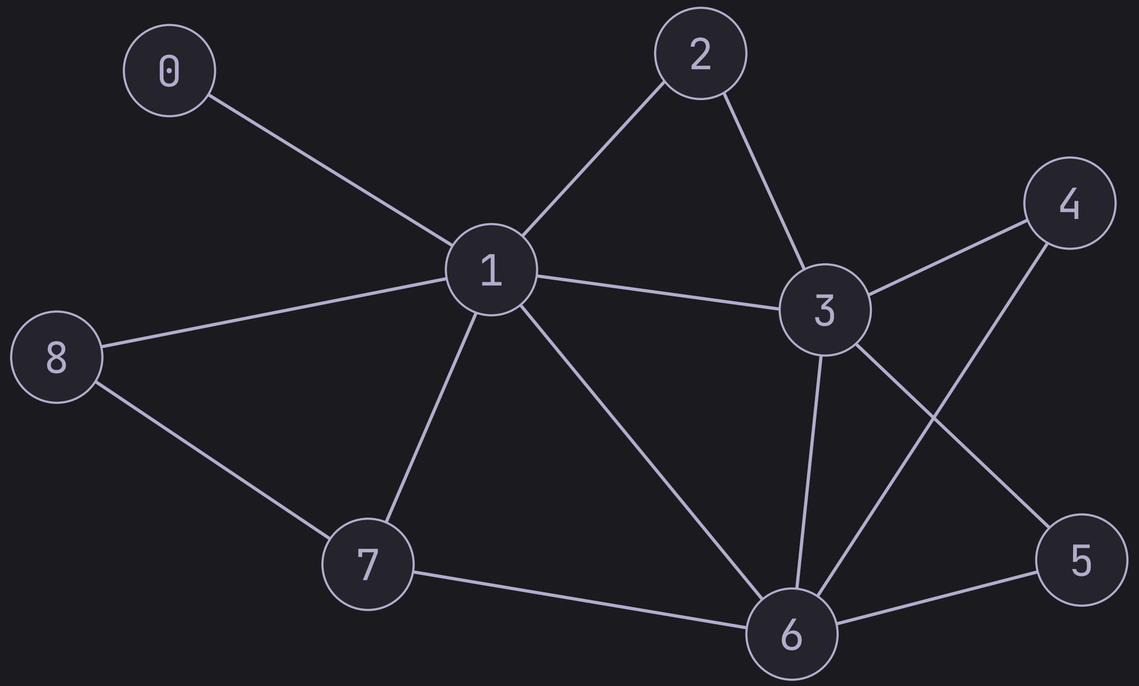


An undirected graph is really a directed graph in disguise

Graphs



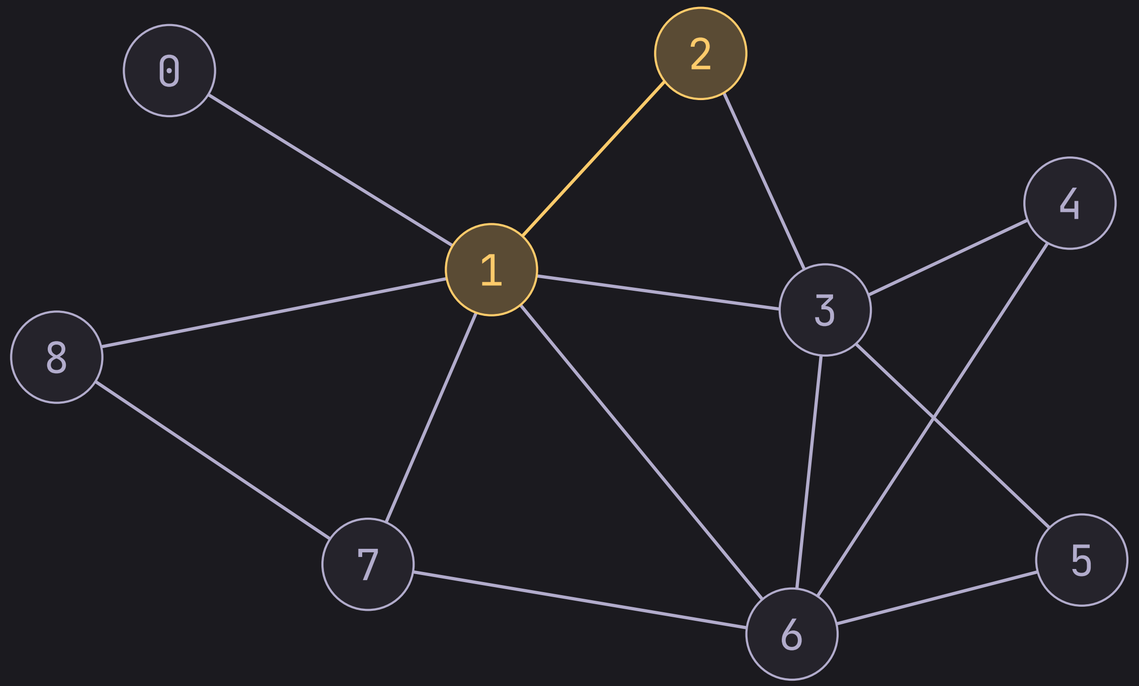
Graphs



from \ to	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

One common way to represent a graph is an adjacency matrix

Graphs

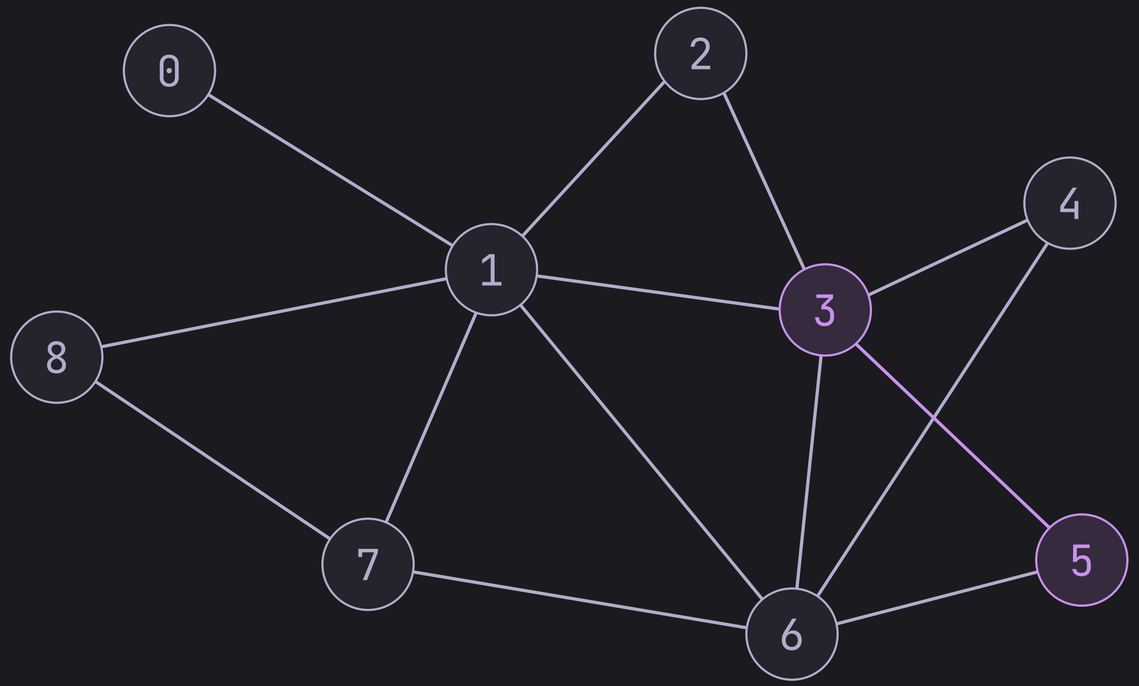


to

from	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

If two vertices share an edge, then we put a 1 in the matrix

Graphs



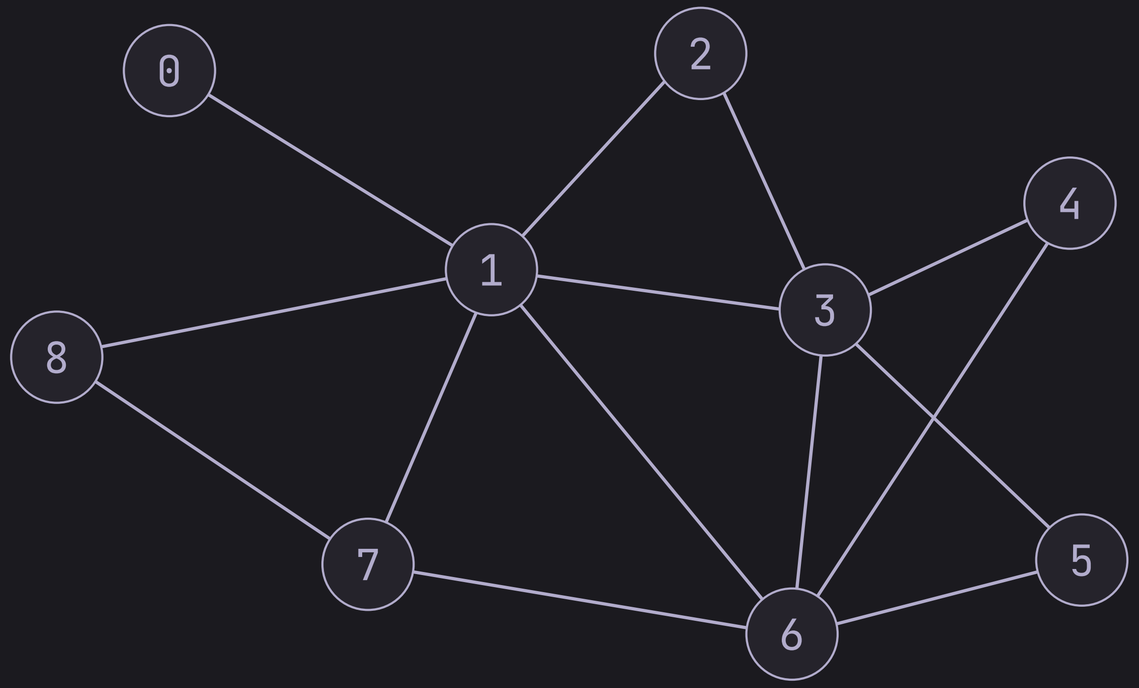
to

from

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

If two vertices share an edge, then we put a 1 in the matrix

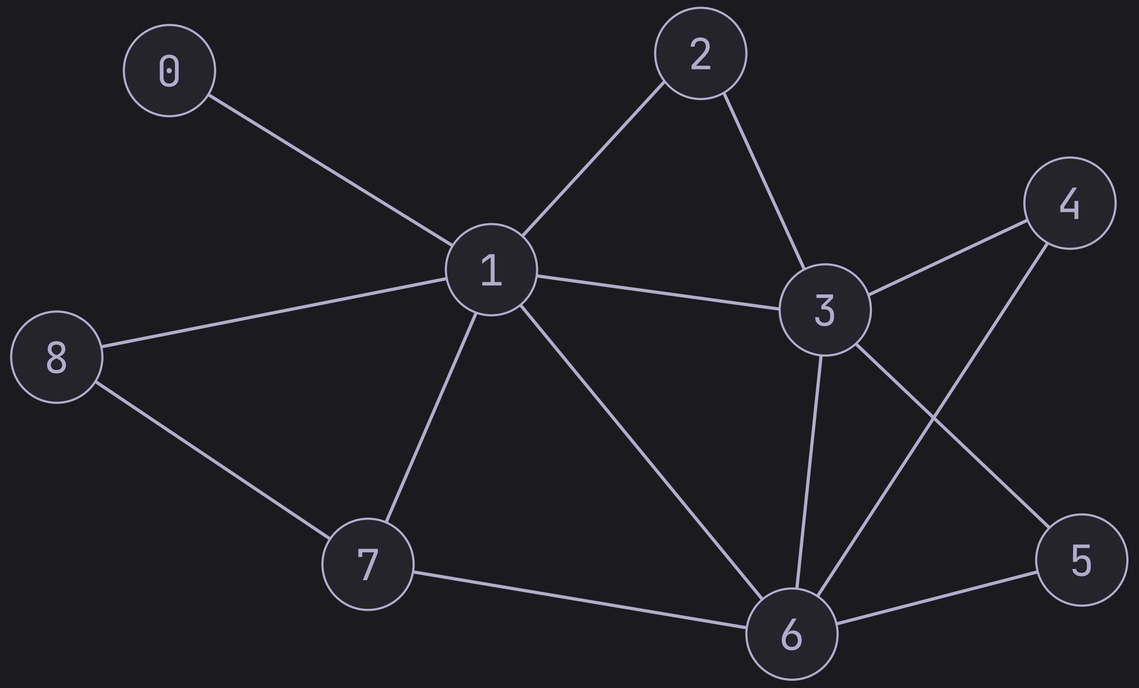
Graphs



from \ to	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	0	0	0	0
1	1	0	1	1	0	0	1	1	1
2	0	1	0	1	0	0	0	0	0
3	0	1	1	0	1	1	1	0	0
4	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	0	1	0	0
6	0	1	0	1	1	1	0	1	0
7	0	1	0	0	0	0	1	0	1
8	0	1	0	0	0	0	0	1	0

Filling this out we get the whole adjacency matrix

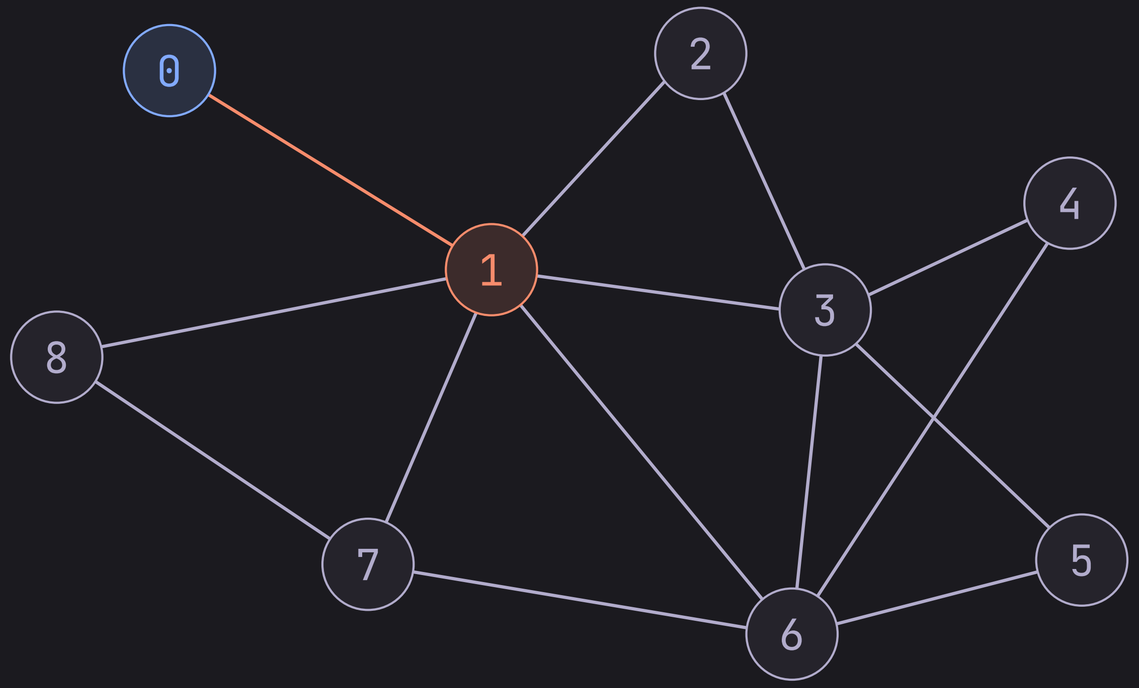
Graphs




```
0: { }
1: { }
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

Another common way to represent a graph is an adjacency list

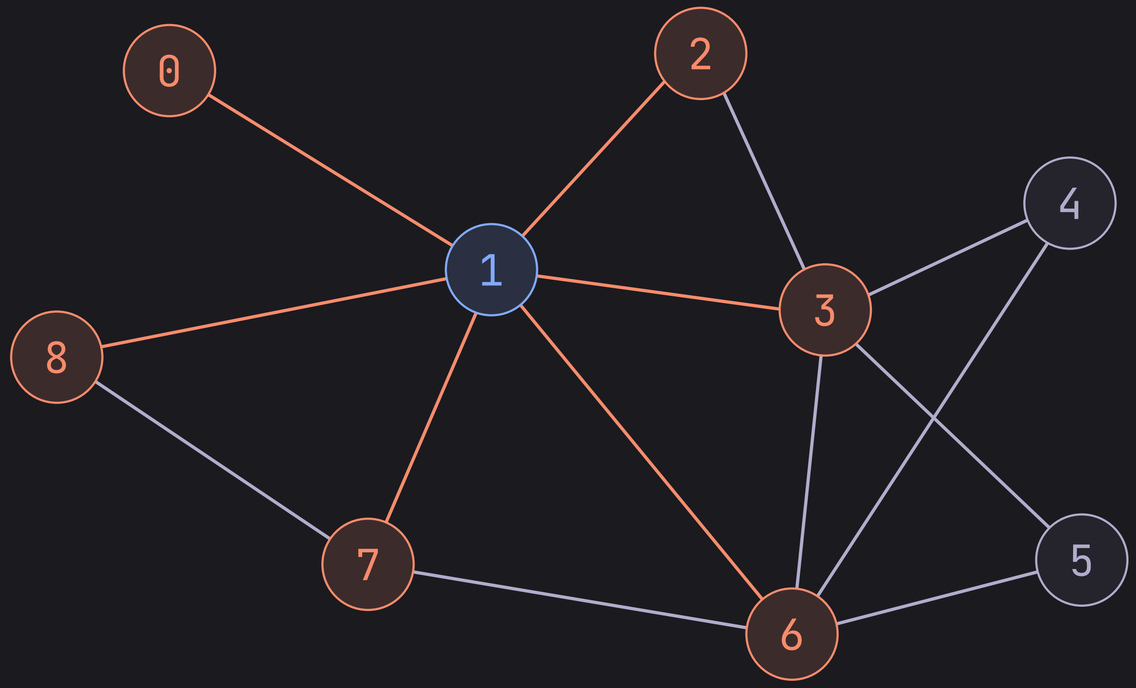
Graphs




```
0: {1}
1: { }
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

For every vertex we store a keep a list of its neighbours

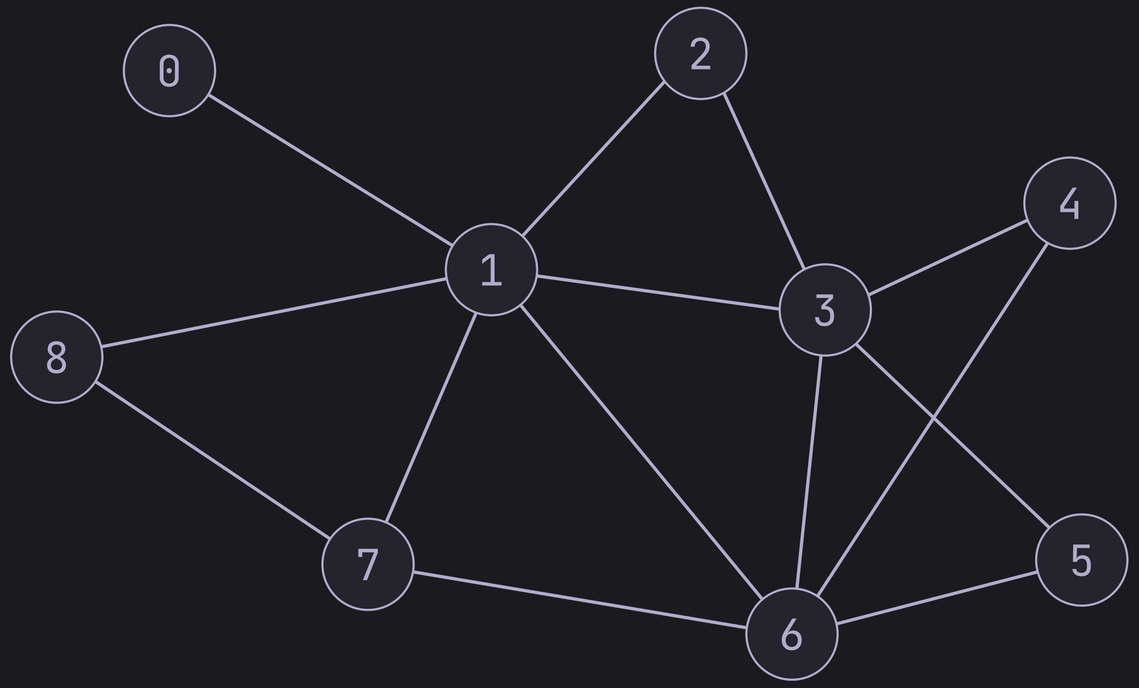
Graphs




```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

For every vertex we store a keep a list of its neighbours

Graphs




```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: {1, 3}
3: {1, 2, 4, 5, 6}
4: {3, 6}
5: {3, 6}
6: {1, 3, 5, 7}
7: {1, 6, 8}
8: {1, 7}
```

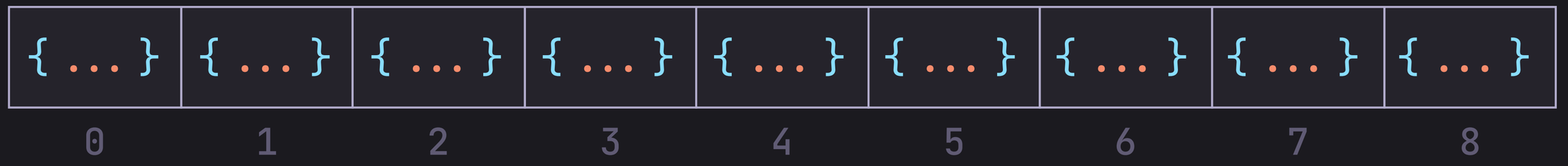
This is what the whole list looks like

Graphs

```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: {1, 3}
3: {1, 2, 4, 5, 6}
4: {3, 6}
5: {3, 6}
6: {1, 3, 5, 7}
7: {1, 6, 8}
8: {1, 7}
```

```
std::vector<std::unordered_set<int>>
```

Graphs




```
std::vector<std::unordered_set<int>>
```

An adjacency list is space efficient and easy to work with

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Inserting an edge
```

```
0: { }
1: { }
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Inserting an edge
std::set<int> neighbours1 = adjacencyList[1];
neighbours1.insert(0);
```

```
0: { }
1: {0}
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Inserting an edge

adjacencyList[1].insert(0);
```

```
0: { }
1: {0}
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Inserting an edge
adjacencyList[1].insert(0);
adjacencyList[0].insert(1);
```

```
0: {1}
1: {0}
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Checking if there is an edge

adjacencyList[3].contains(5);
```

```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: {1, 3}
3: {1, 2, 4, 5, 6}
4: {3, 6}
5: {3, 6}
6: {1, 3, 5, 7}
7: {1, 6, 8}
8: {1, 7}
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Checking if there is an edge

adjacencyList[3].contains(5);
```

```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: {1, 3}
3: {1, 2, 4, 5, 6}
4: {3, 6}
5: {3, 6}
6: {1, 3, 5, 7}
7: {1, 6, 8}
8: {1, 7}
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Checking if there is an edge

adjacencyList[3].contains(5);
```

```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: {1, 3}
3: {1, 2, 4, 5, 6}
4: {3, 6}
5: {3, 6}
6: {1, 3, 5, 7}
7: {1, 6, 8}
8: {1, 7}
```

Graph Search Algorithms

There are two really famous algorithms

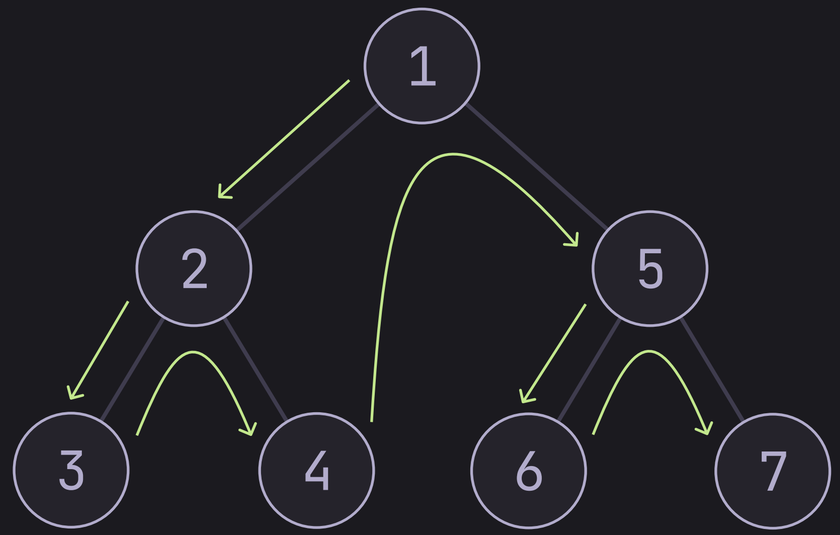
Depth First Search (DFS)

Breadth First Search (BFS)

And the good thing is they only differ very slightly in terms of coding them

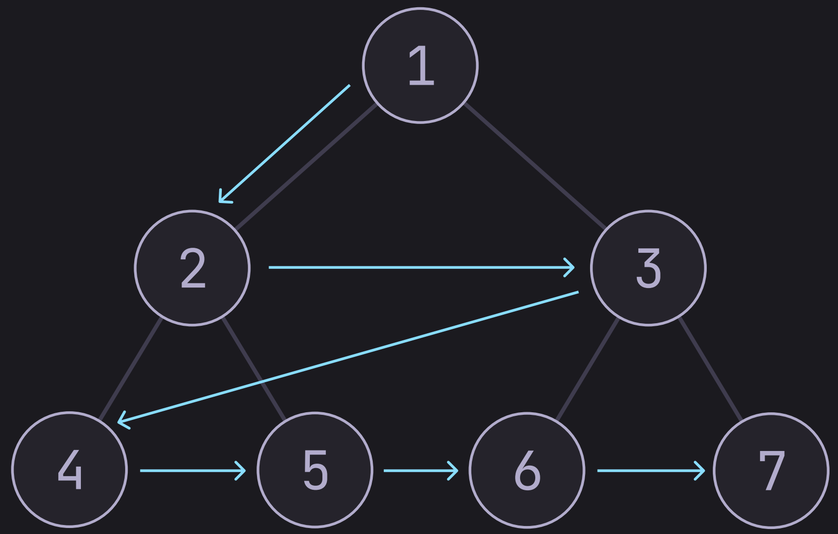
Graph Search Algorithms

Graph Search Algorithms



DFS

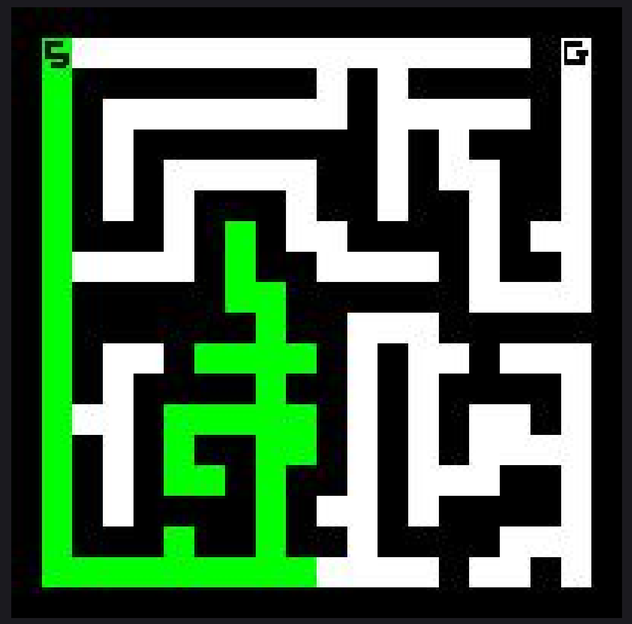
Drill down all the way down



BFS

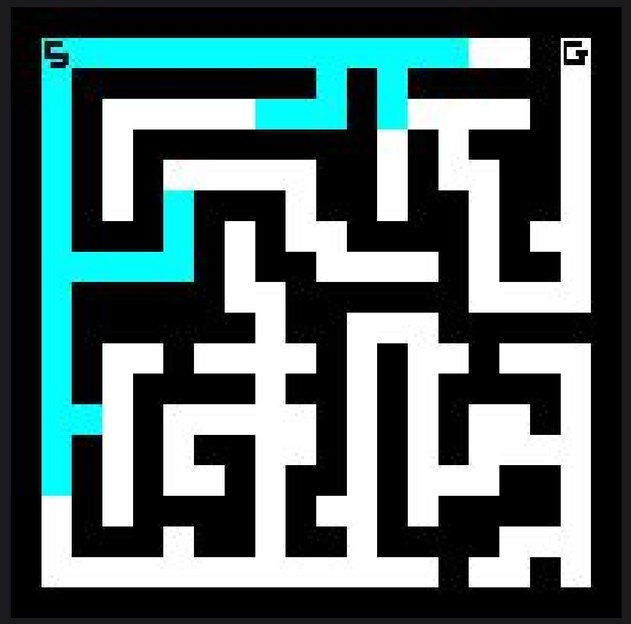
Explore Layer by layer

Graph Search Algorithms



DFS

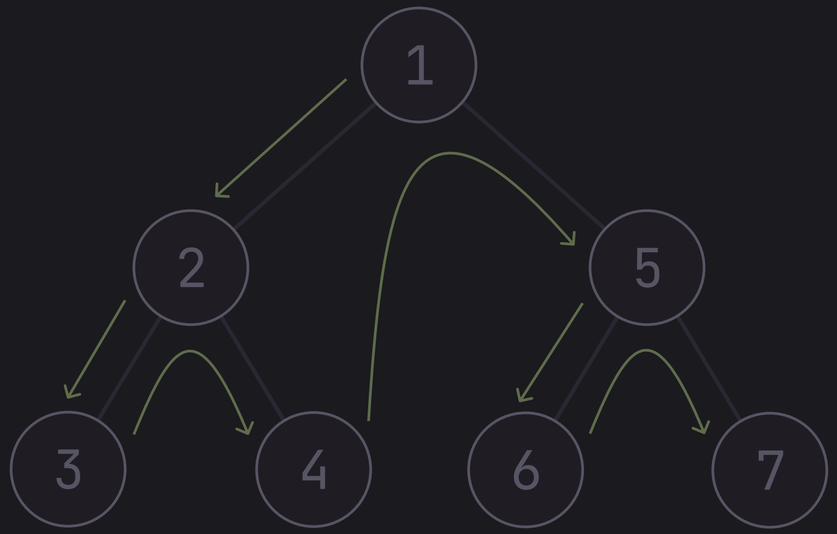
Like how a human
would search



BFS

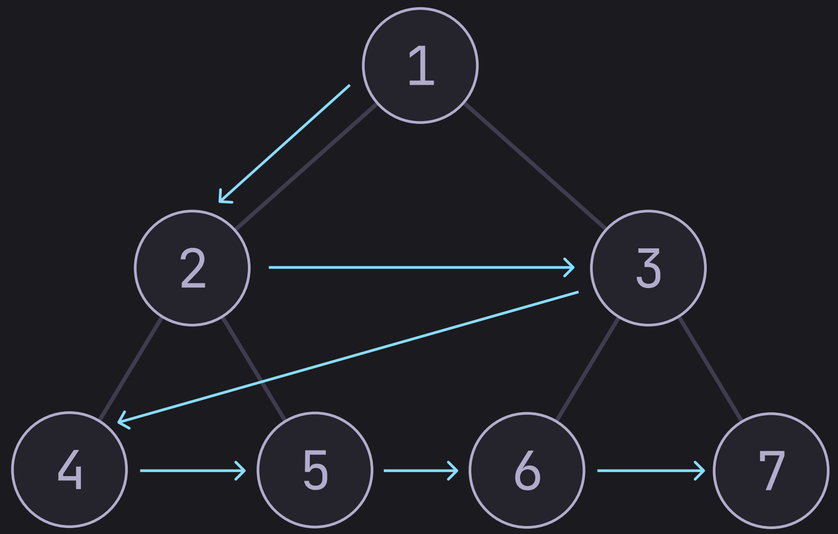
Like how a water
would flow

Graph Search Algorithms



DFS

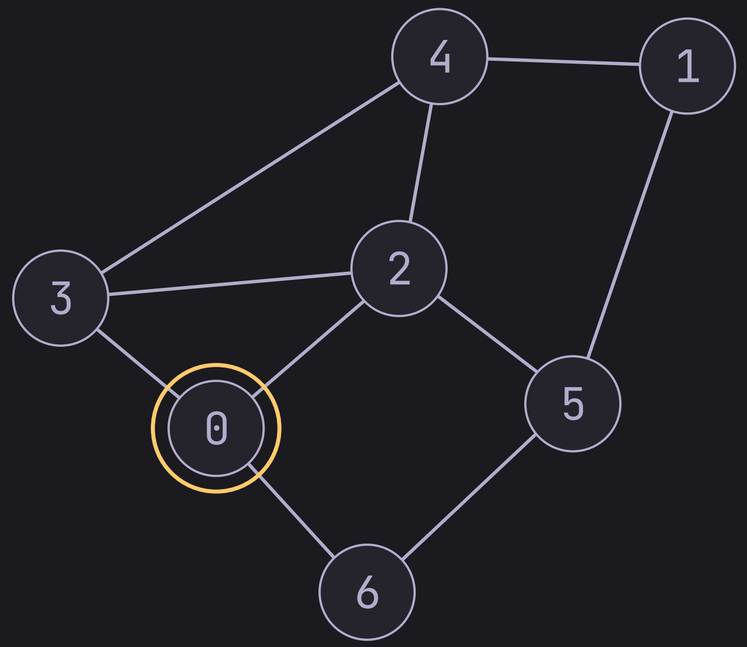
Drill down all the way down



BFS

Explore Layer by layer

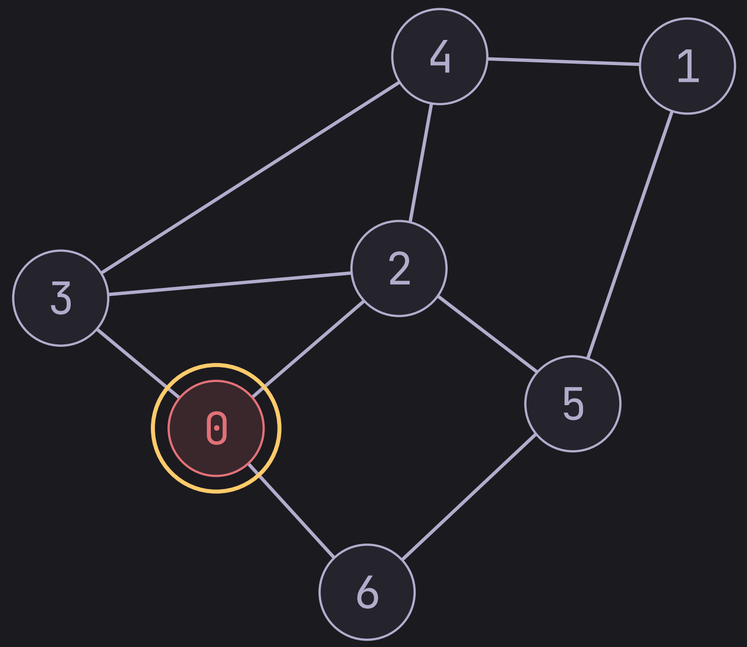
Breadth First Search



bfsQueue

```
⟶ void bfs(int start) {
    std::set<int> visited {}
    std::queue<int> bfsQueue;

    visited.insert(start);
    bfsQueue.push(start);

    while (!bfsQueue.empty()) {
      int v = bfsQueue.front();
      bfsQueue.pop();
      for (int u : adjacencyList[v]) {
        if (!visited.contains(u)) {
          visited.insert(u);
          bfsQueue.push(u);
        }
      }
    }
  }
```

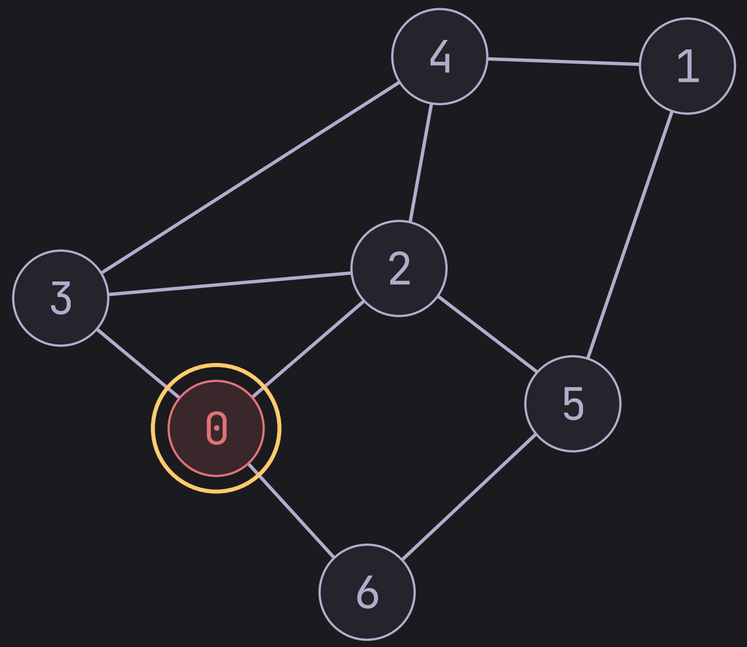
Breadth First Search

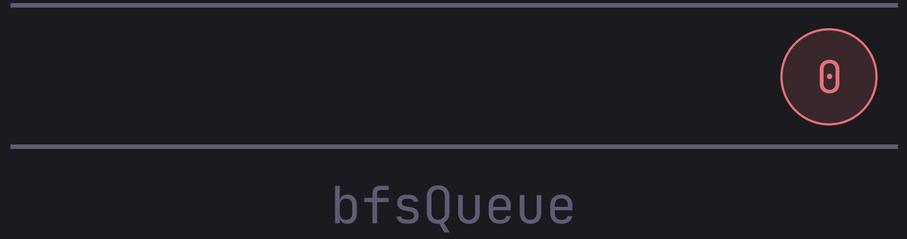


bfsQueue

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

→ visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

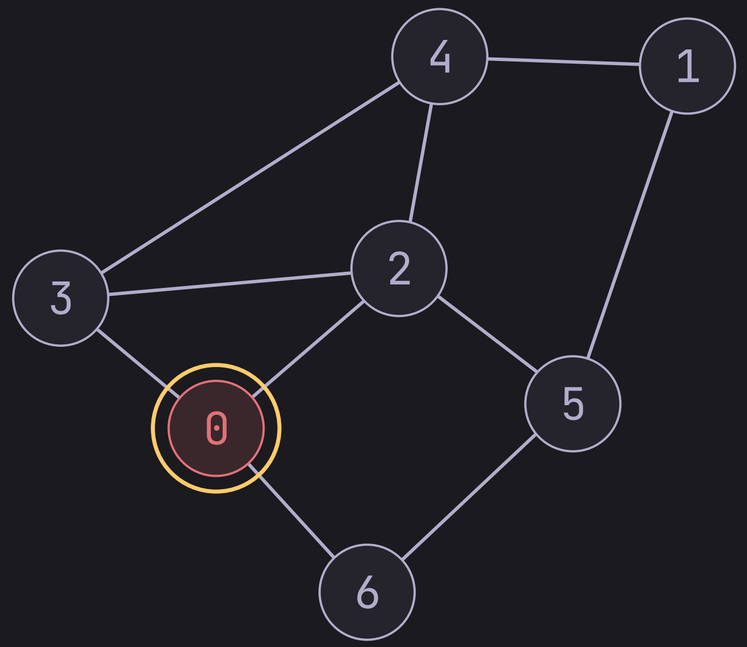
Breadth First Search






```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

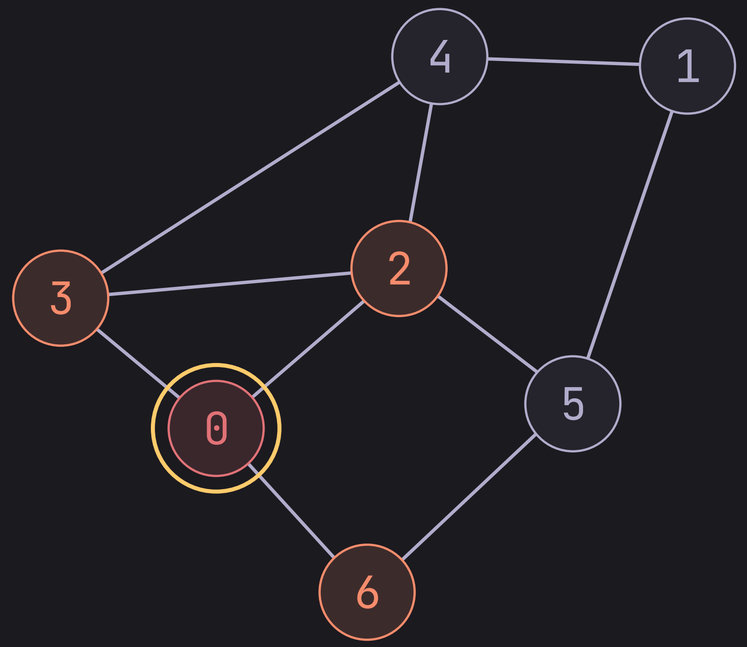
Breadth First Search



bfsQueue

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

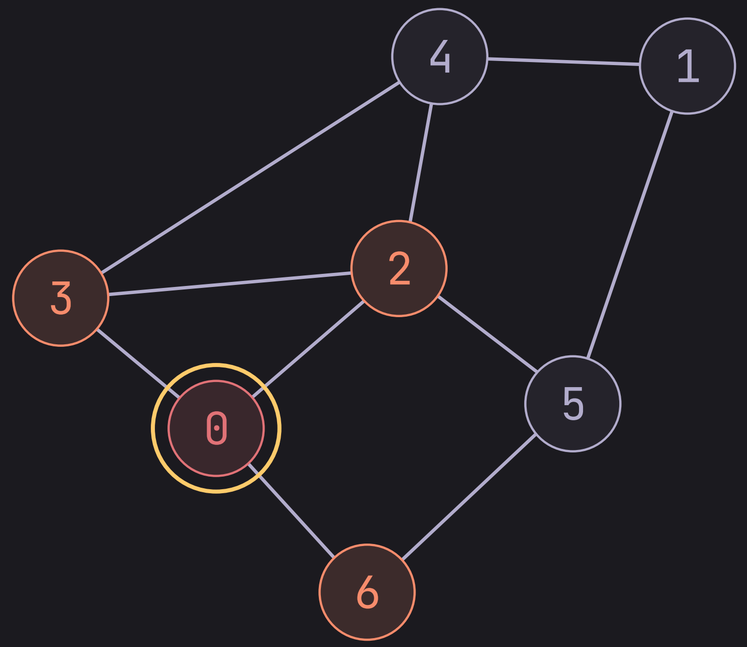
Breadth First Search



bfsQueue

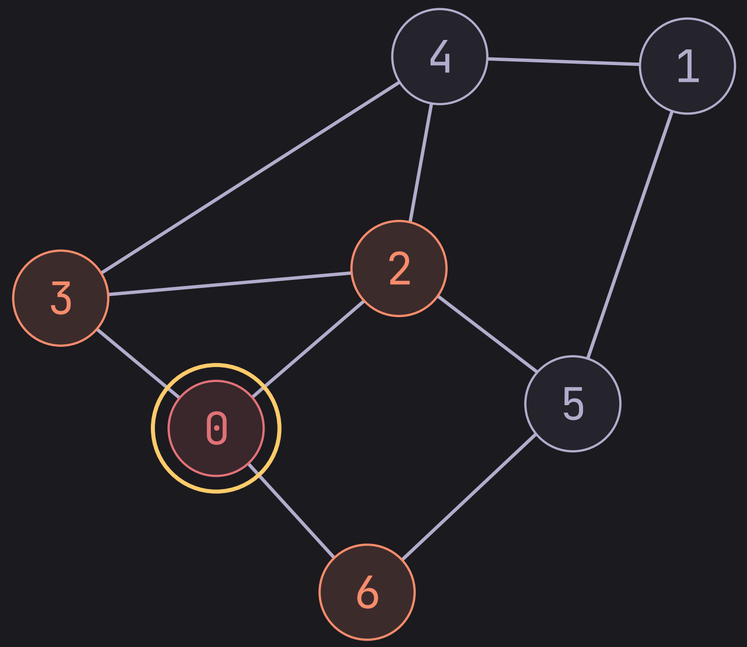
```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
    ⟶  visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



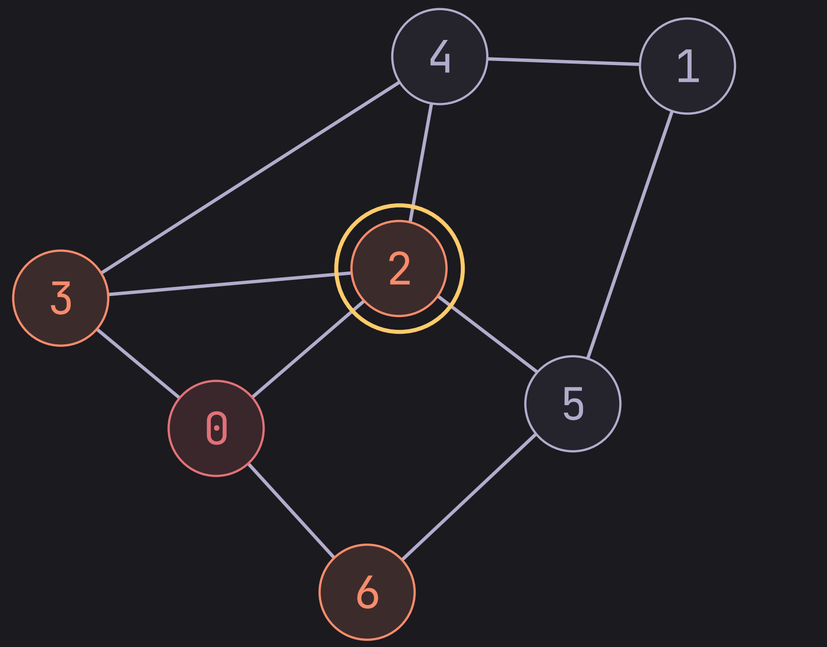

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
      ⟶ bfsQueue.push(u);
      }
    }
  }
}
```

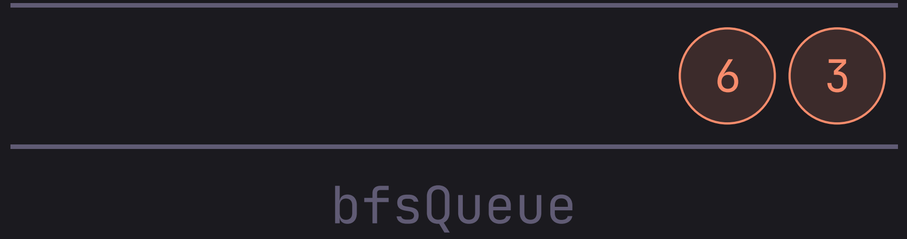
Breadth First Search




```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

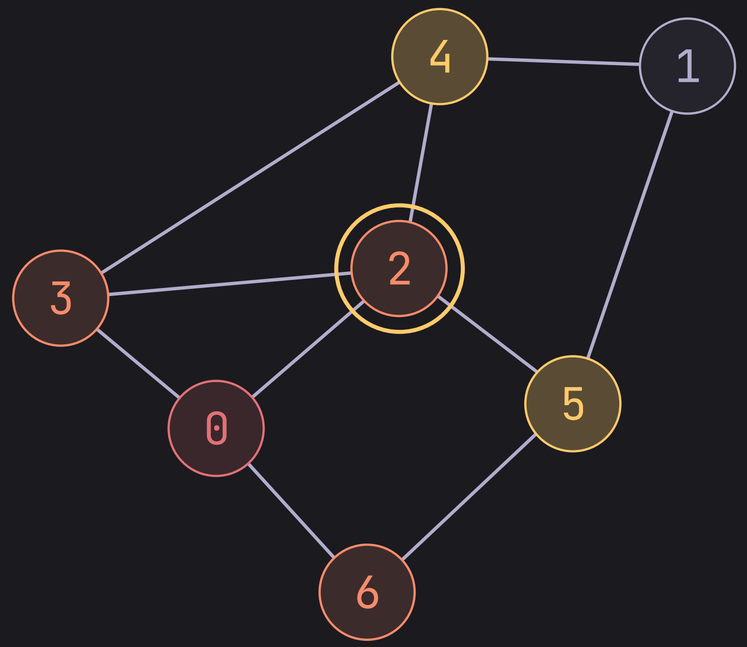
Breadth First Search

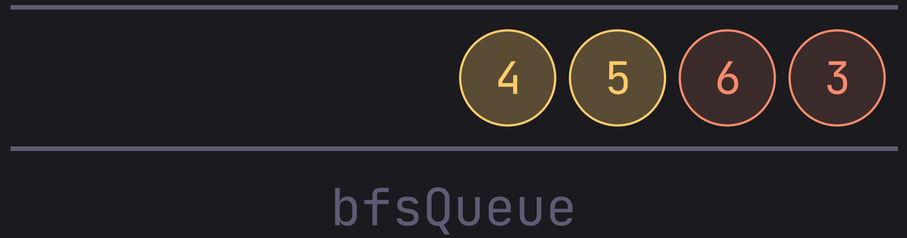





```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

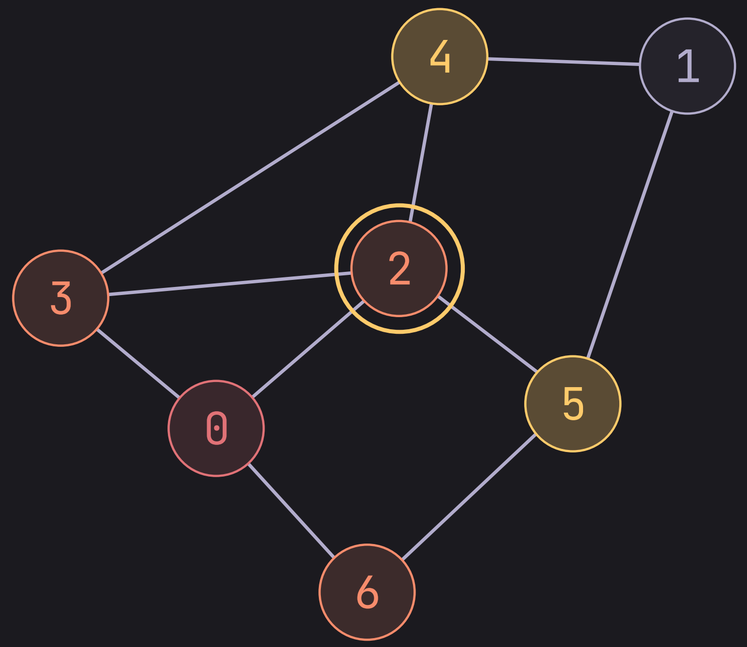
Breadth First Search

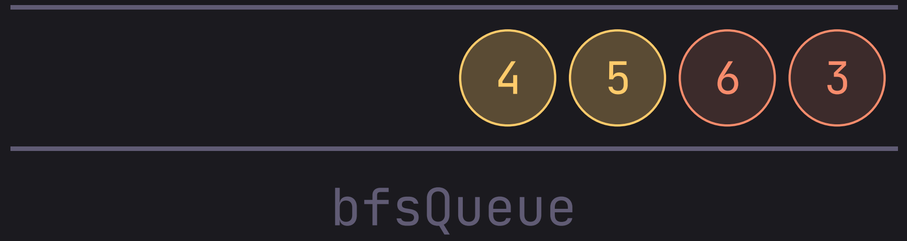





```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
      ⟶ bfsQueue.push(u);
        }
    }
  }
}
```

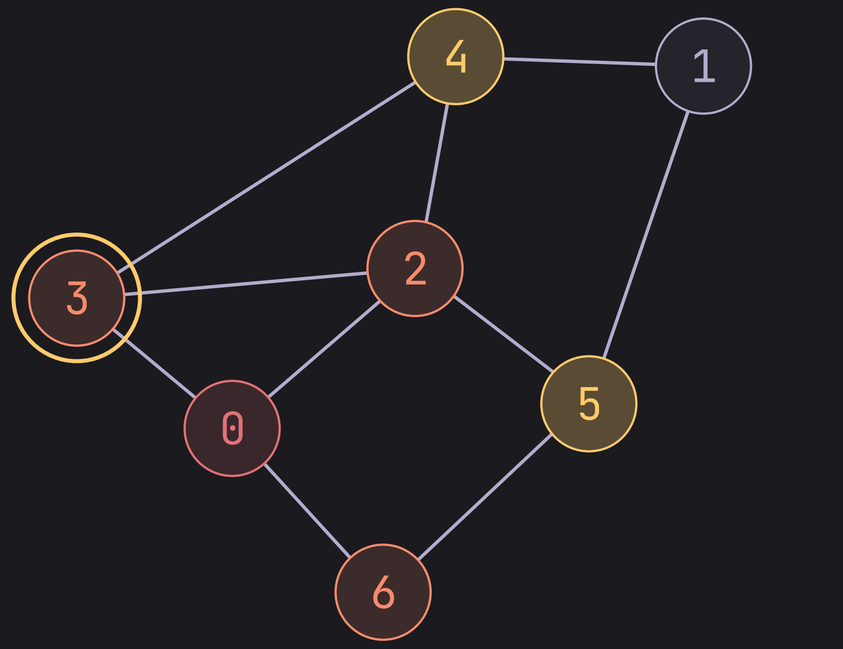
Breadth First Search





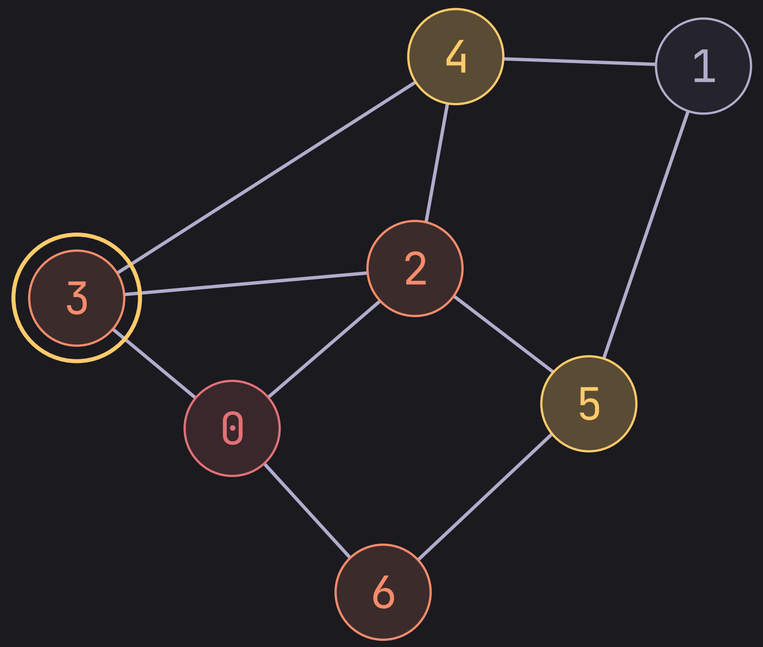

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



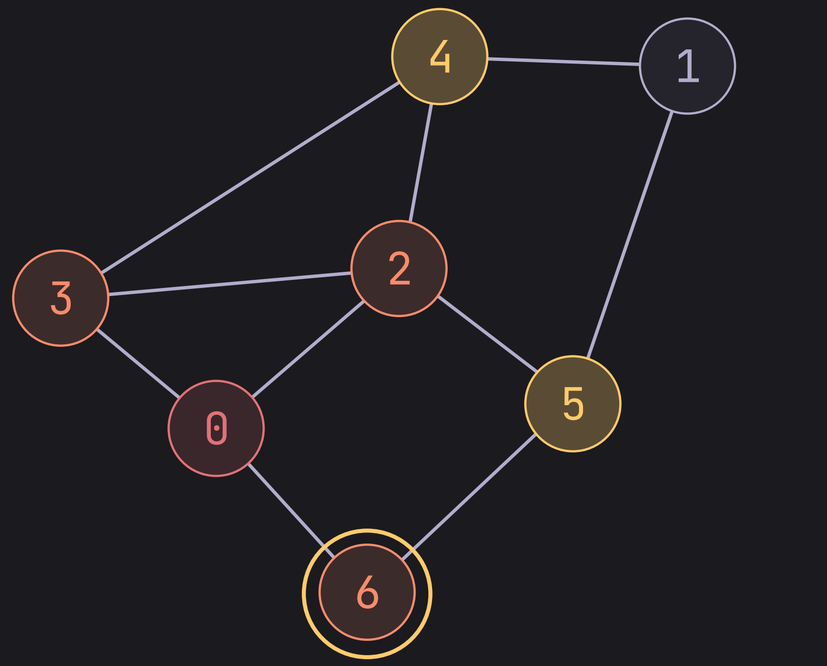

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
→   for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

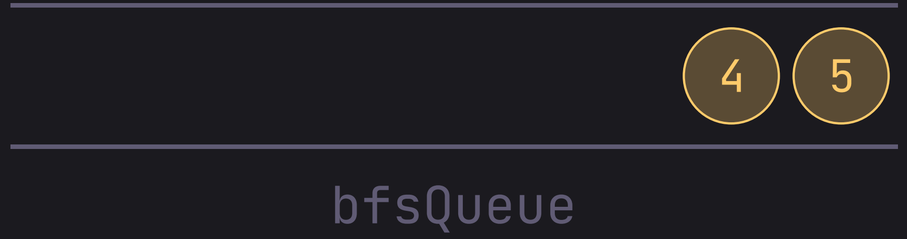
Breadth First Search




```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

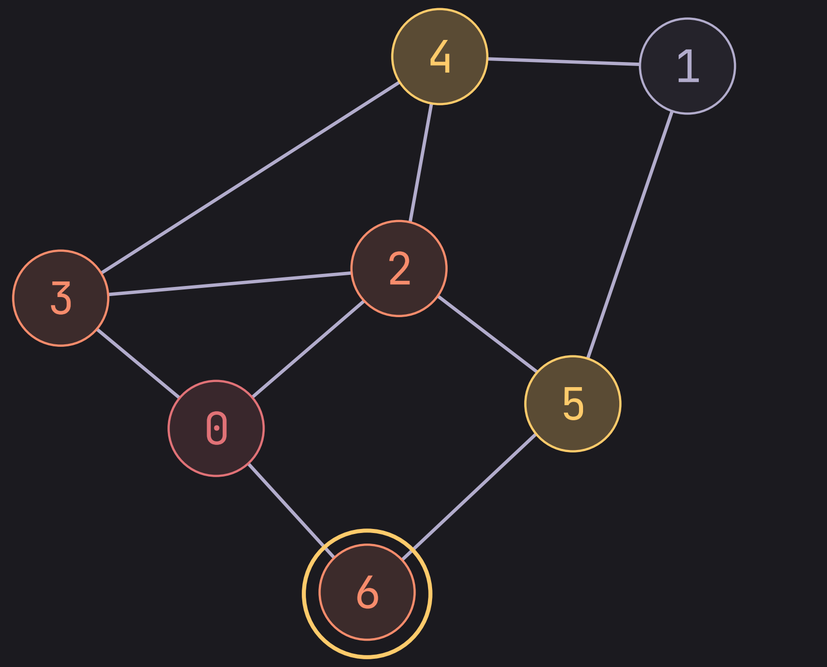
Breadth First Search

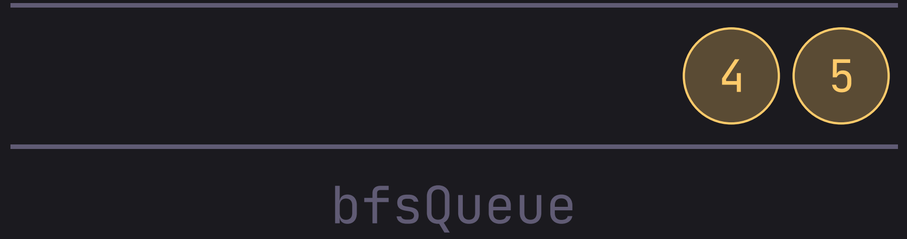





```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

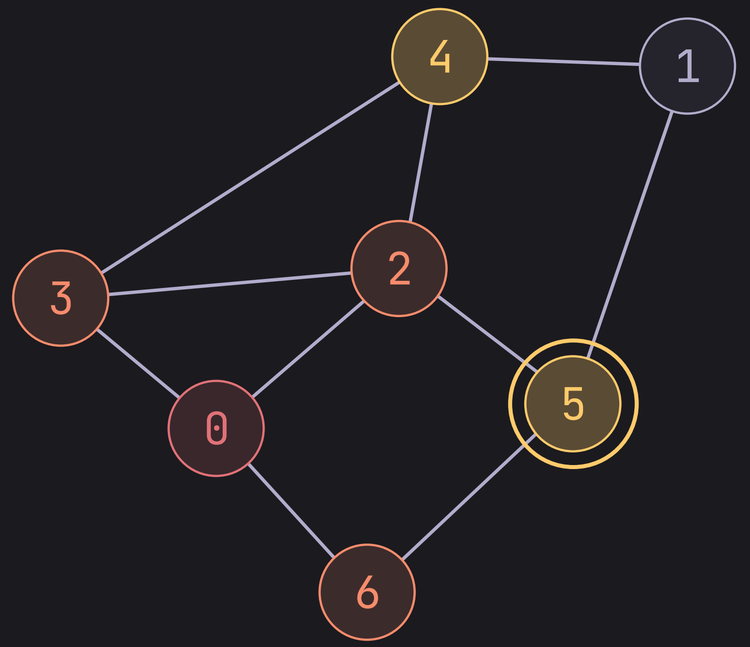
Breadth First Search

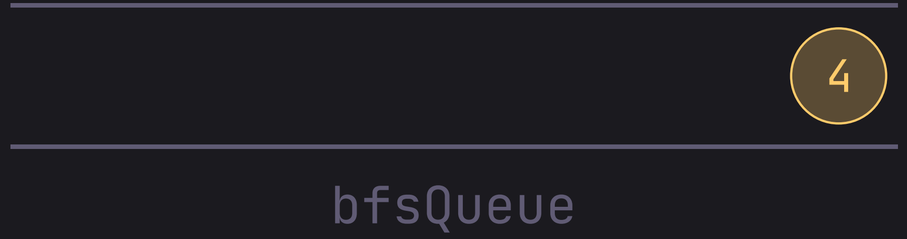





```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

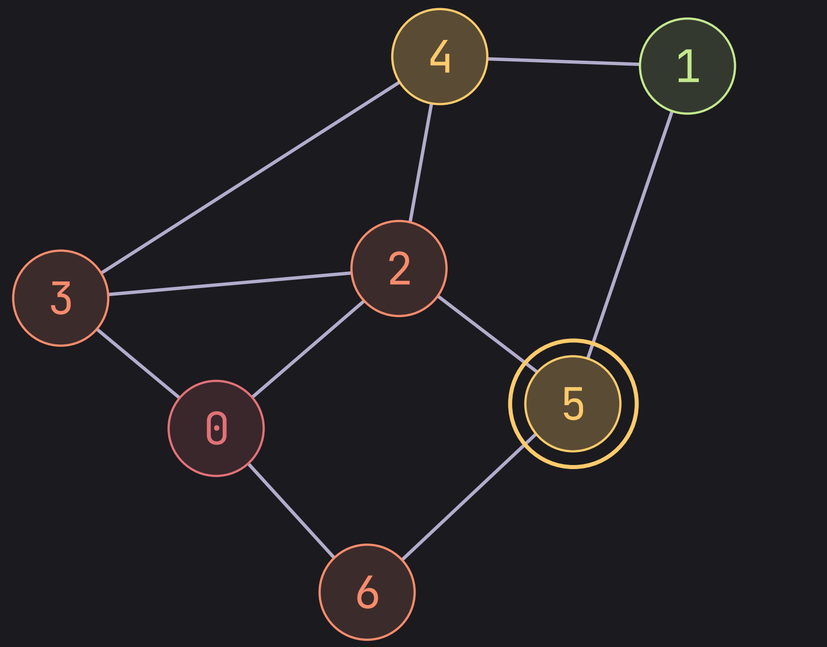
Breadth First Search

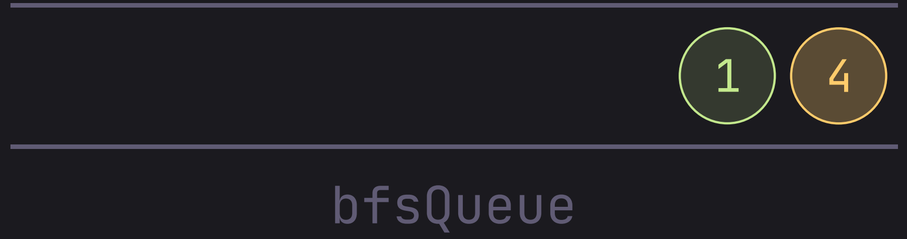





```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

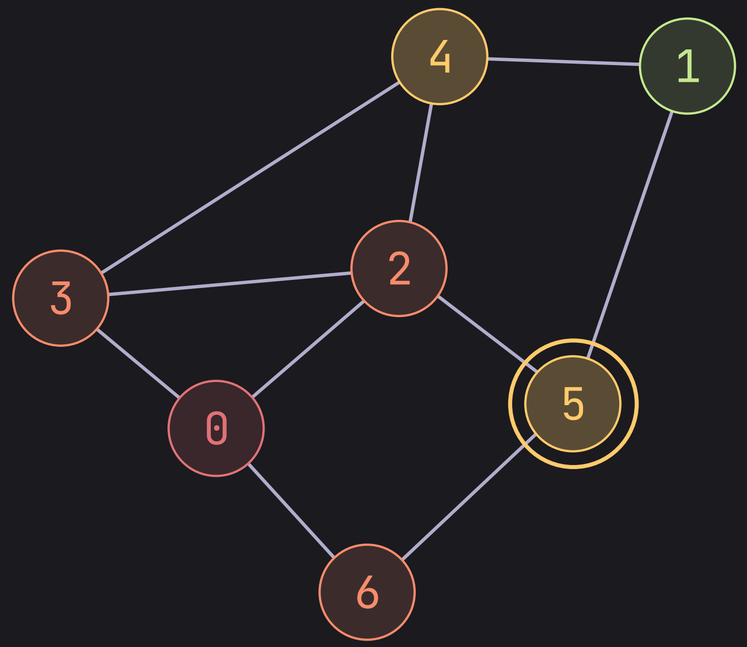
Breadth First Search

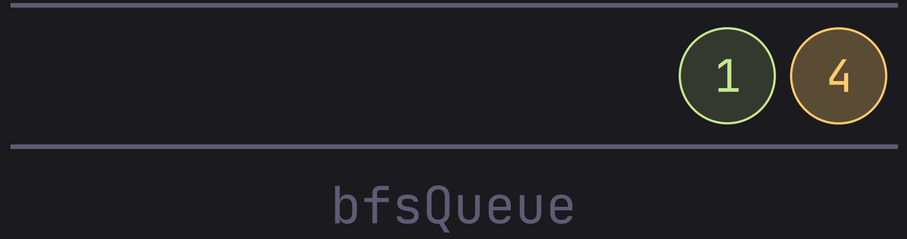





```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
      ⟶ bfsQueue.push(u);
      }
    }
  }
}
```

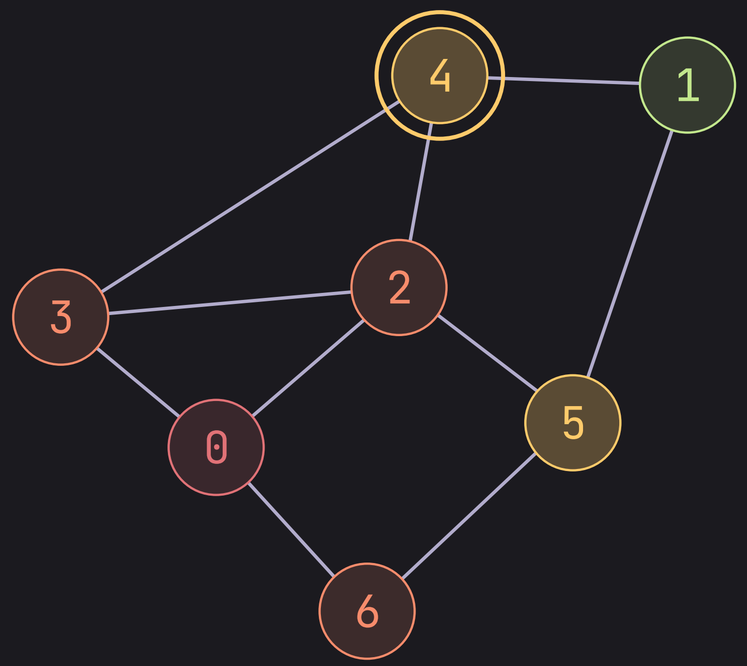
Breadth First Search

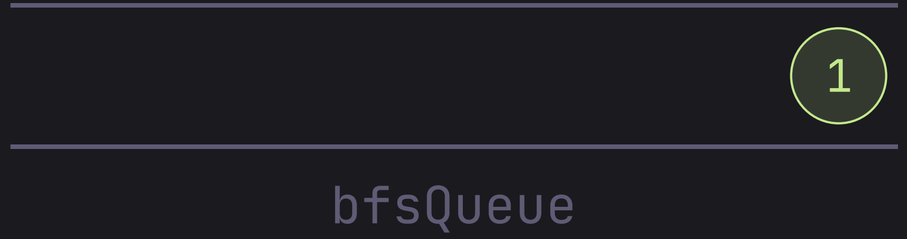





```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

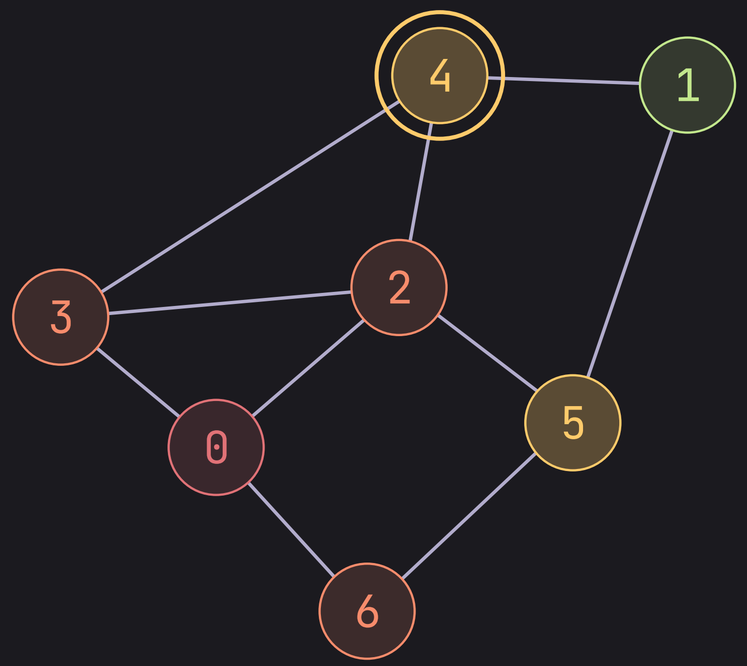
Breadth First Search

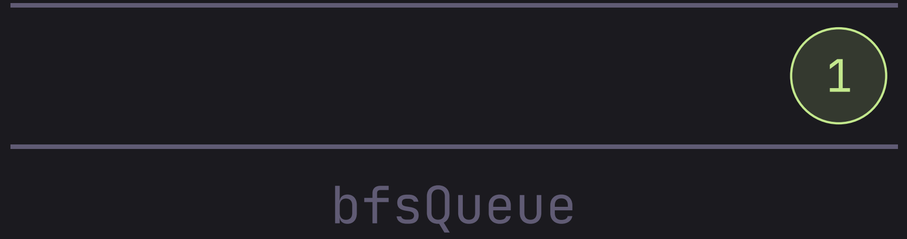





```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

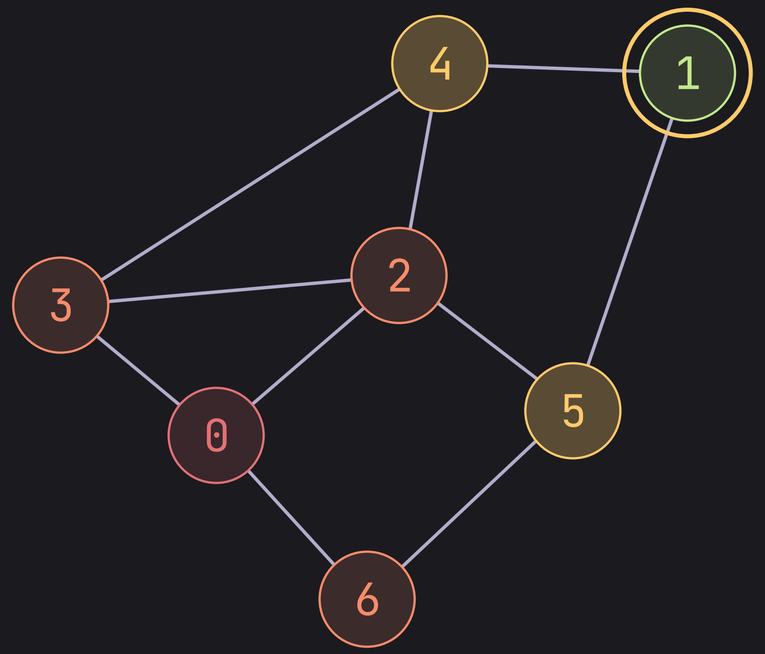
Breadth First Search






```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

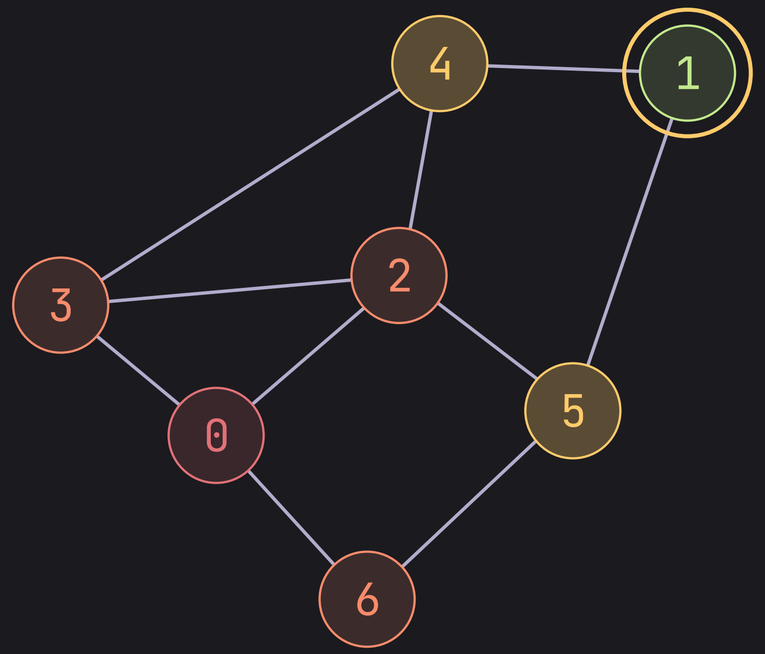
Breadth First Search



bfsQueue

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

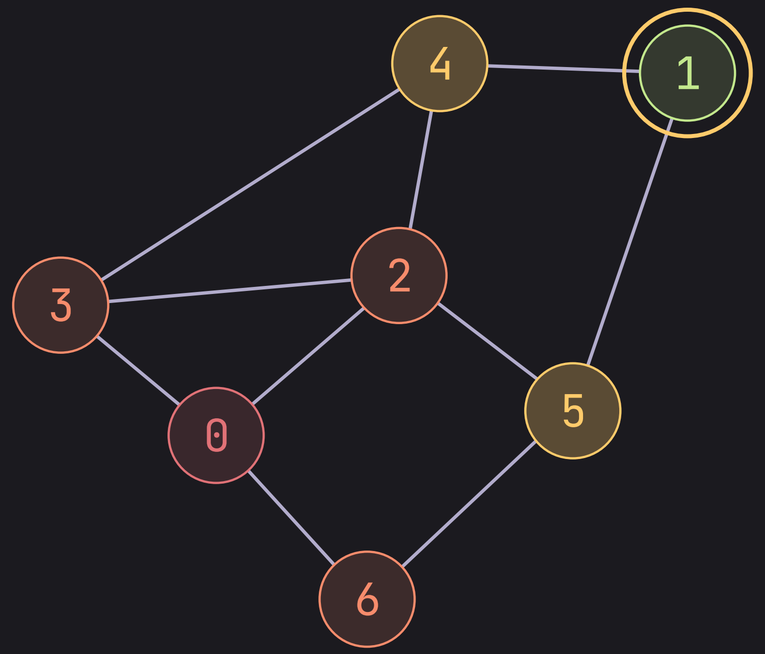
Breadth First Search



bfsQueue

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

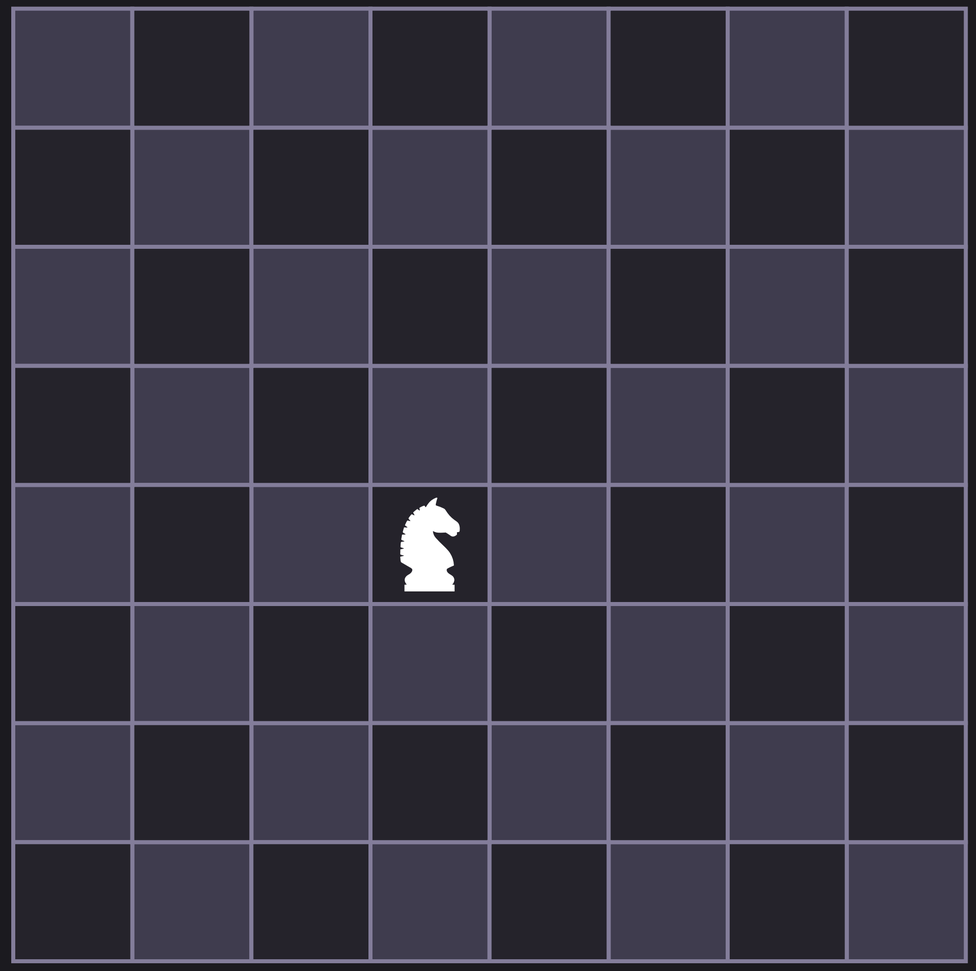
Breadth First Search



bfsQueue

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
→ }
```

Knight Moves

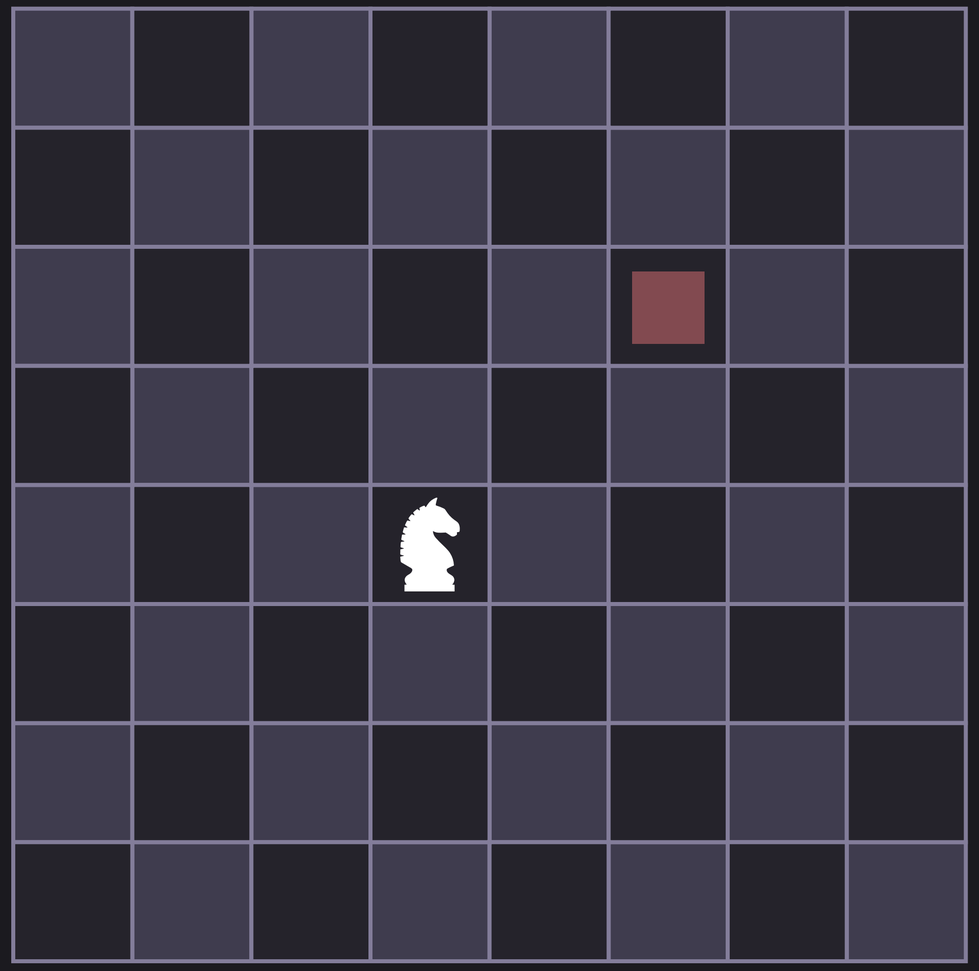


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

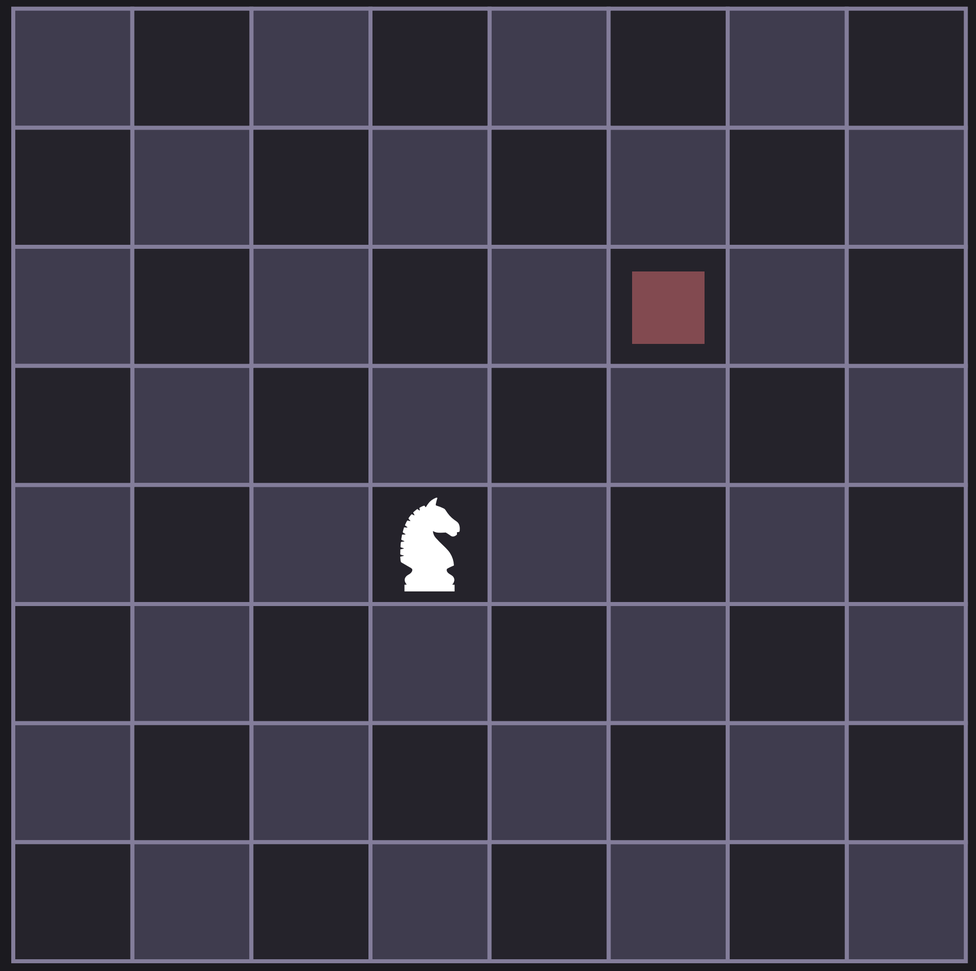


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

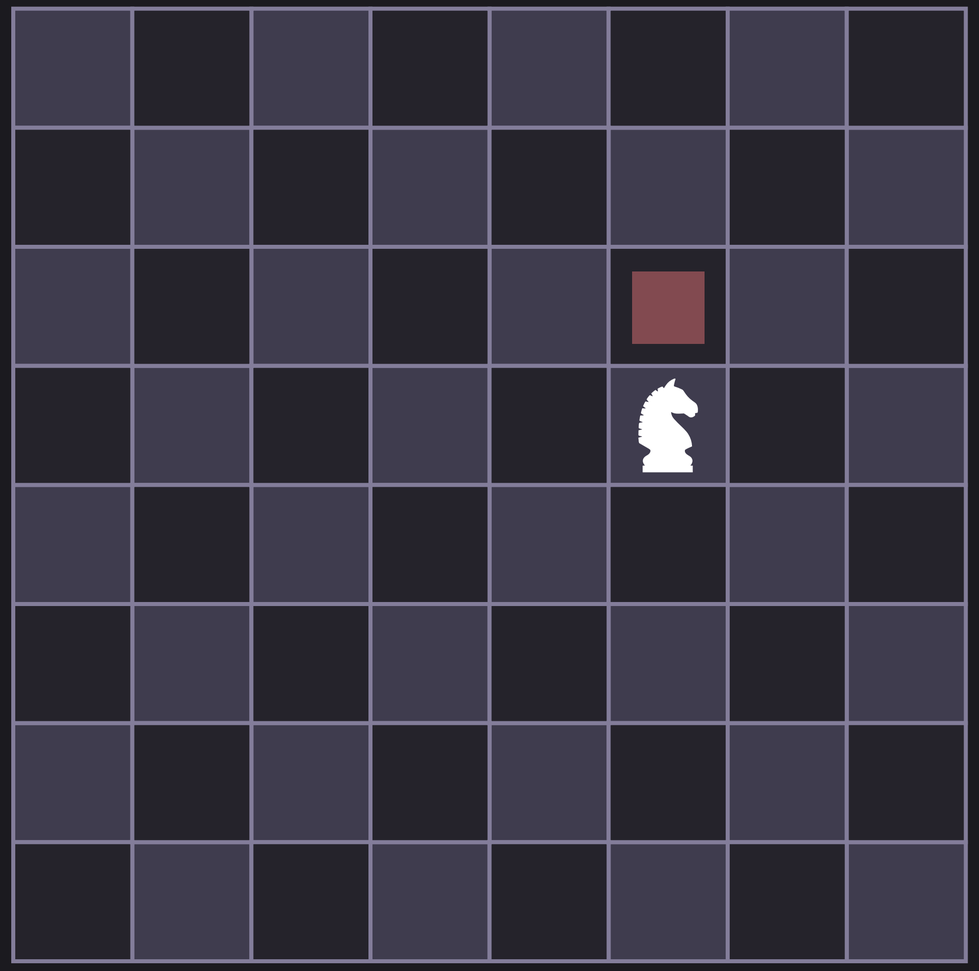


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

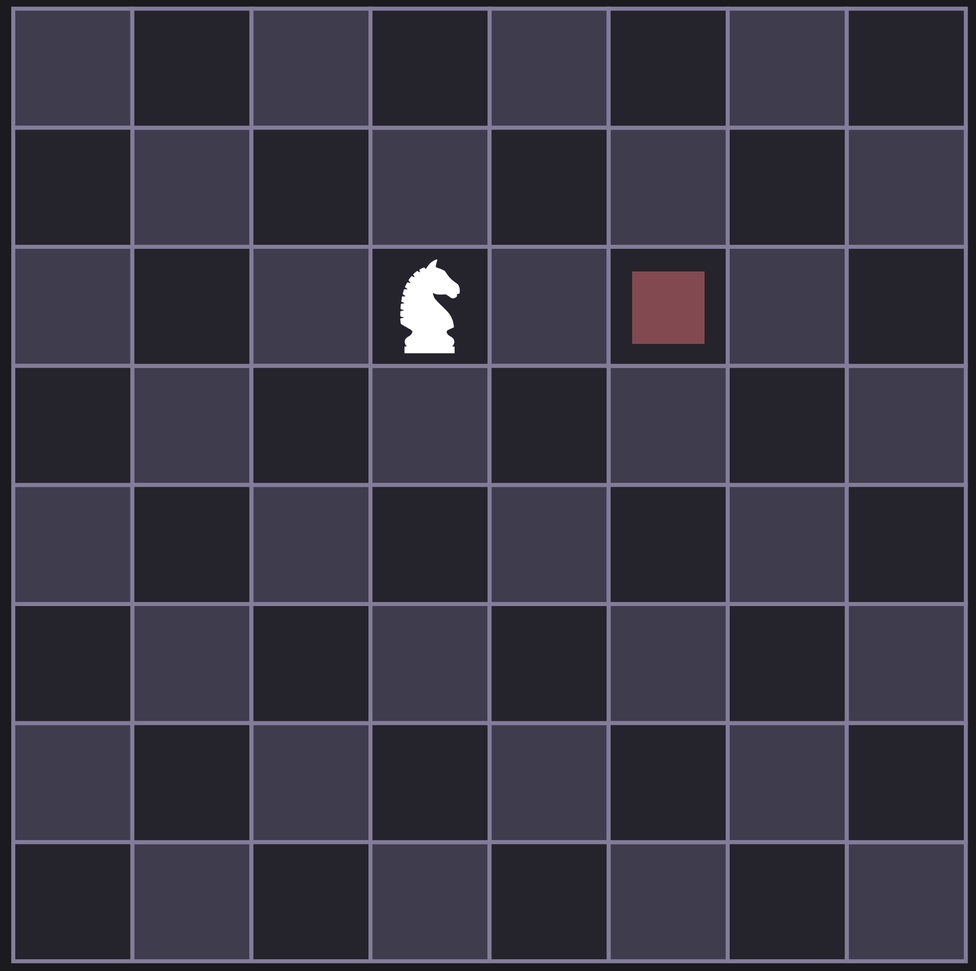


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

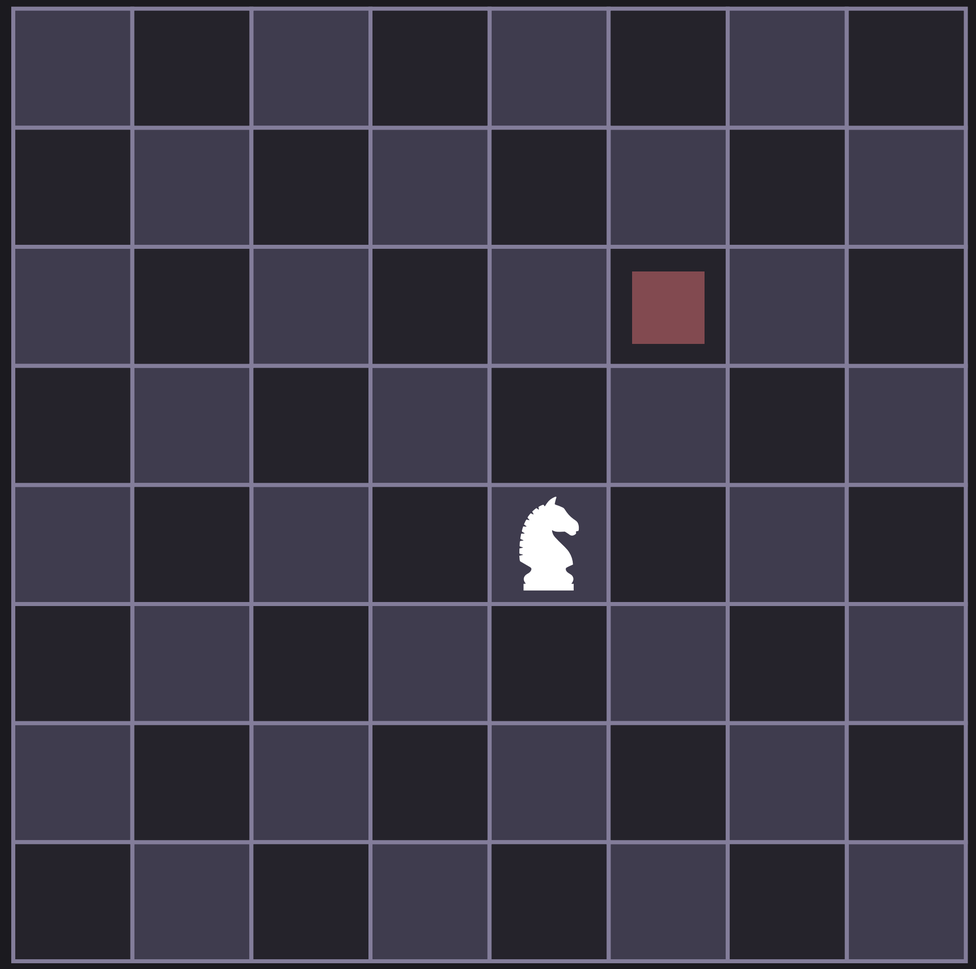


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

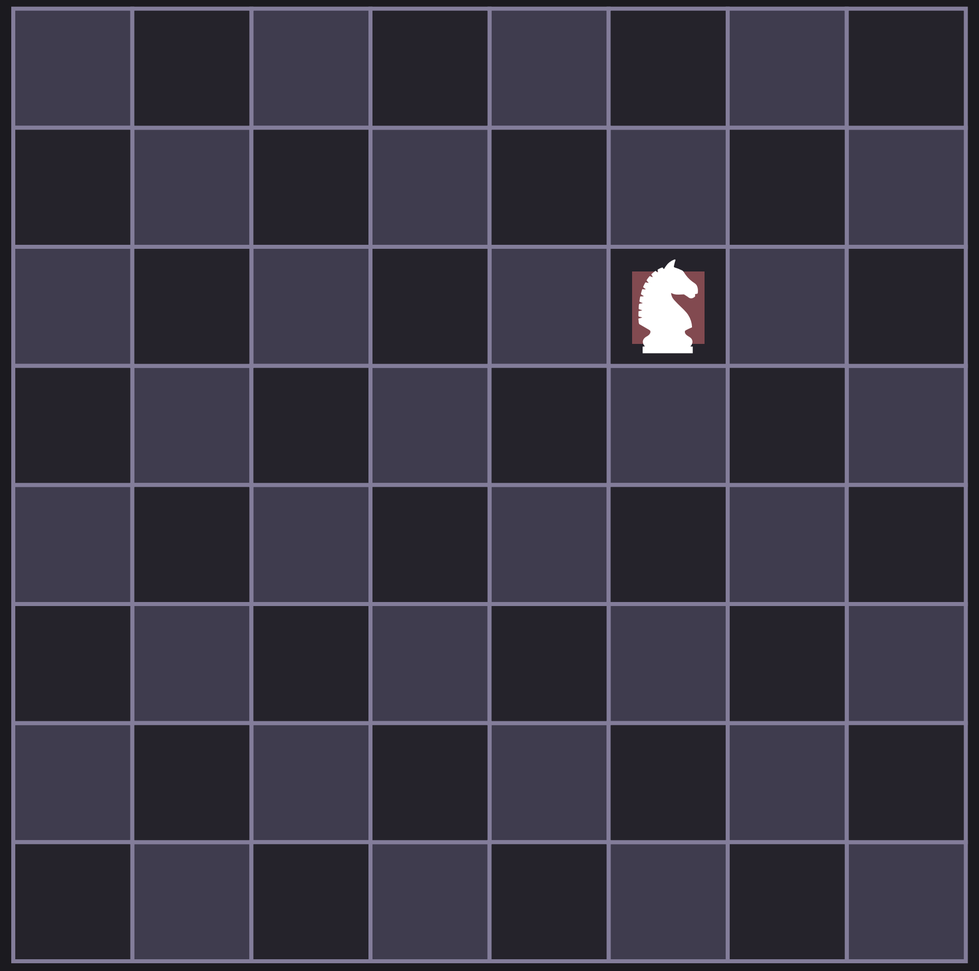


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

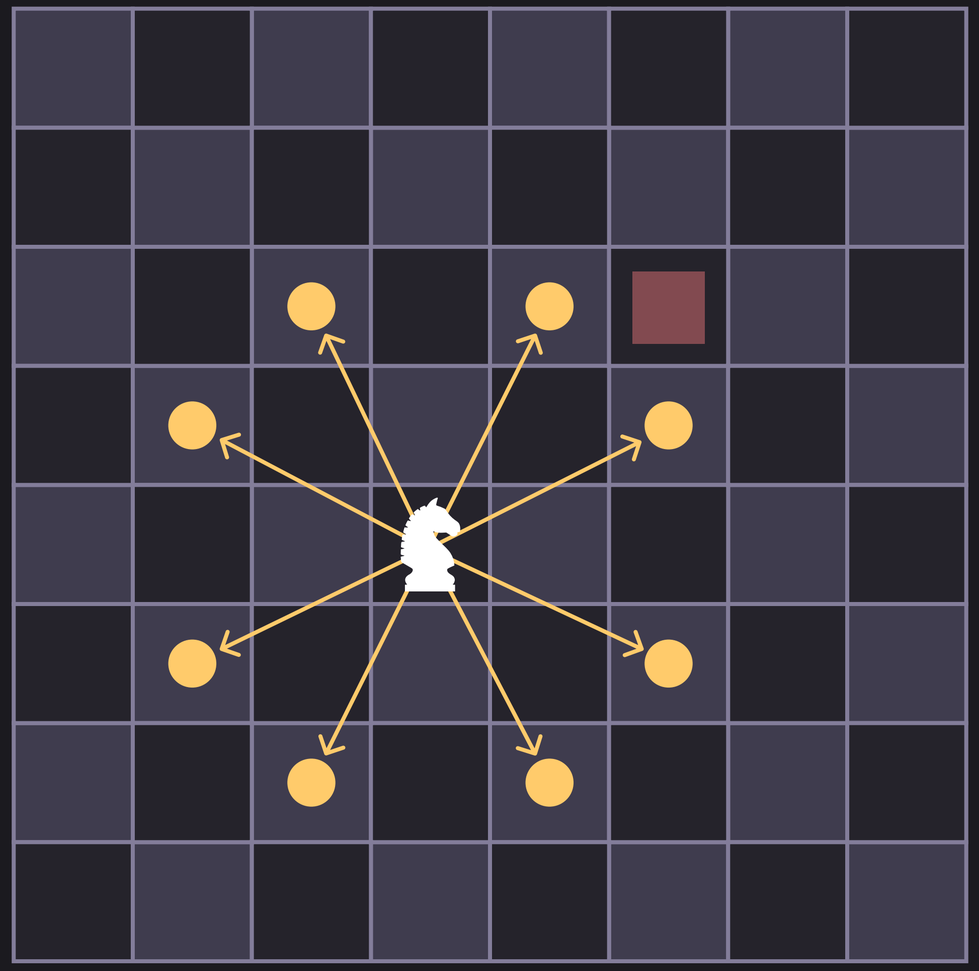


Imagine an infinite chessboard and a knight sitting on square (0, 0).

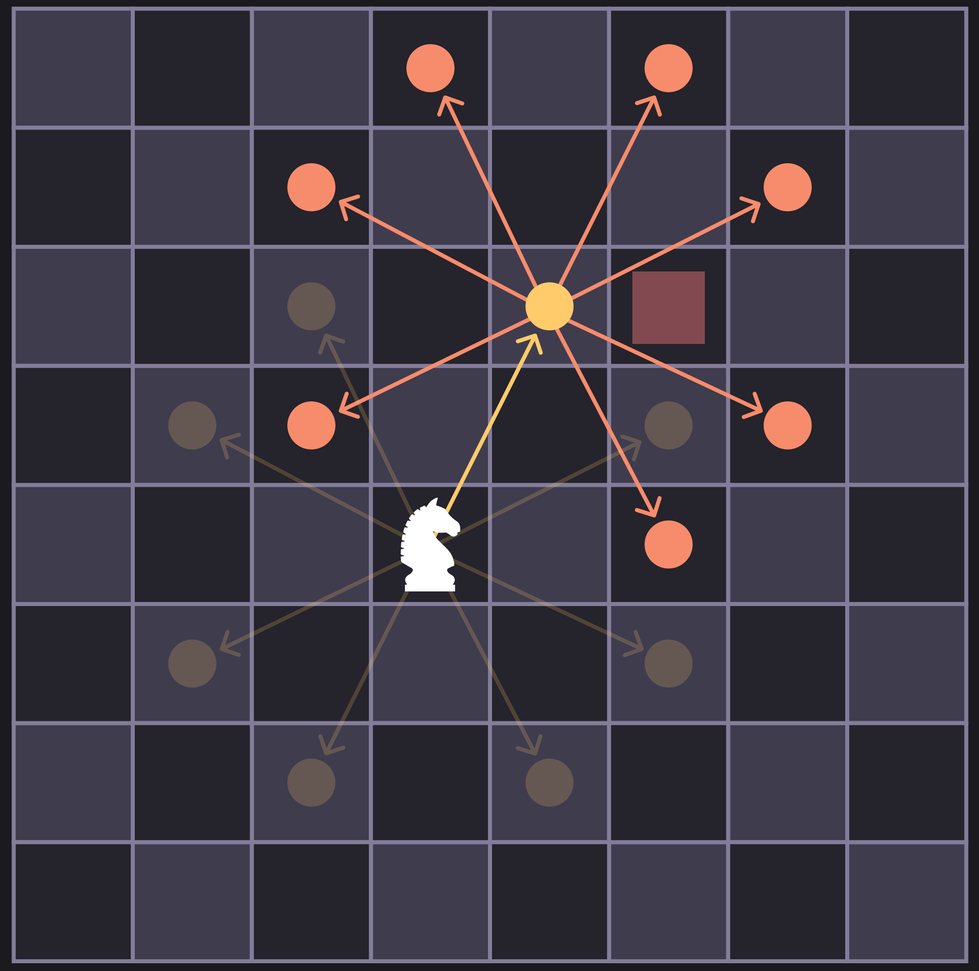
We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

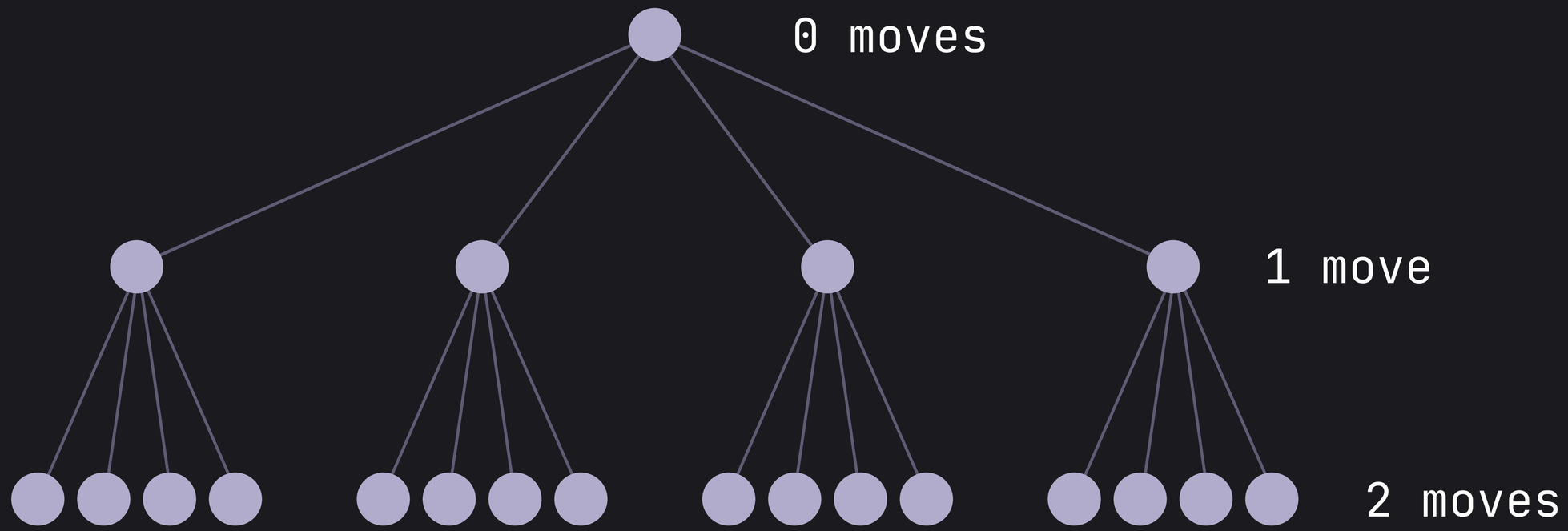
Knight Moves



Knight Moves

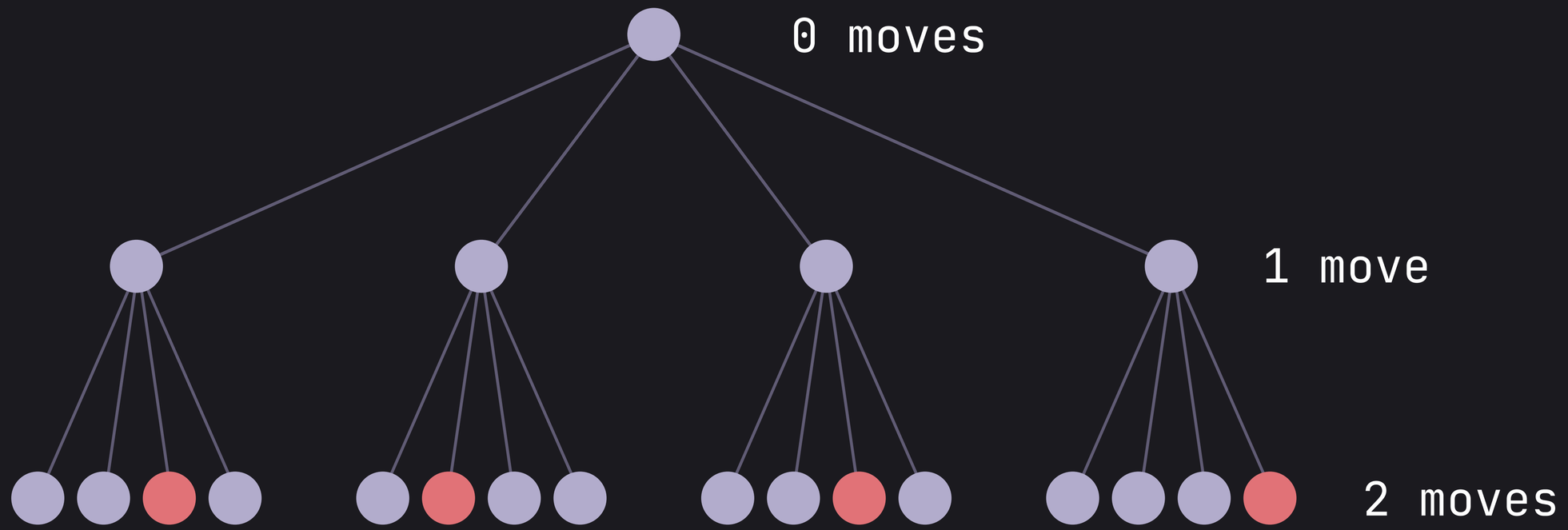


Knight Moves



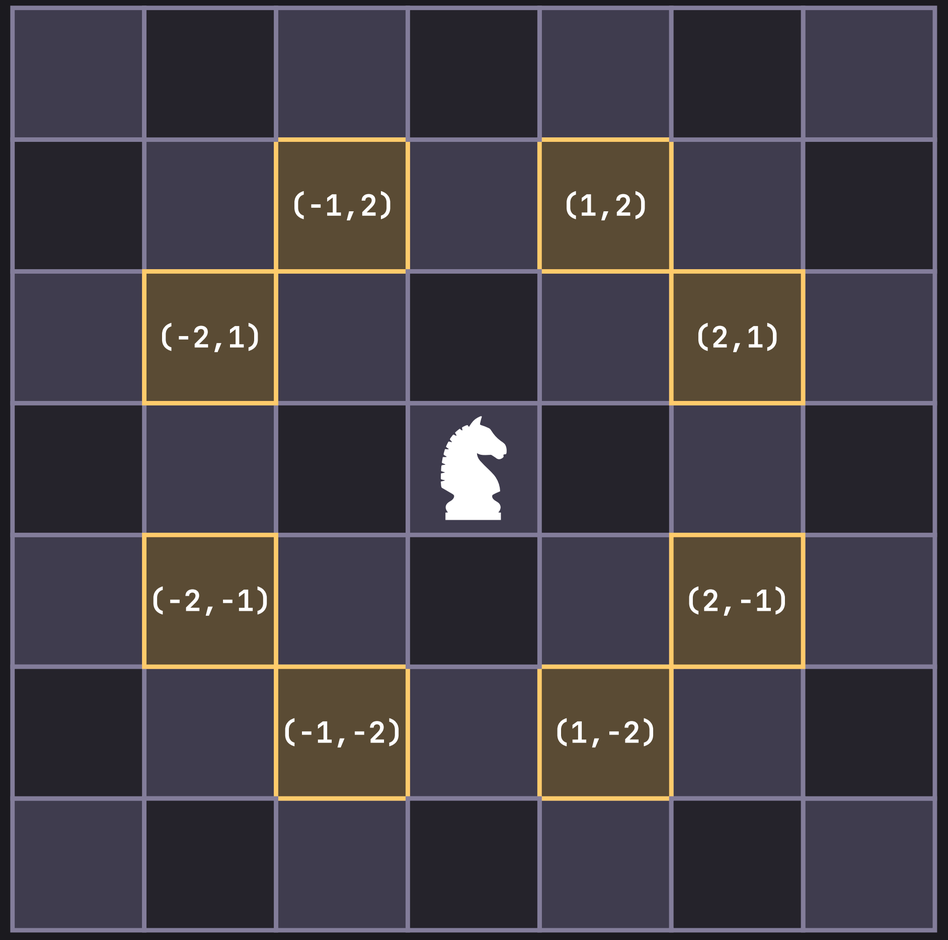
To find the shortest path we want to first explore all the vertices that are one move away, and then 2 moves away, and so forth

Knight Moves



We also don’t want to go in circles so we can ignore vertices we have already seen

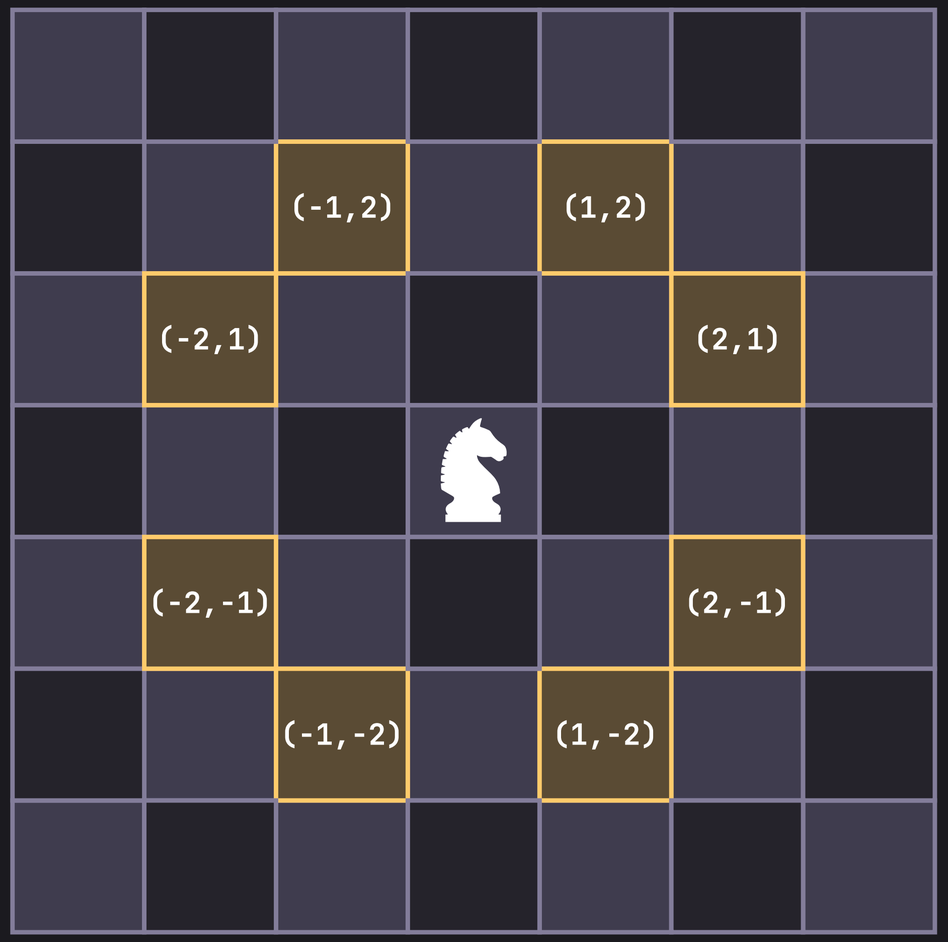
Night Moves



Now here is a **really** important idea!

We dont actually need to generate the whole graph. We just need to be able to calculate the neighbouring verticies

Night Moves



So if we are currently at point

$$(x,y)$$

Then the neighbouring points are

$$(x+2,y+1) \quad (x+1,y+2)$$

$$(x+2,y-1) \quad (x+1,y-2)$$

$$(x-2,y+1) \quad (x-1,y+2)$$

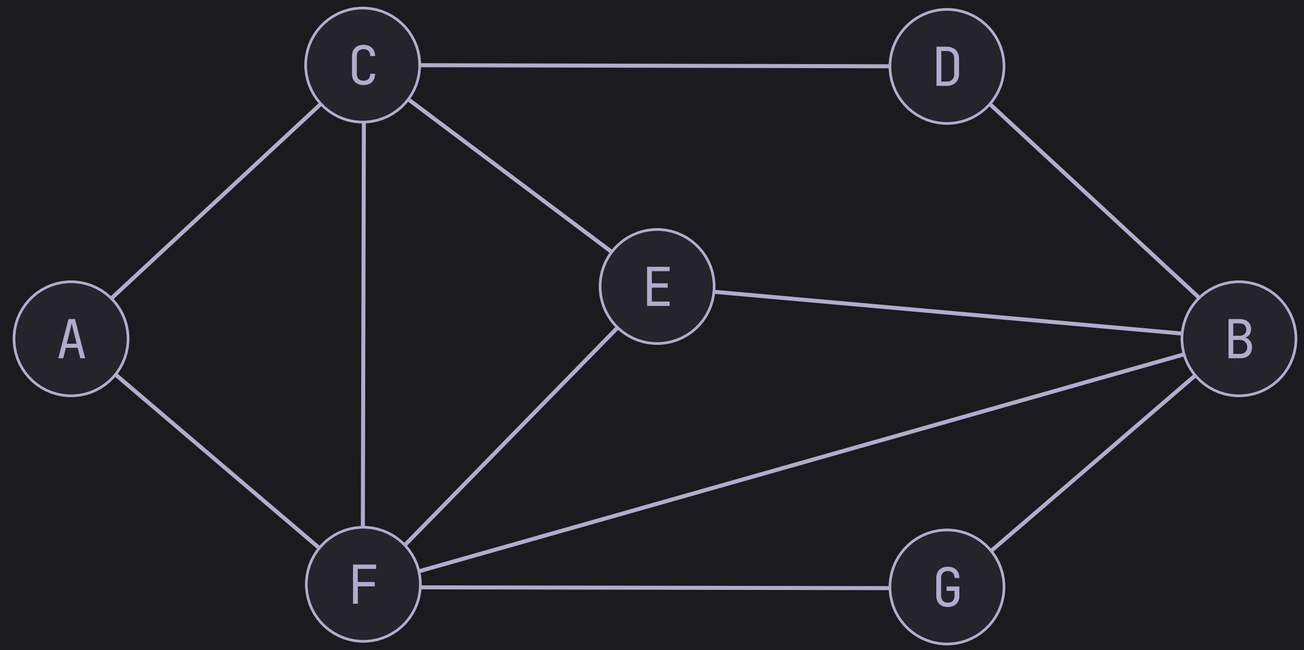
$$(x-2,y-1) \quad (x-1,y-2)$$

Night Moves

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

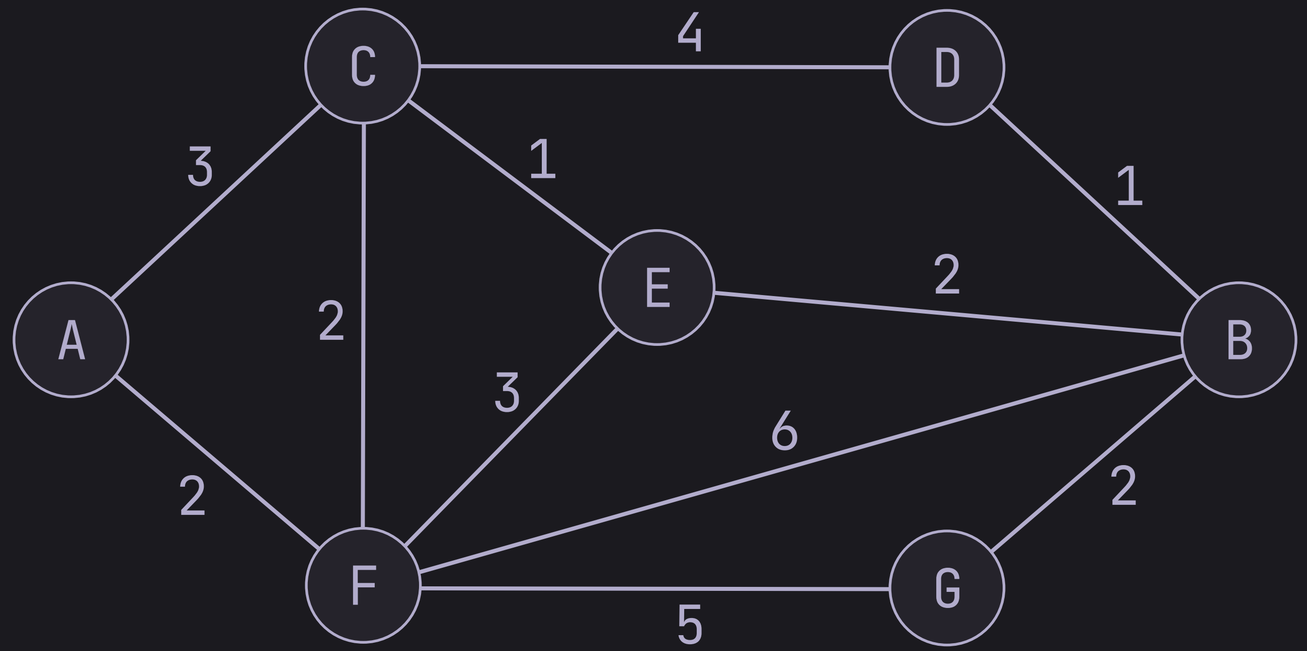
```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : getNeighbours(v)) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Weighted Graphs



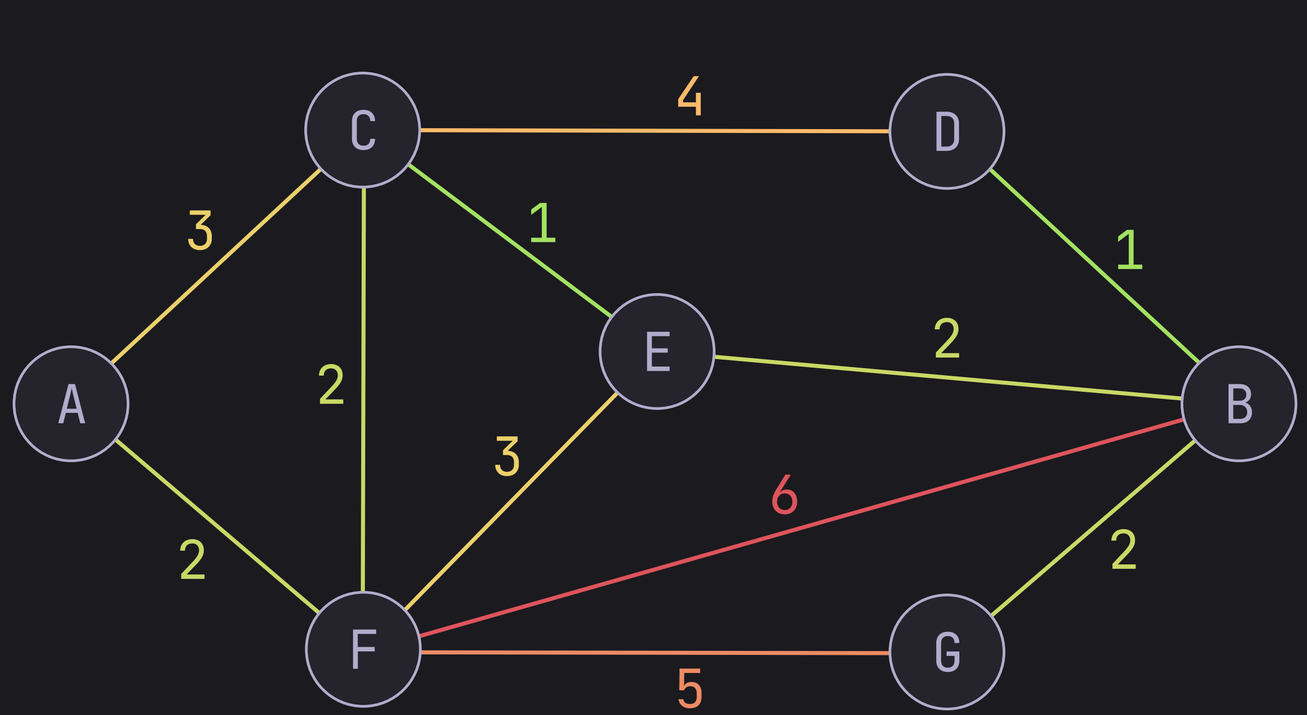
Now let take our regular graph

Weighted Graphs



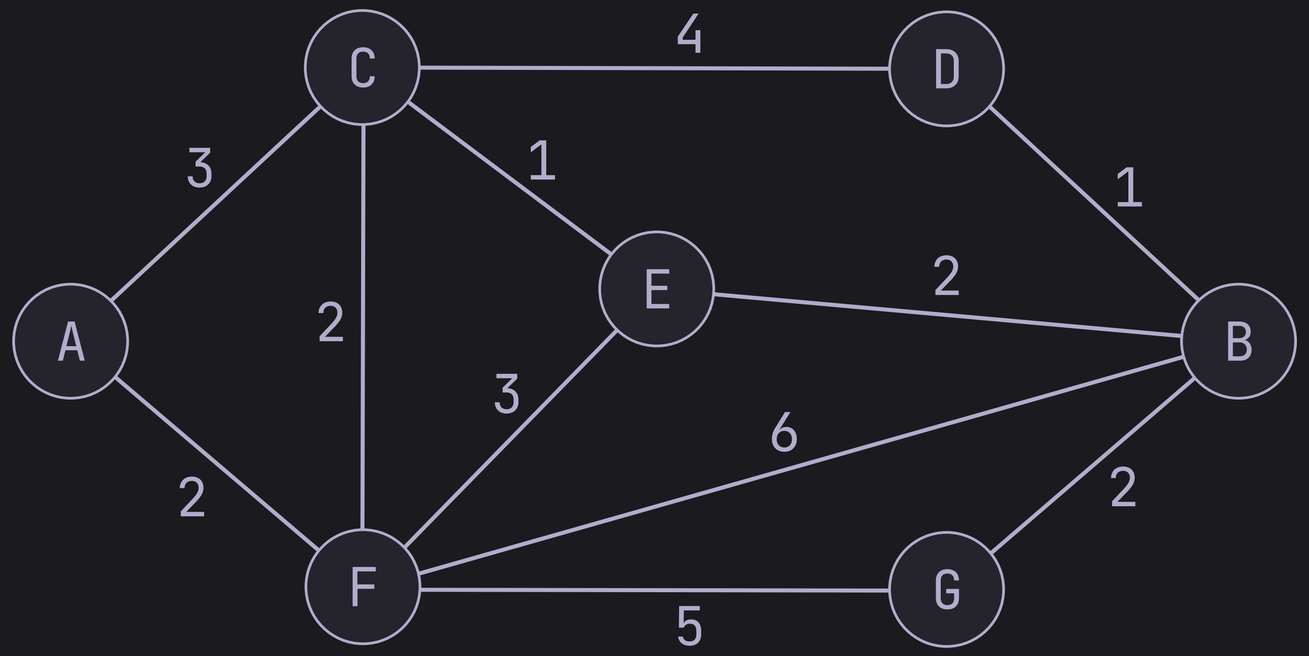
And add weights to the graph

Weighted Graphs



These weights can represent many different things

Weighted Graphs



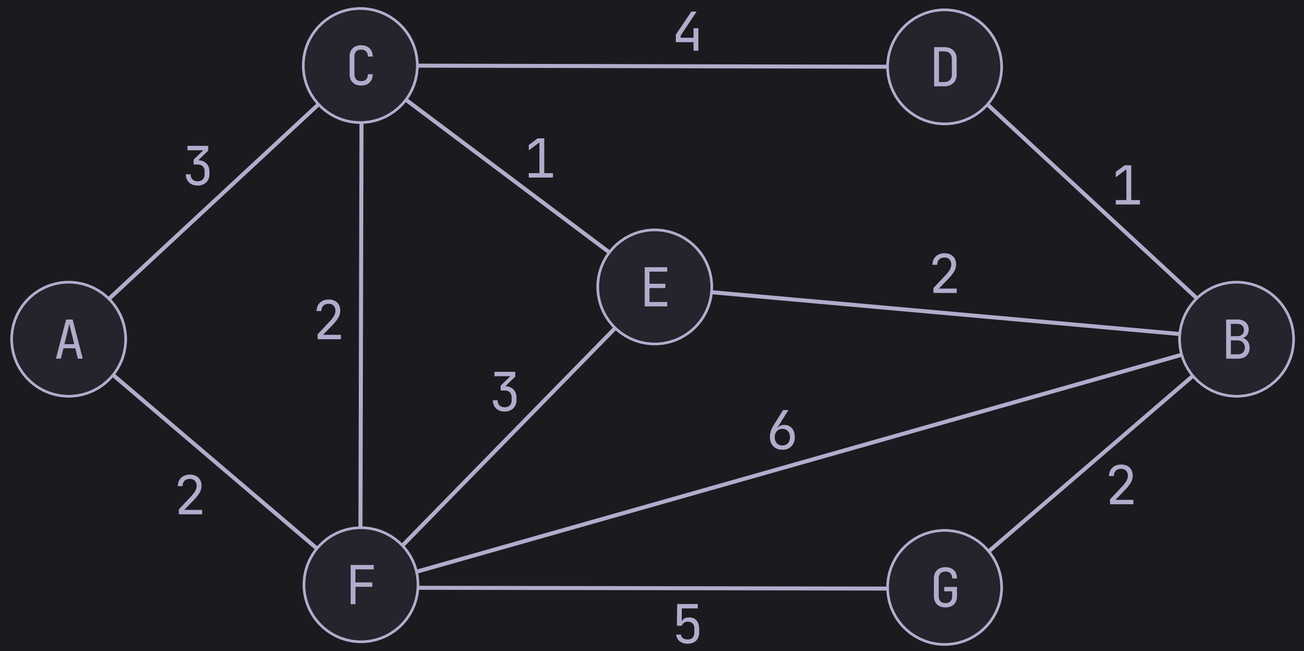
to

from

	A	B	C	D	E	F	G
A	∞	∞	3	∞	∞	2	∞
B	∞	∞	∞	1	2	6	2
C	3	∞	∞	4	1	2	∞
D	∞	1	4	∞	∞	∞	∞
E	∞	2	1	∞	∞	3	∞
F	2	6	2	∞	3	∞	5
G	∞	2	∞	∞	∞	5	∞

We can use an adjacency matrix to store these weights

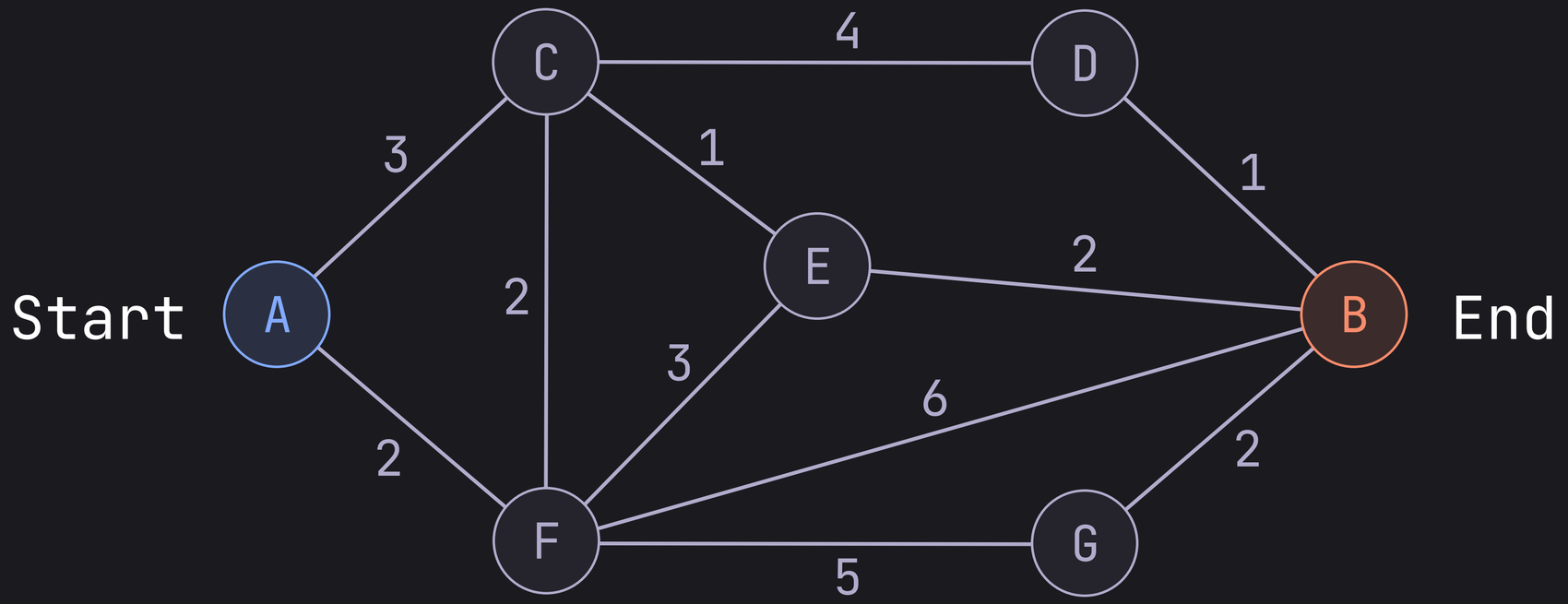
Weighted Graphs




```
A: {(C,3), (F,2)}
B: {(D,1), (E,2), (G,2)}
C: {(A,3), (E,1), (F,2)}
D: {(B,1), (C,4)}
E: {(B,2), (C,1), (F,3)}
F: {(A,2), (B,6), (C,2), (G,5)}
G: {(B,2), (F,5)}
```

Or we can use an adjacency list to store these weights

Shortest Path



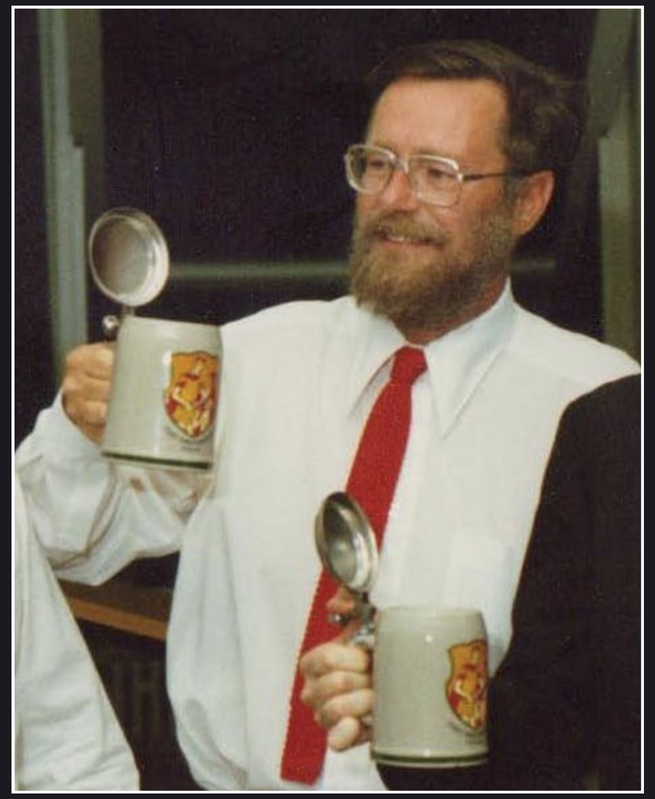
We want to find the **shortest path** from the **start** to the **end**

Dijkstra Algorithm

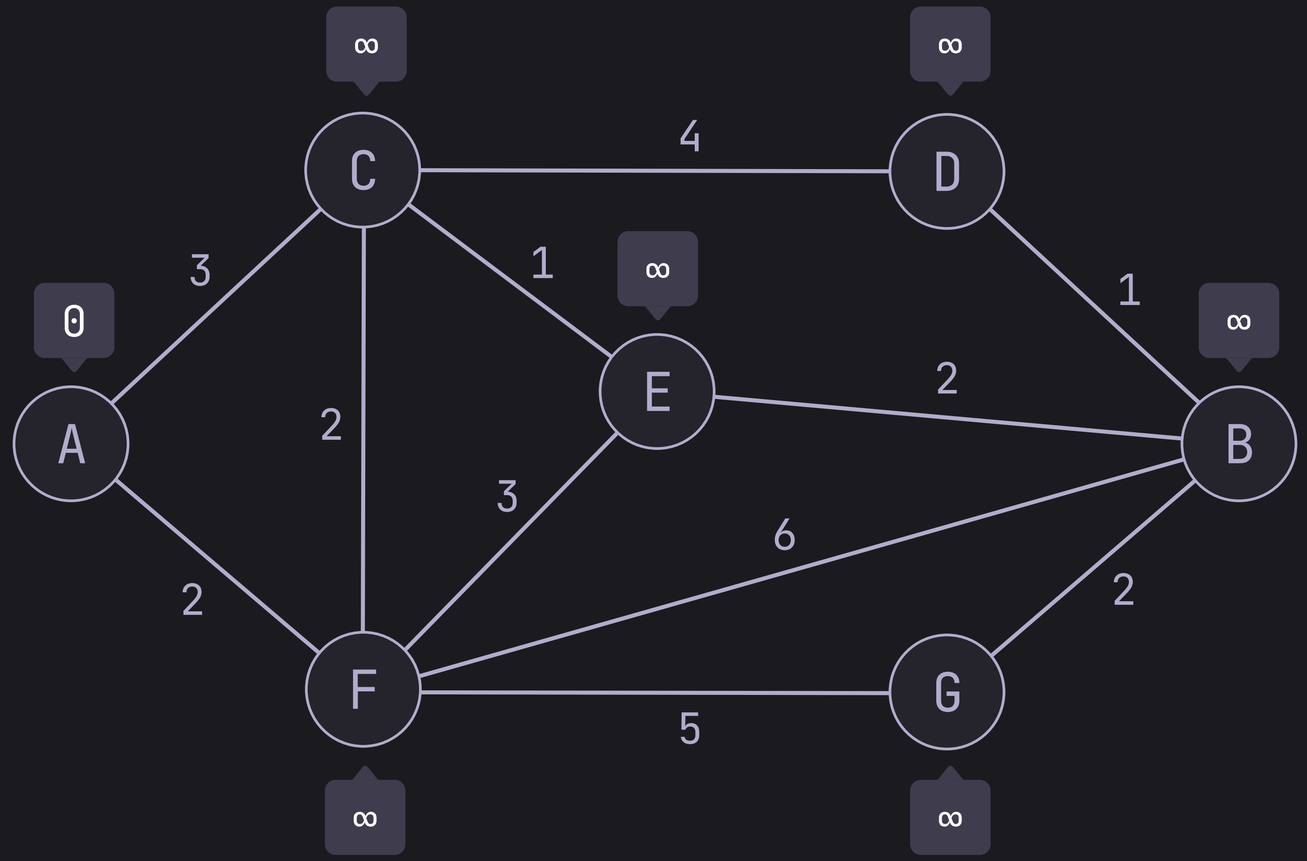
To the rescue! 🦸‍♀️

Who is Dijkstra?

- Edsger Dijkstra was a Dutch computer scientist and one of the most influential figures in the field of computer science.
- He received the Turing Award in 1972 for his fundamental contributions to the field of programming languages and the design of high-level programming systems.
- Hated the "goto" statement in programming

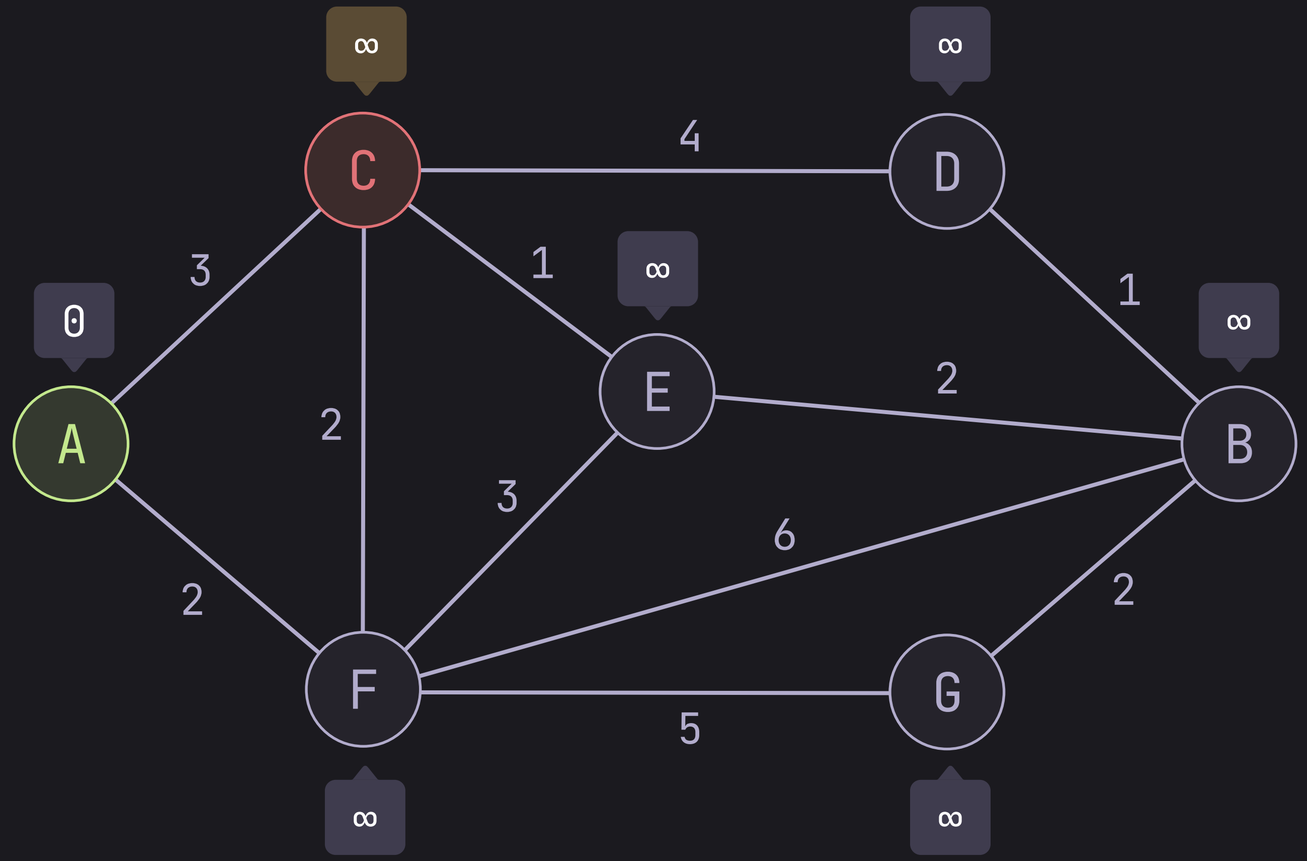


Dijkstra's Algorithm



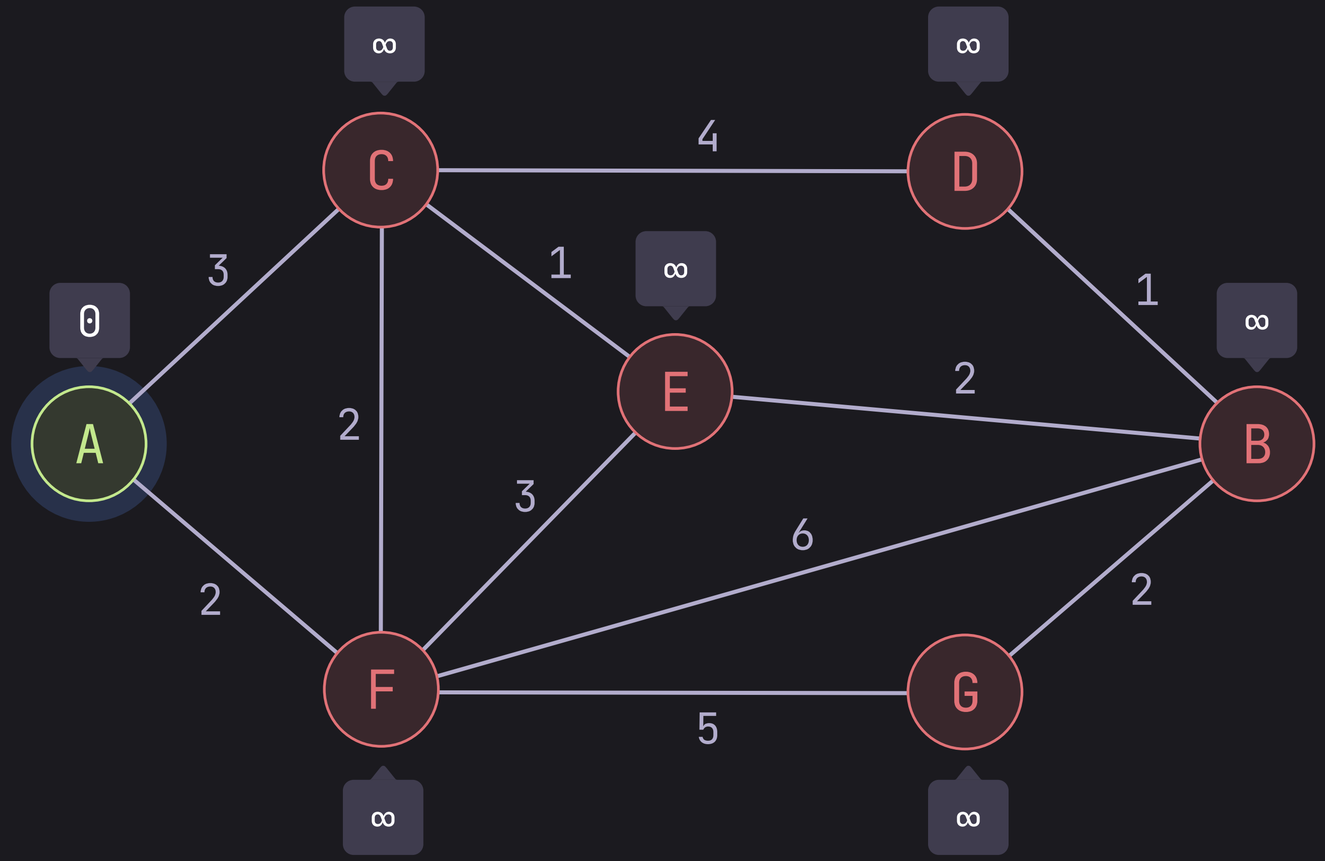
So with this algorithm we keep track of **bestDistanceTo** each node

Dijkstra's Algorithm



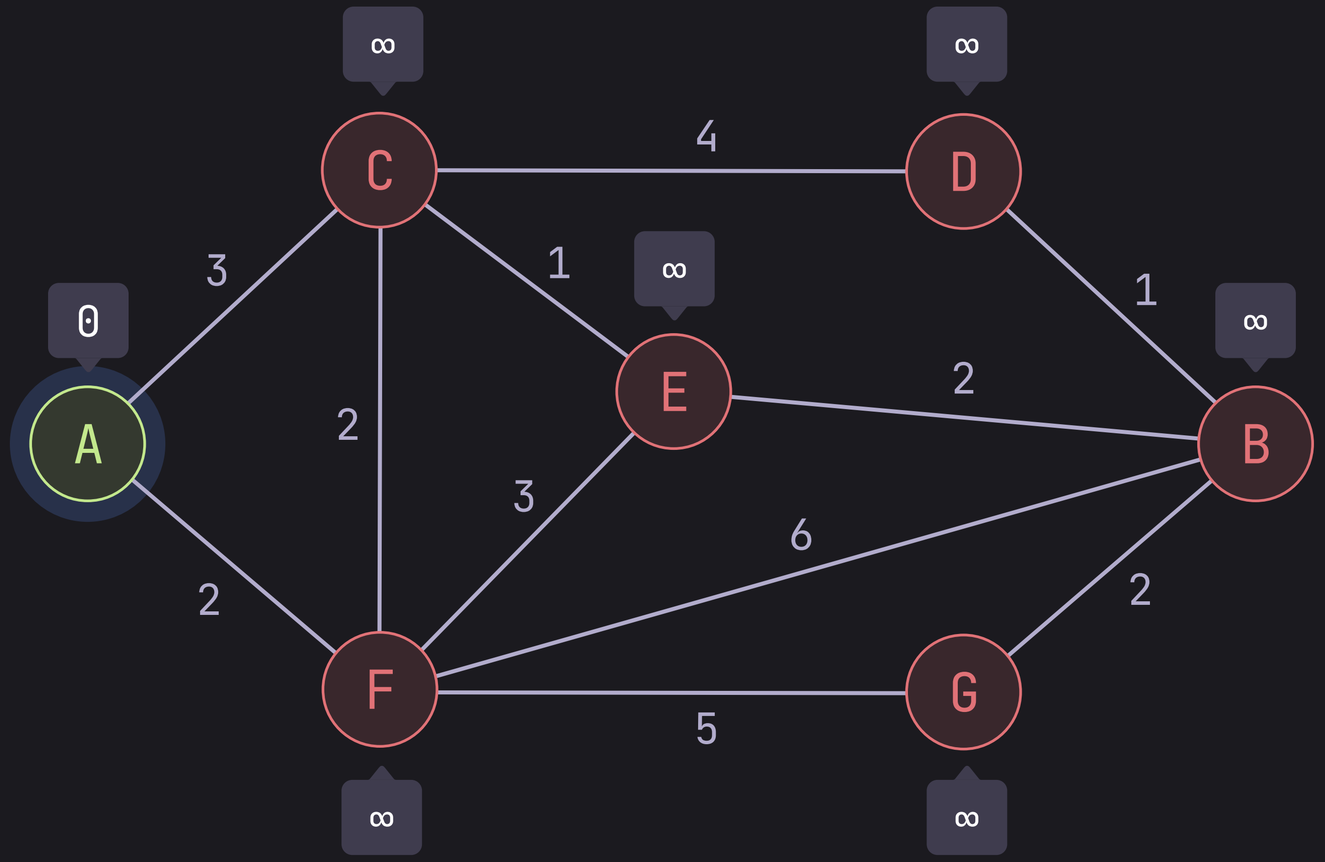
The number above each node represents the shortest distance we have **found so far** from the start to that node

Dijkstra's Algorithm



So just like with BFS we have **visited** and **unvisited** nodes

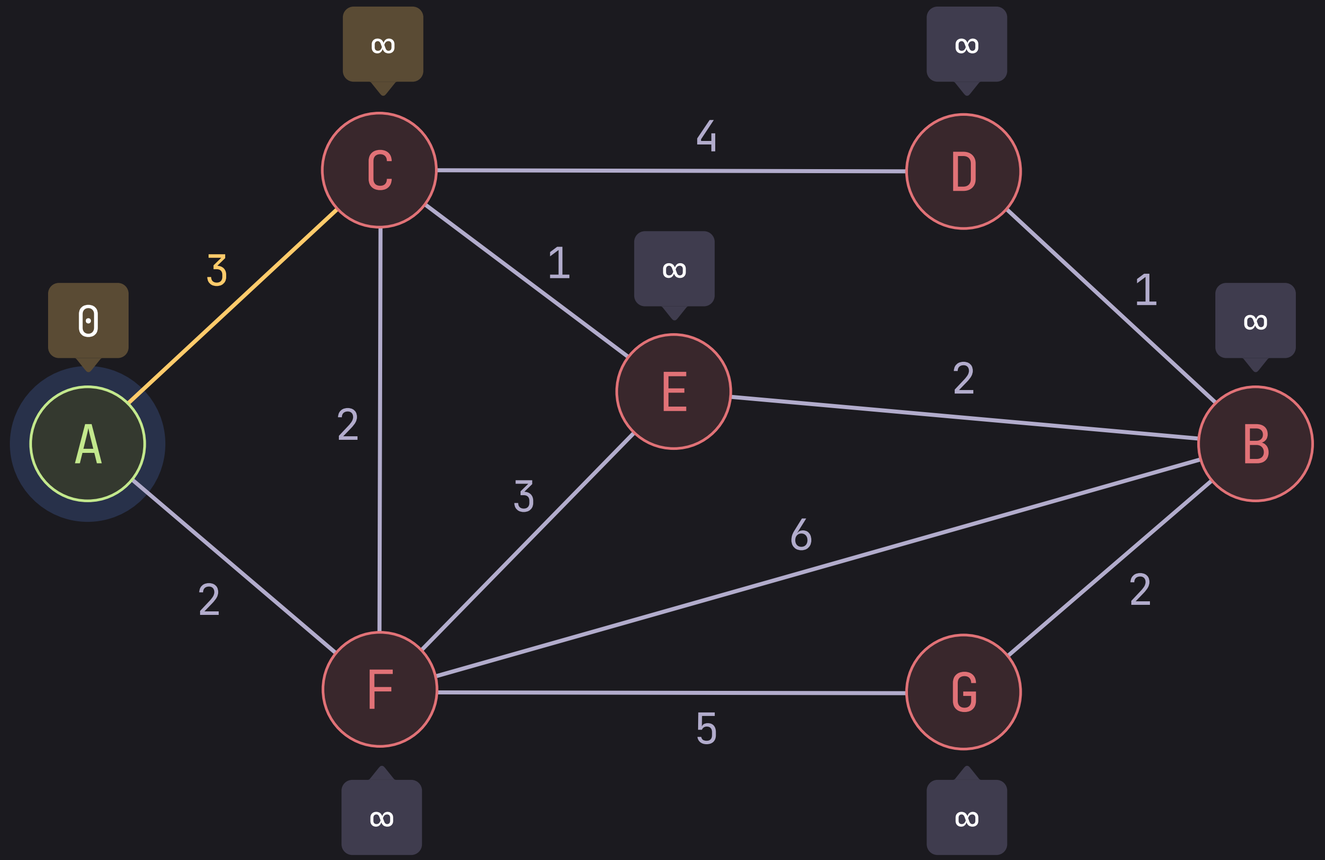
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

So just like with BFS we have **visited** and **unvisited** nodes

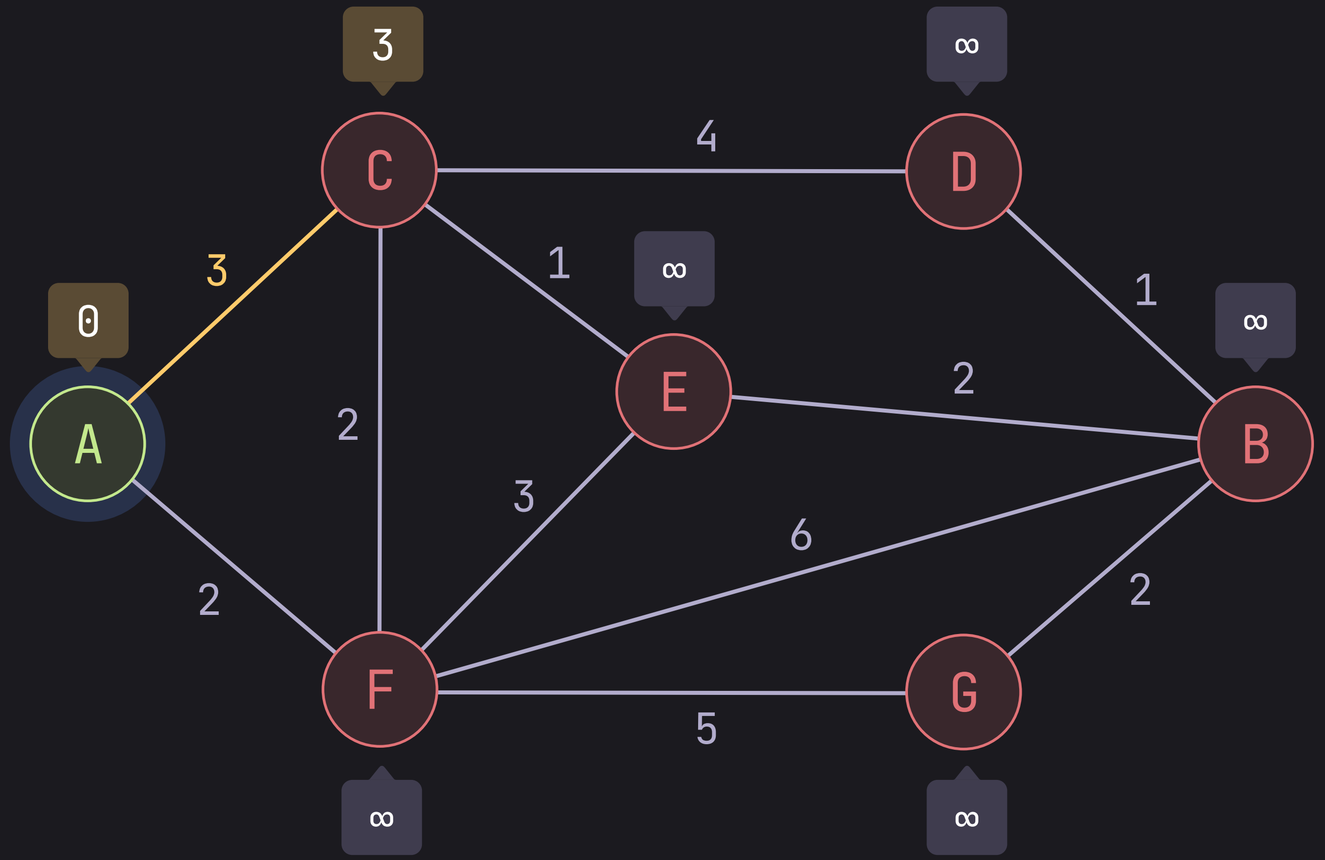
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

We can get to vertex C through A in 0 + 3 = 3 minutes

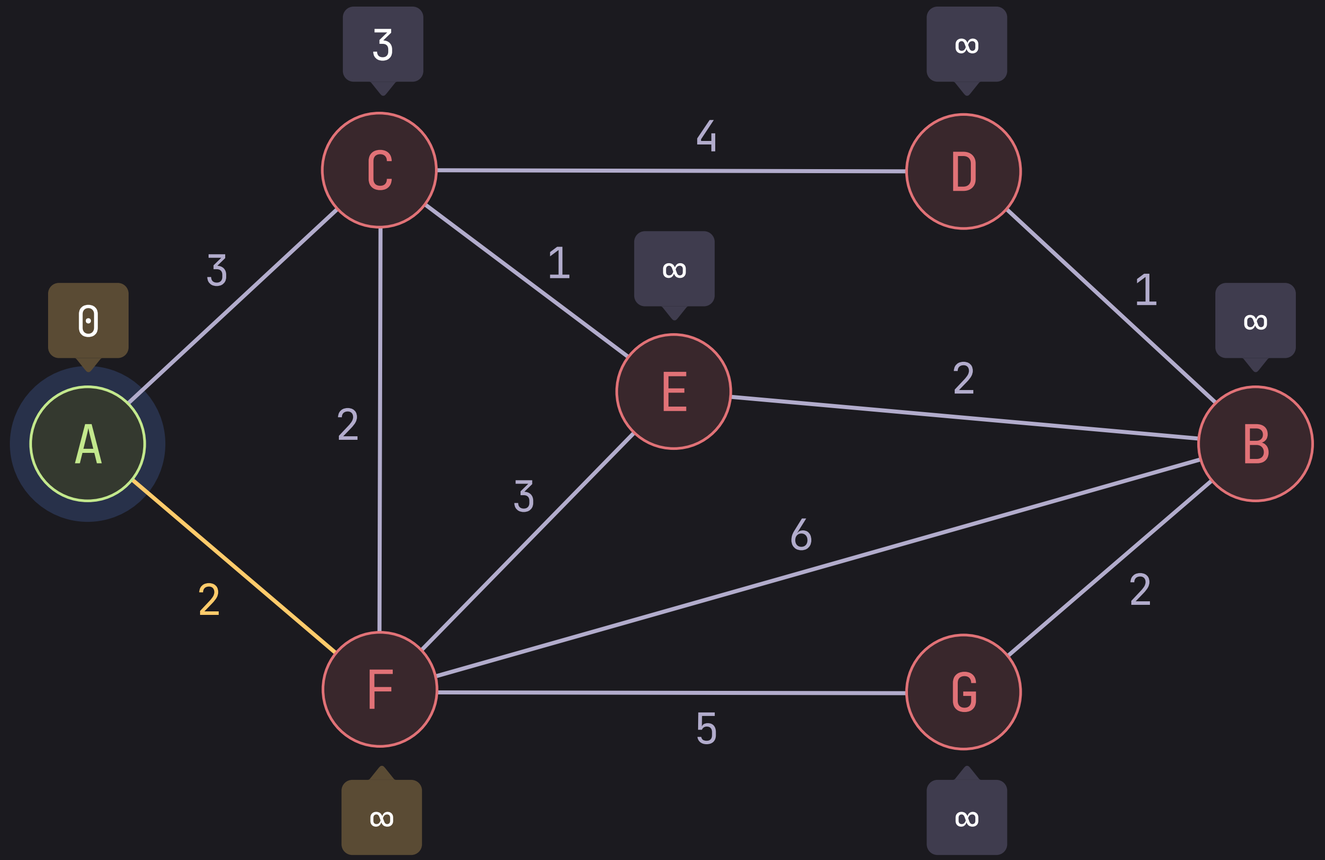
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

As 3 minutes is better than ∞ minutes, we update the estimate

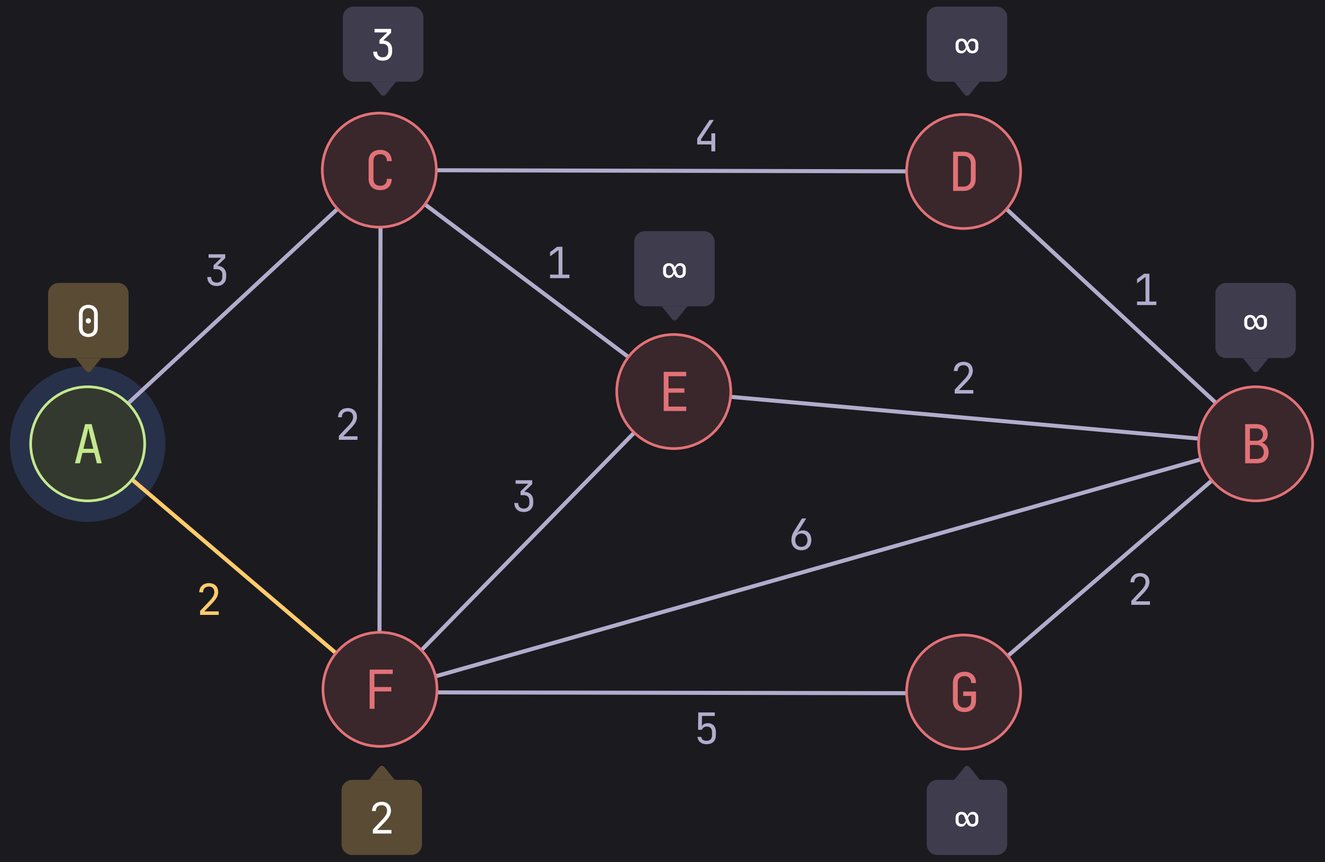
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

We can get to vertex F through A in 0 + 2 = 2 minutes

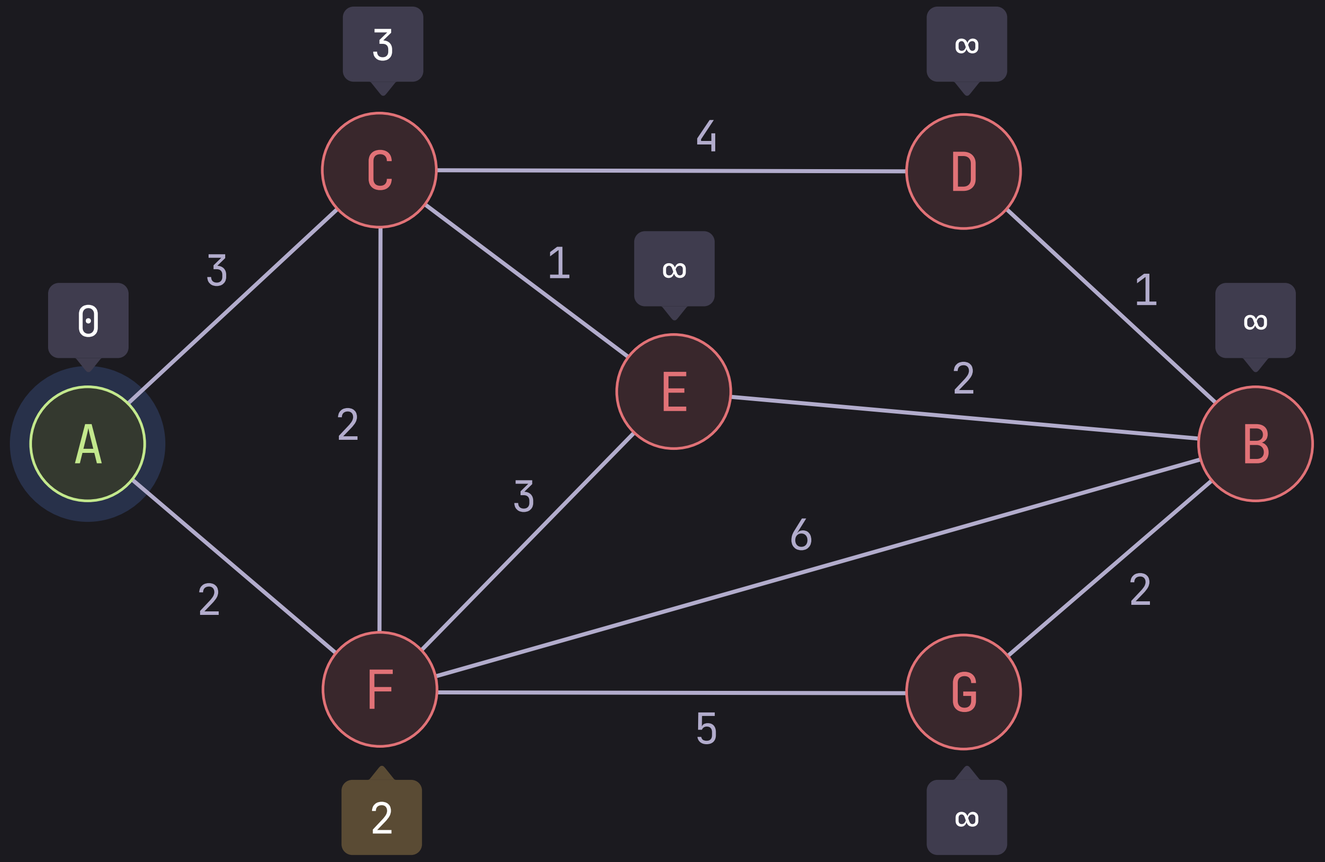
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

As 2 minutes is better than ∞ minutes, we update the estimate

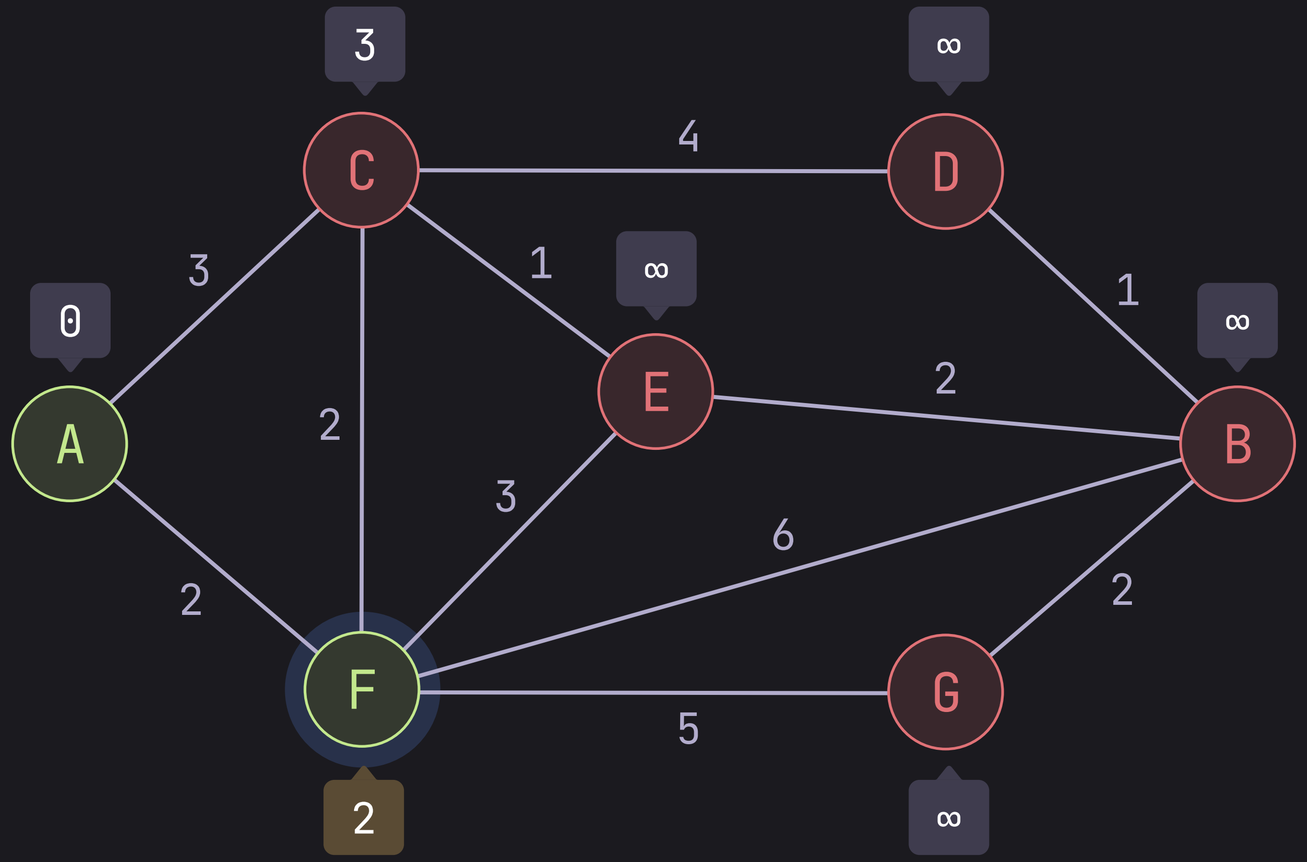
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

Vertex F has the shortest estimate so we will visit F

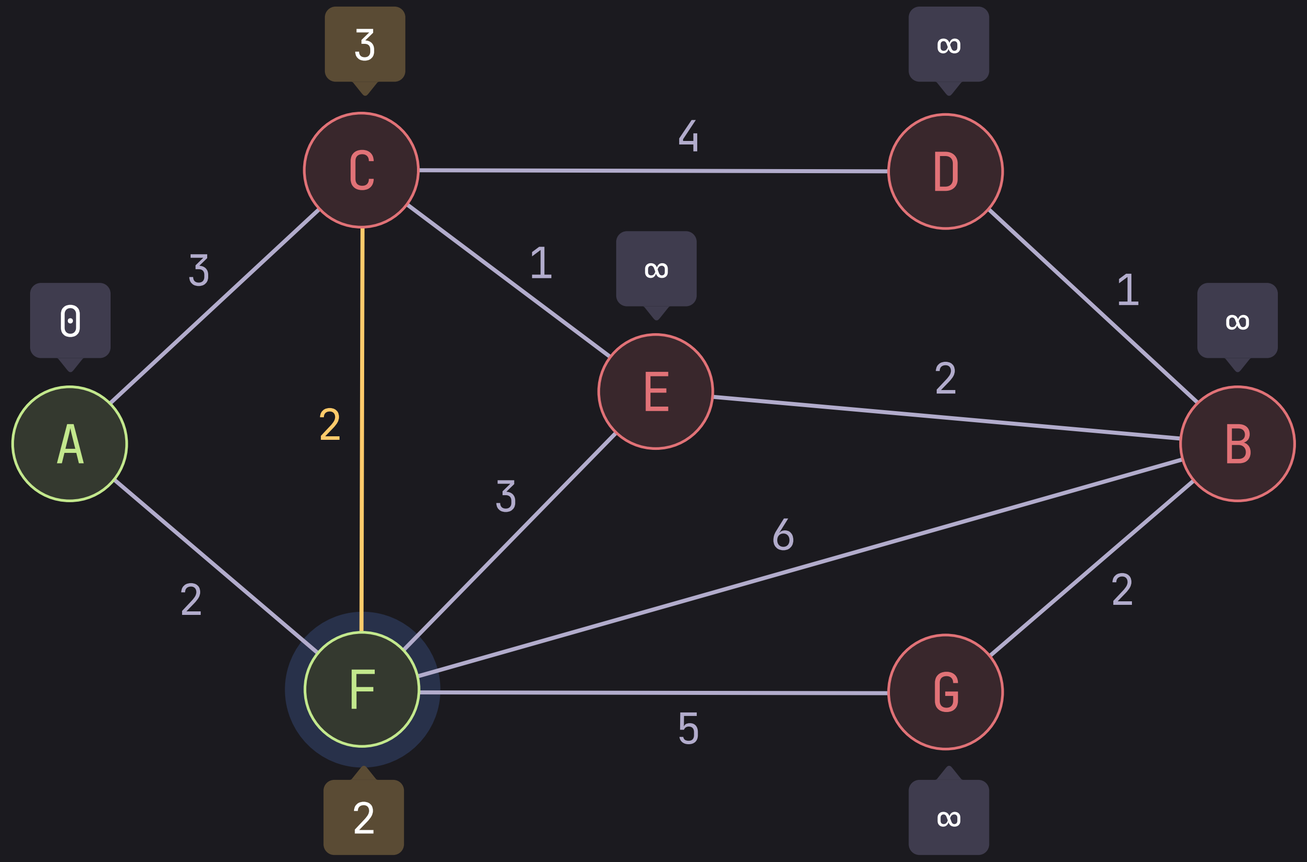
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

Vertex F has the shortest estimate so we will visit F

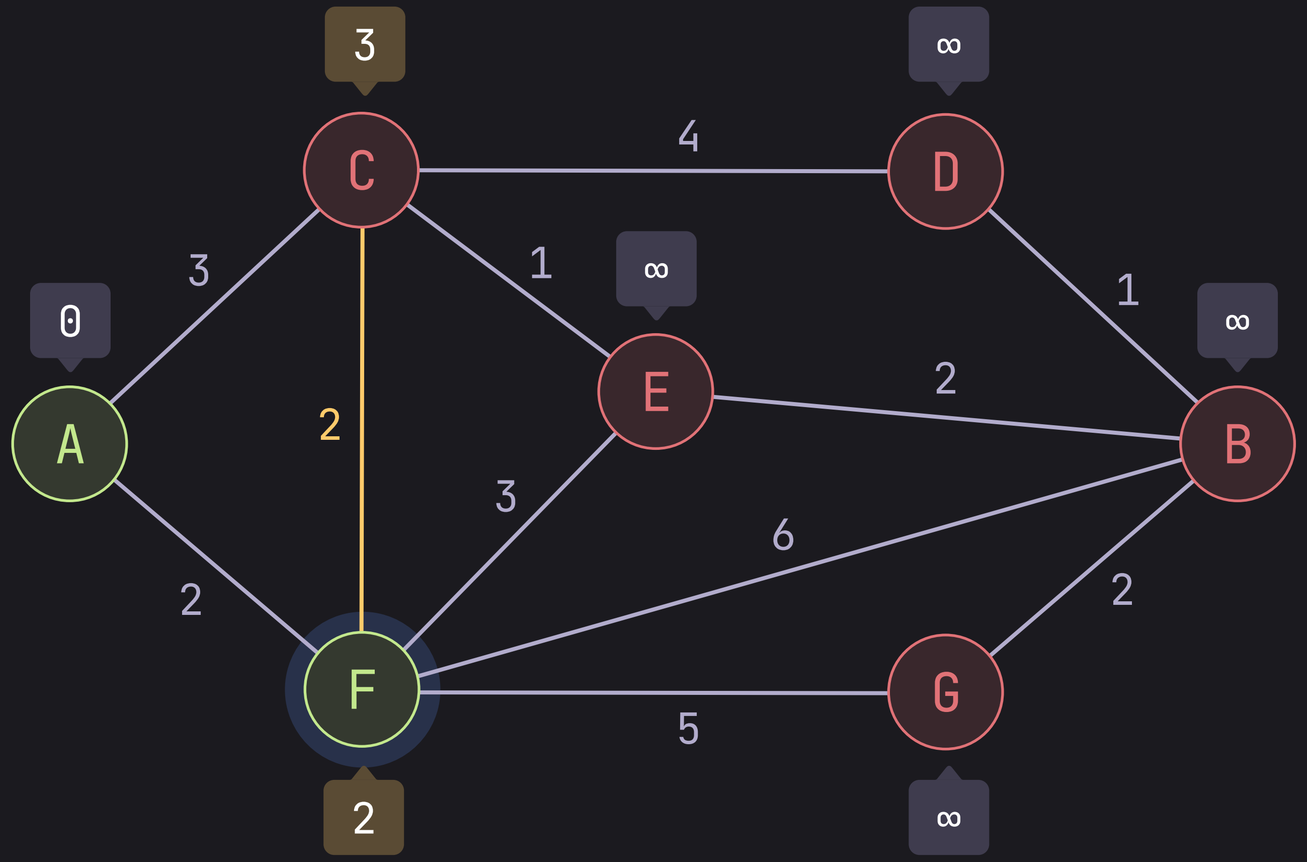
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

We can get to vertex C through F in 2 + 2 = 4 minutes

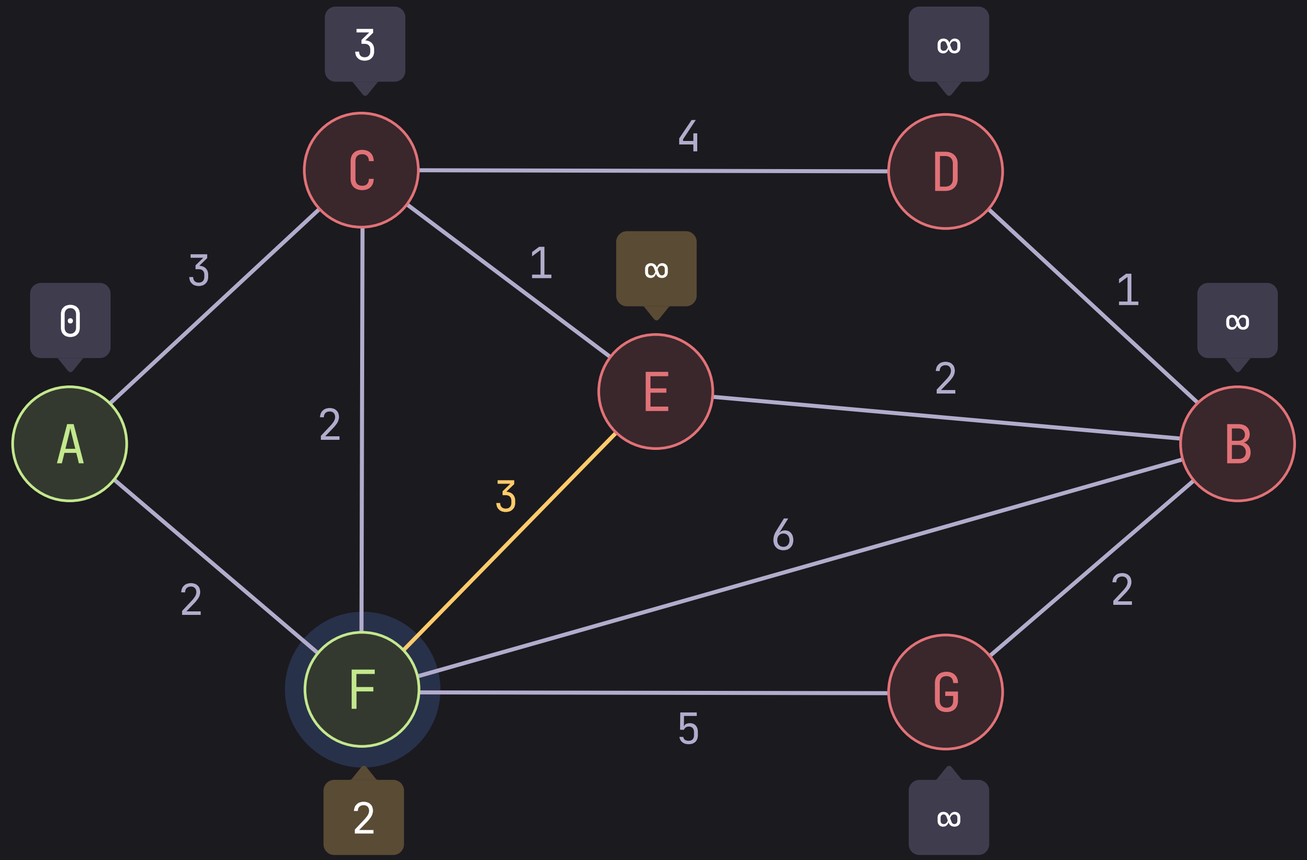
Dijkstra’s Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

As 4 minutes is worse than 3 minutes we don’t update the estimate

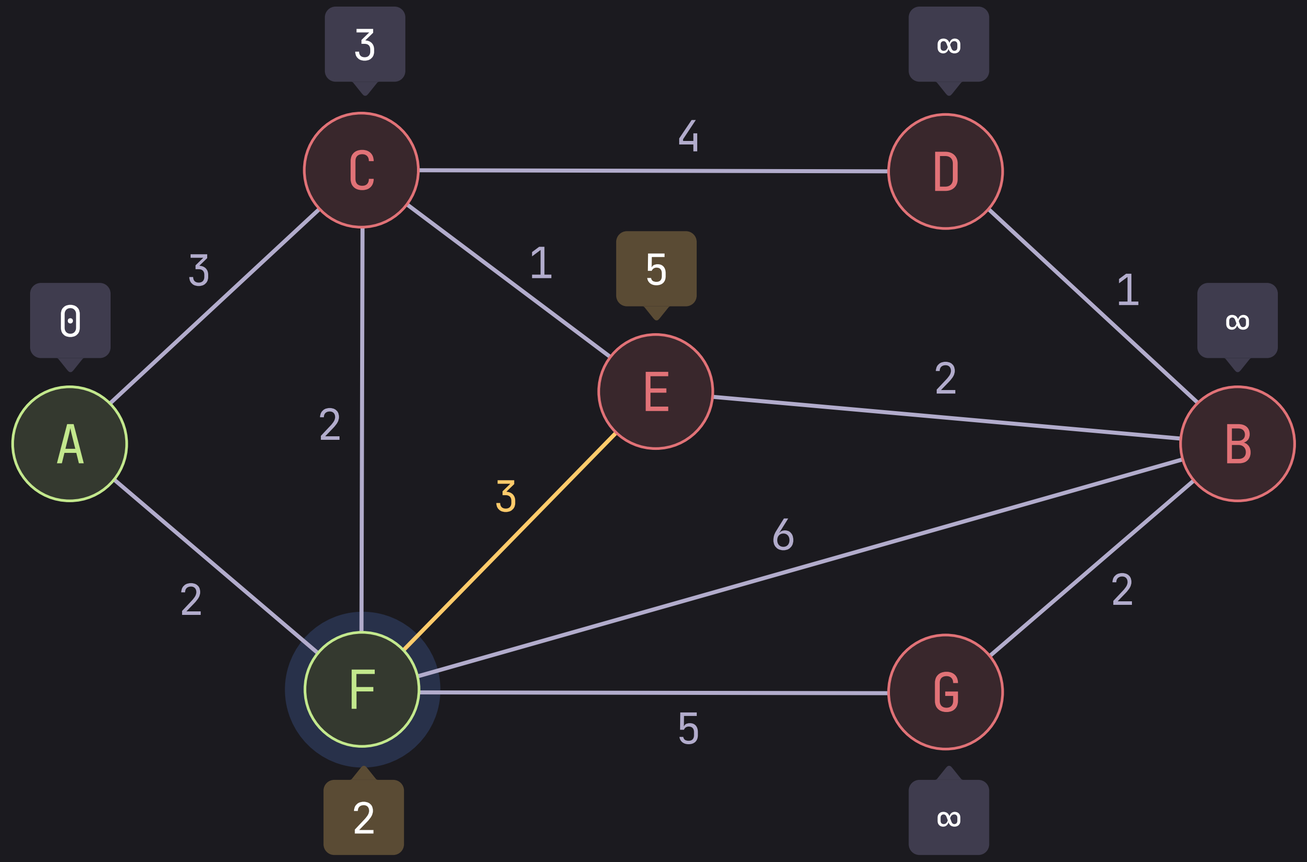
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

We can get to vertex E through F in 2 + 3 = 5 minutes

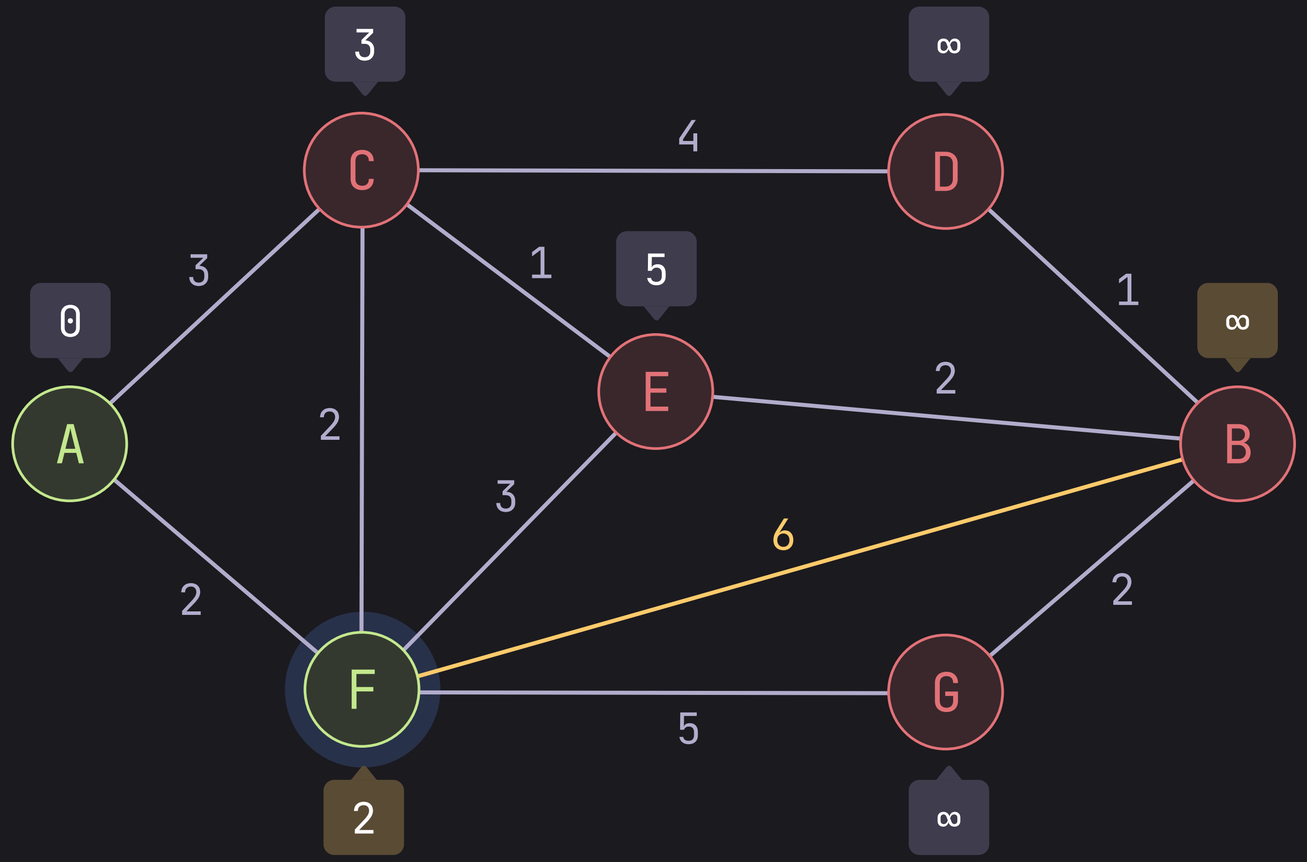
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

As 5 minutes is better than ∞ minutes we update the estimate

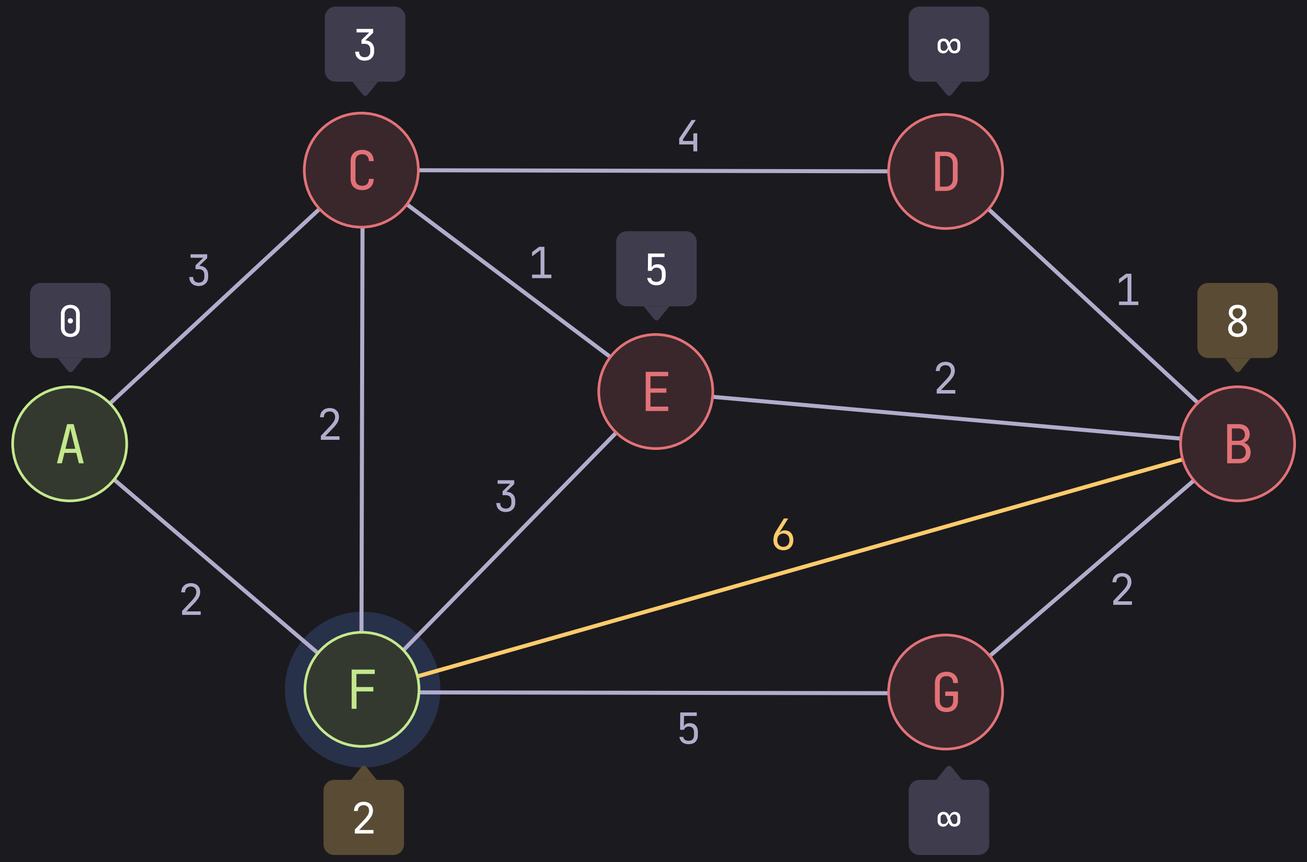
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

We can get to vertex B through F in 2 + 6 = 8 minutes

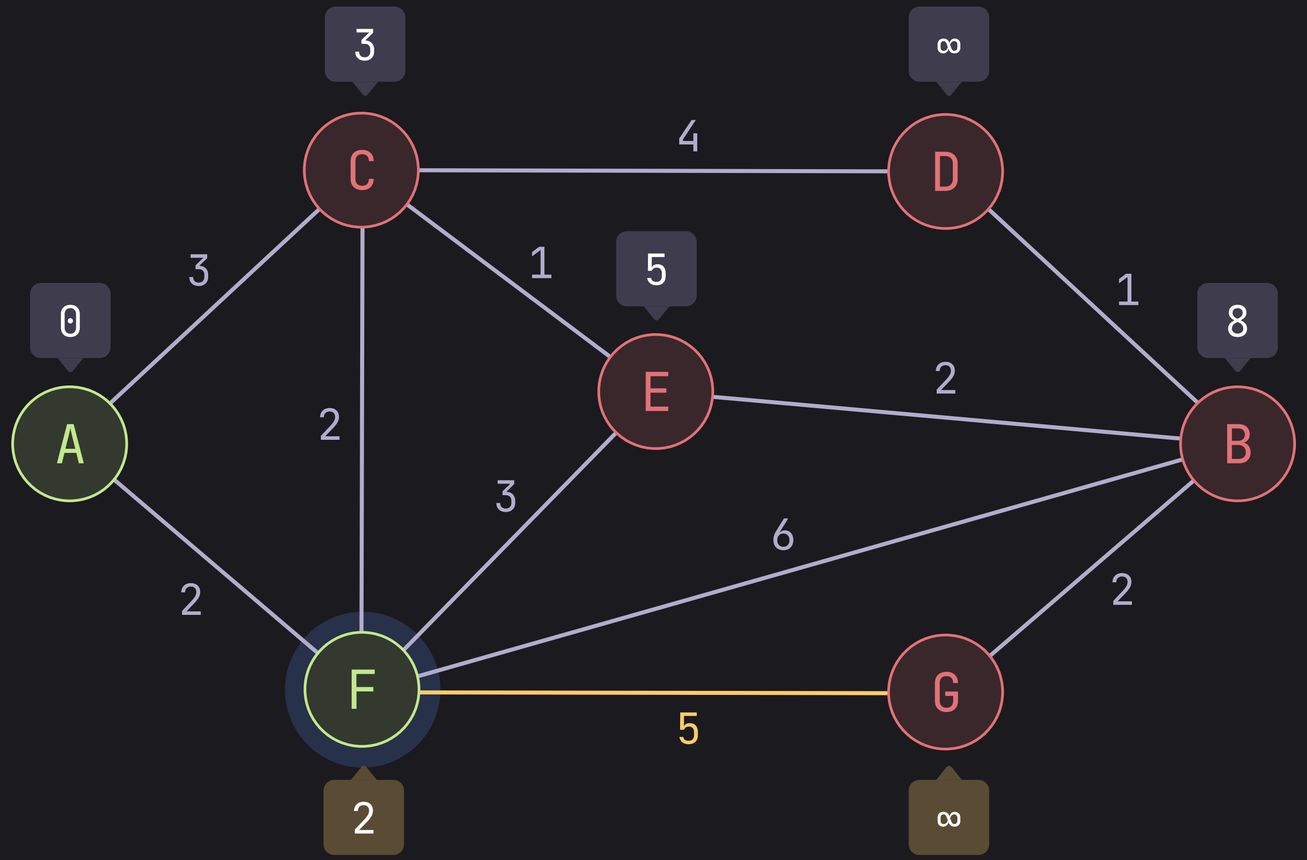
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

As 8 minutes is better than ∞ minutes we update the estimate

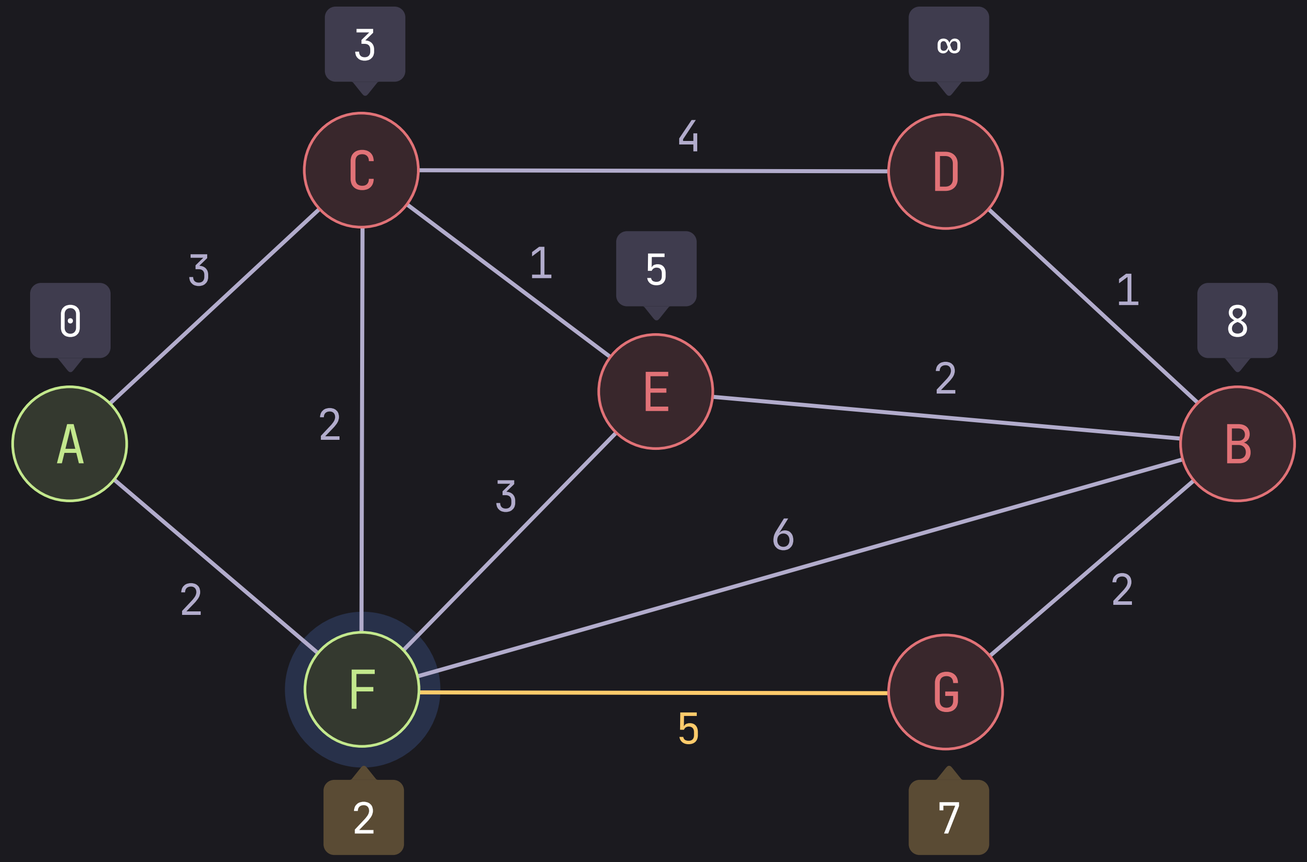
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

We can get to vertex G through F in 2 + 5 = 7 minutes

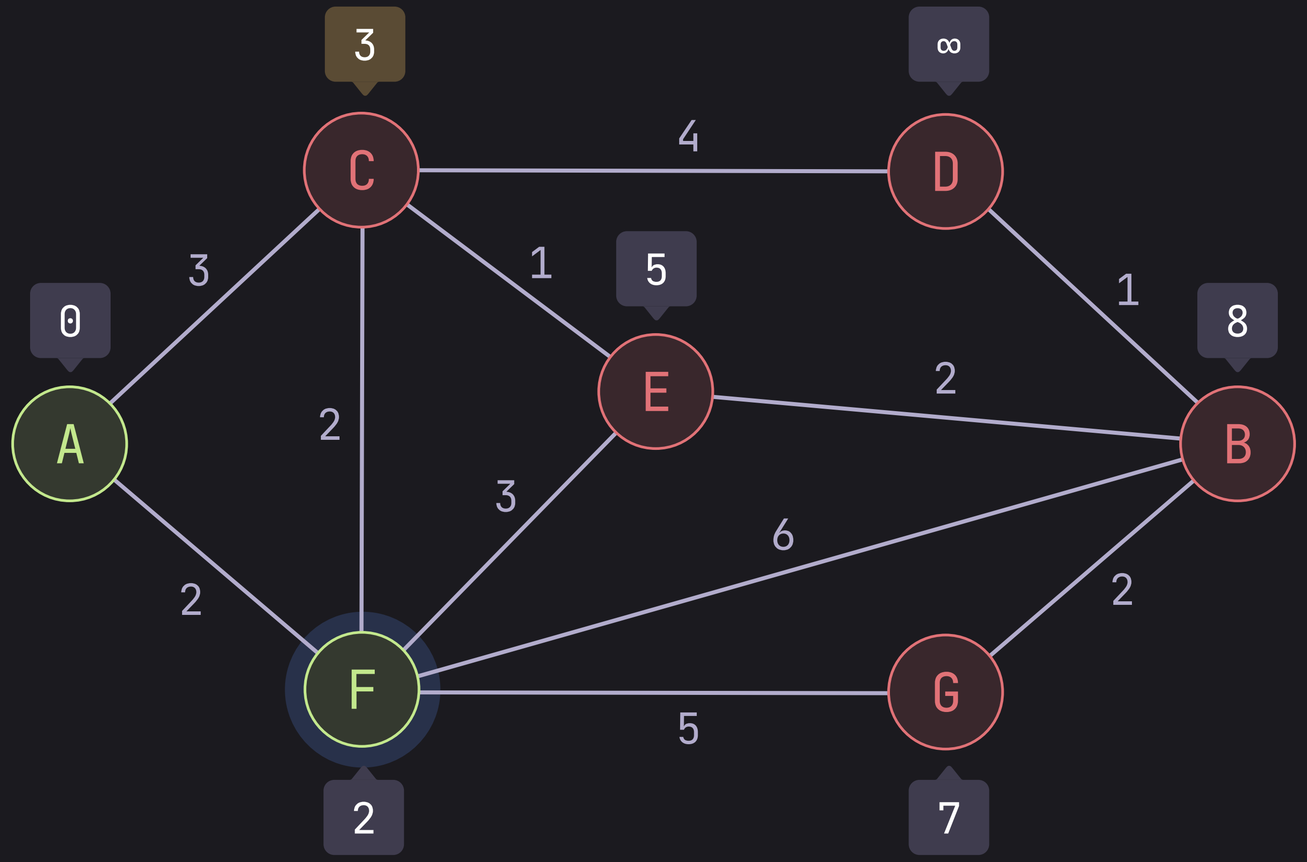
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

As 7 minutes is better than ∞ minutes we update the estimate

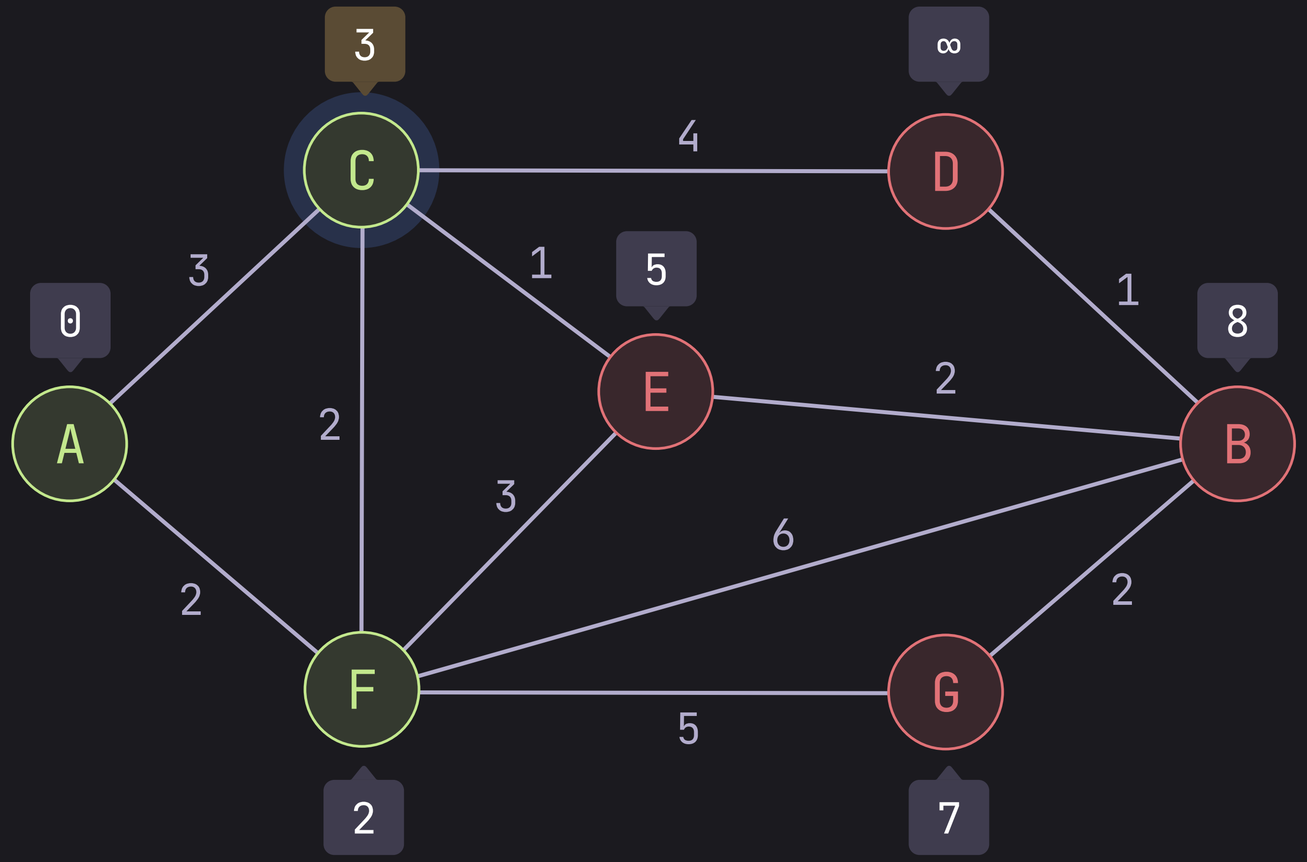
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

Vertex C has the shortest estimate so we will visit C

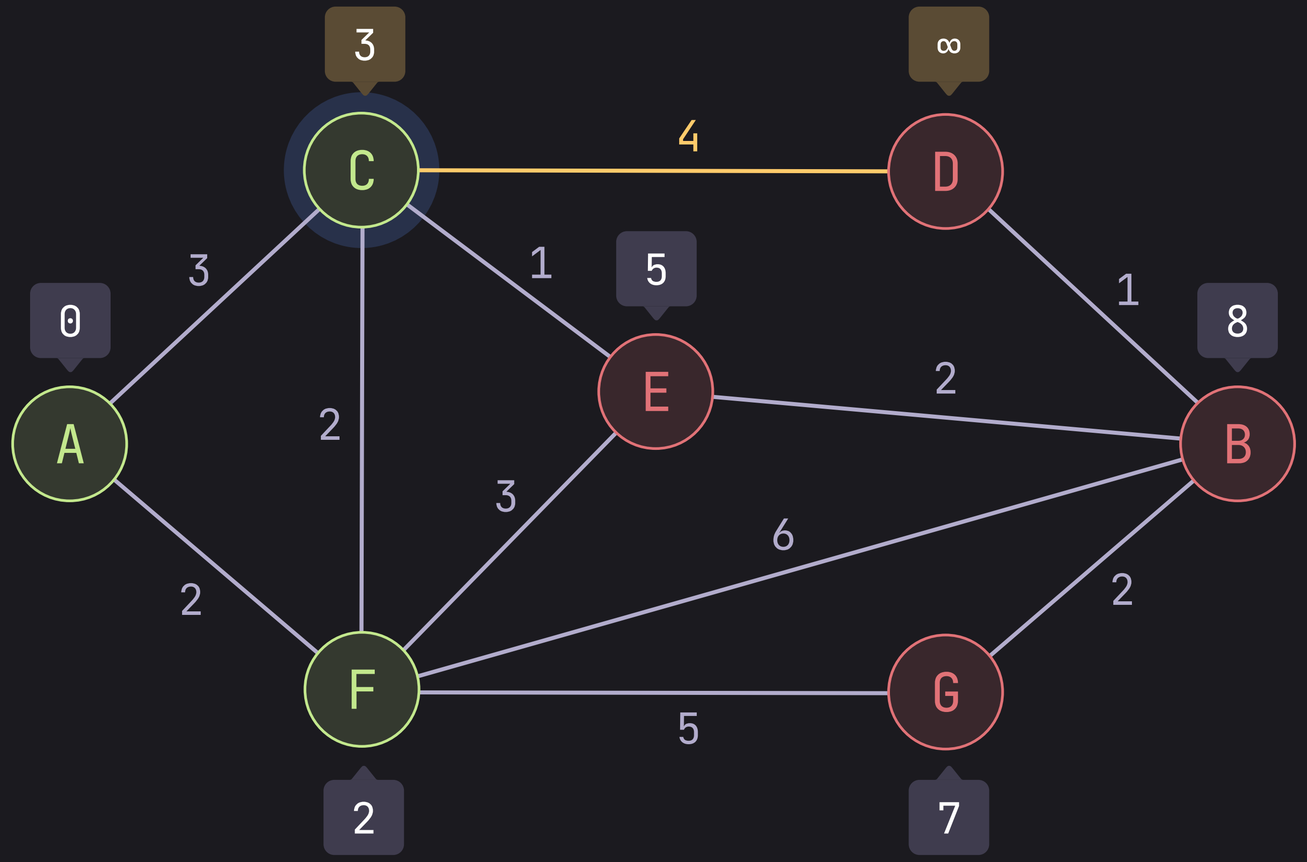
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

Vertex C has the shortest estimate so we will visit C

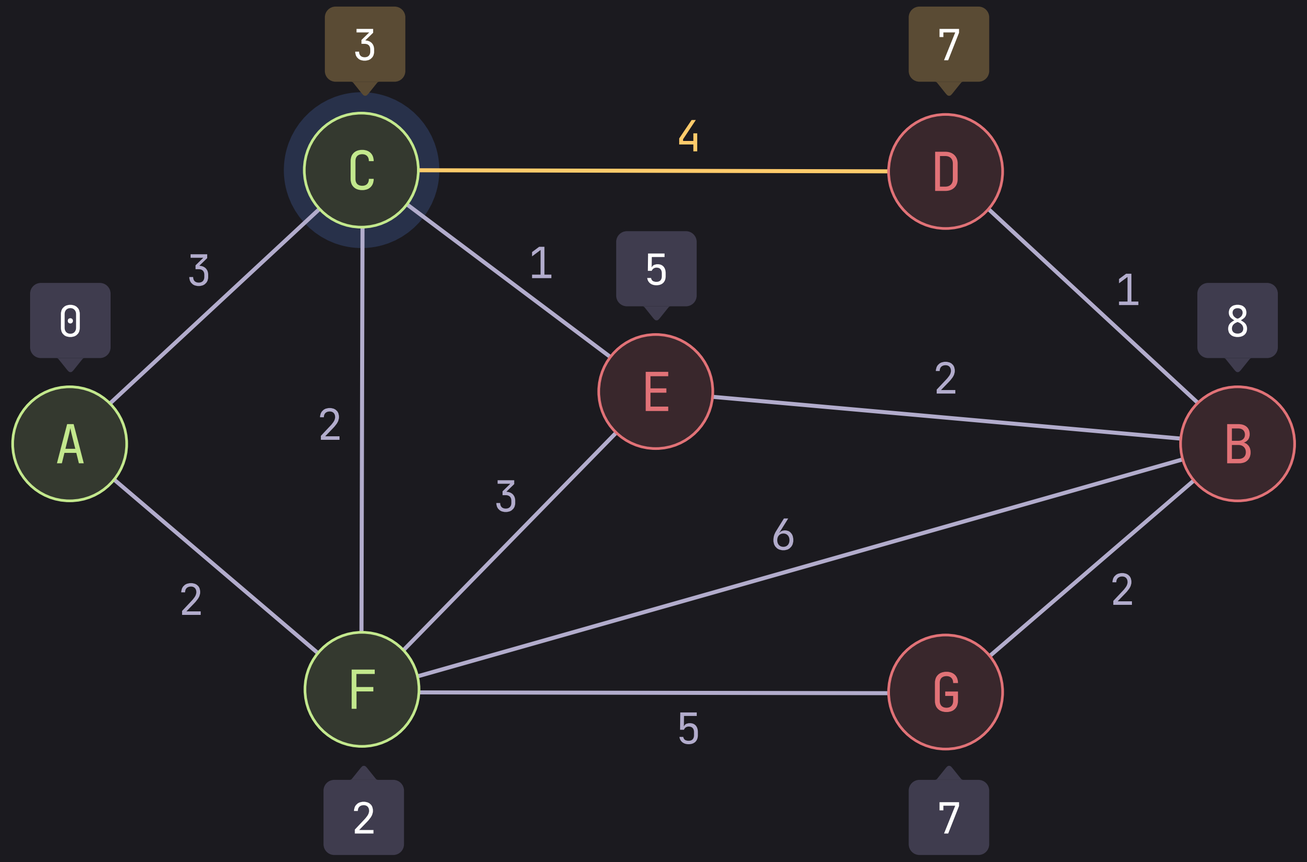
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

We can get to vertex D through C in 3 + 4 = 7 minutes

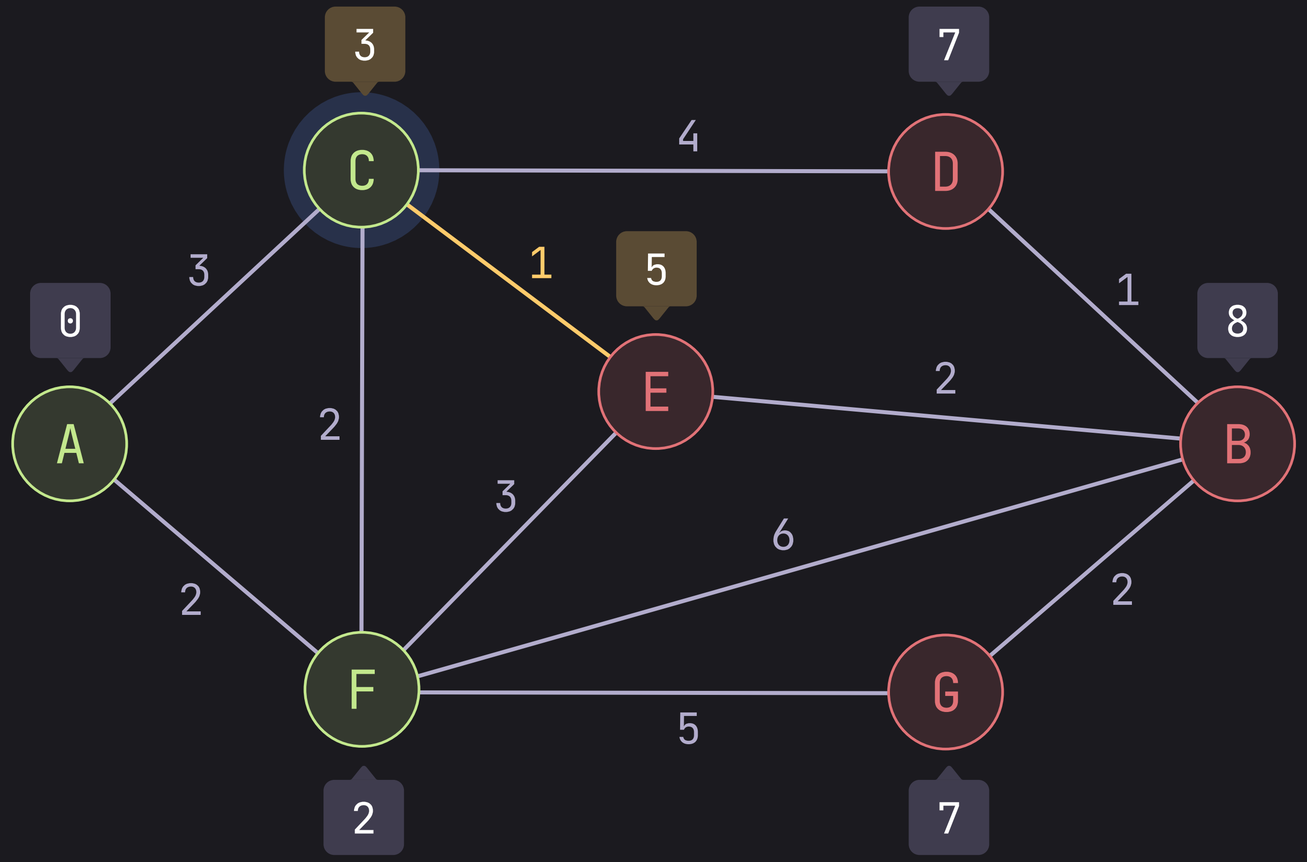
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

As 7 minutes is better than ∞ minutes, we update the estimate

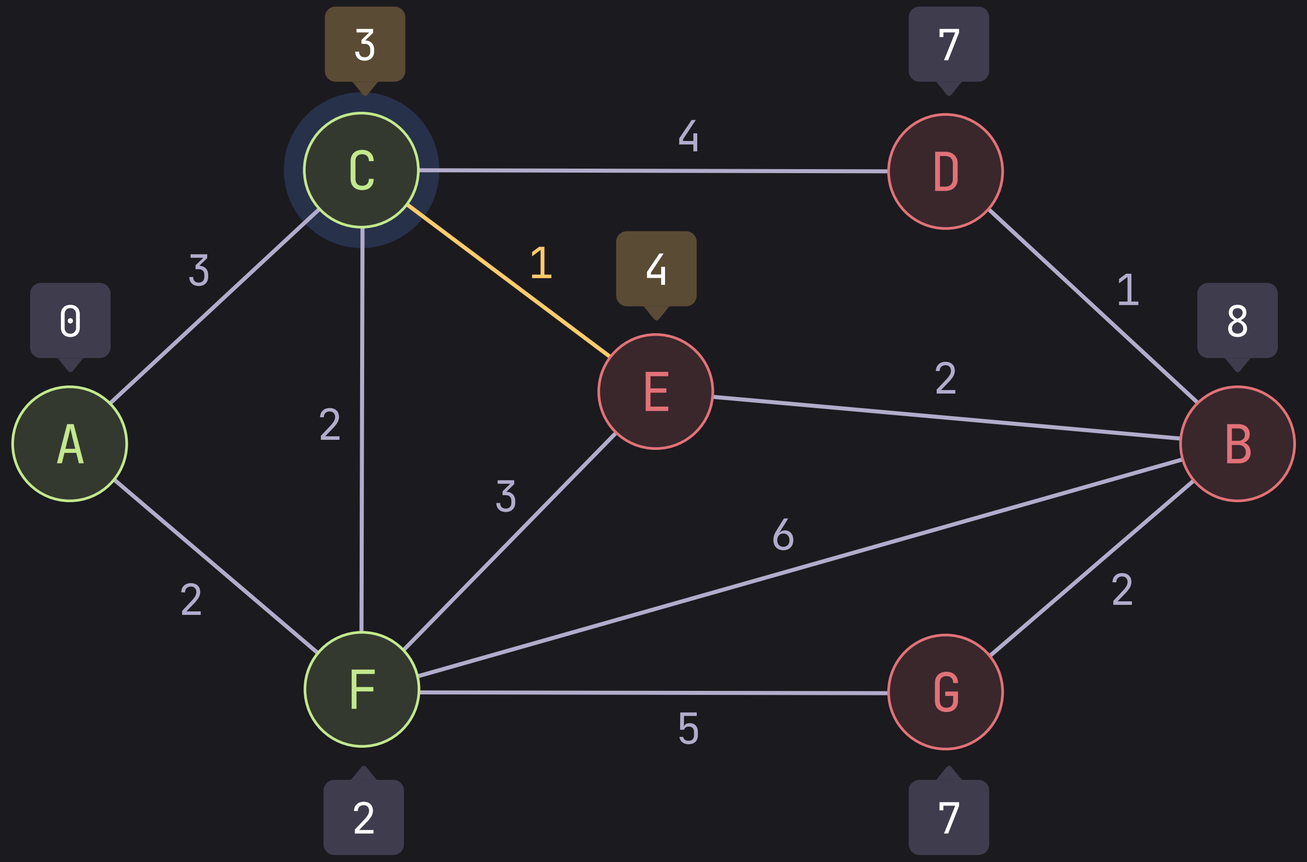
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

We can get to vertex E through C in 3 + 1 = 4 minutes

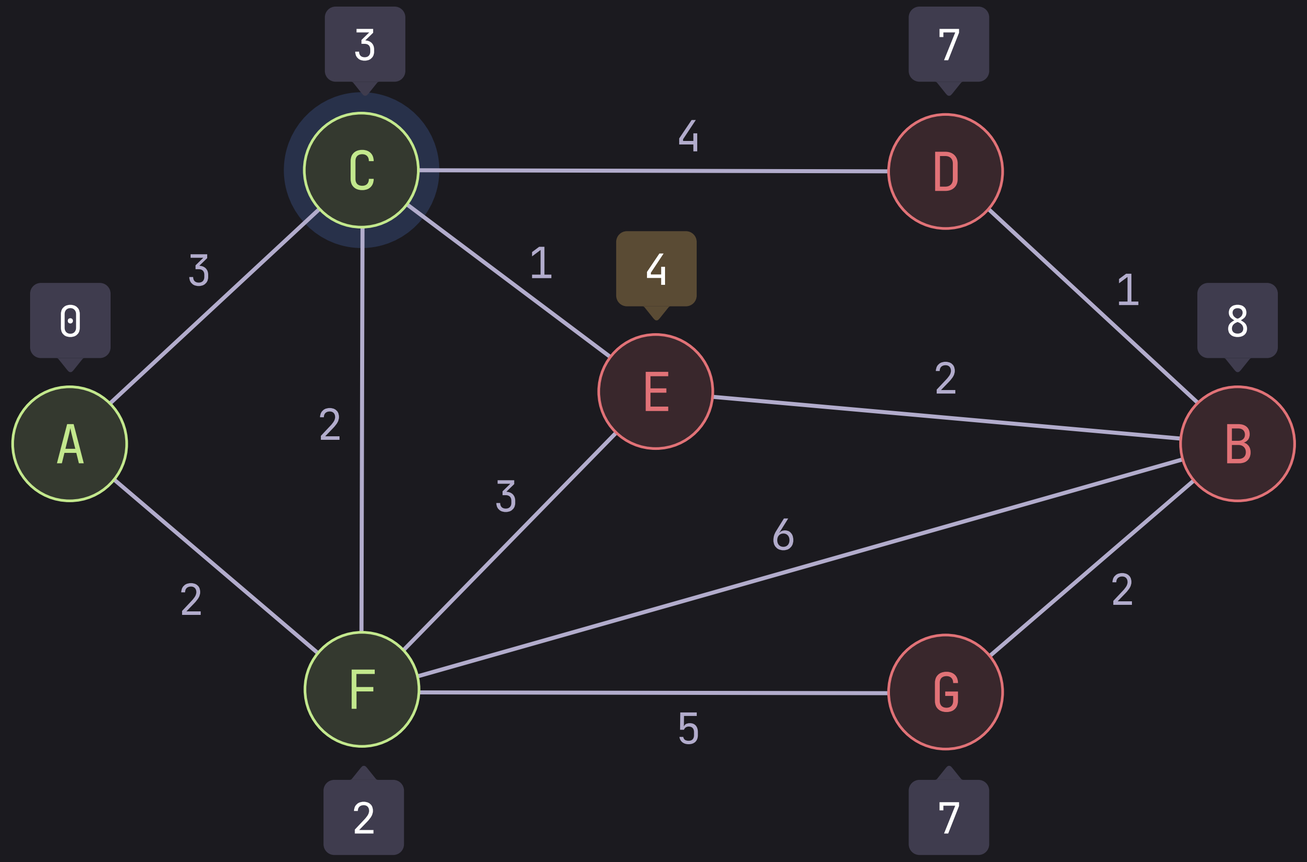
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

As 4 minutes is better than 5 minutes, we update the estimate

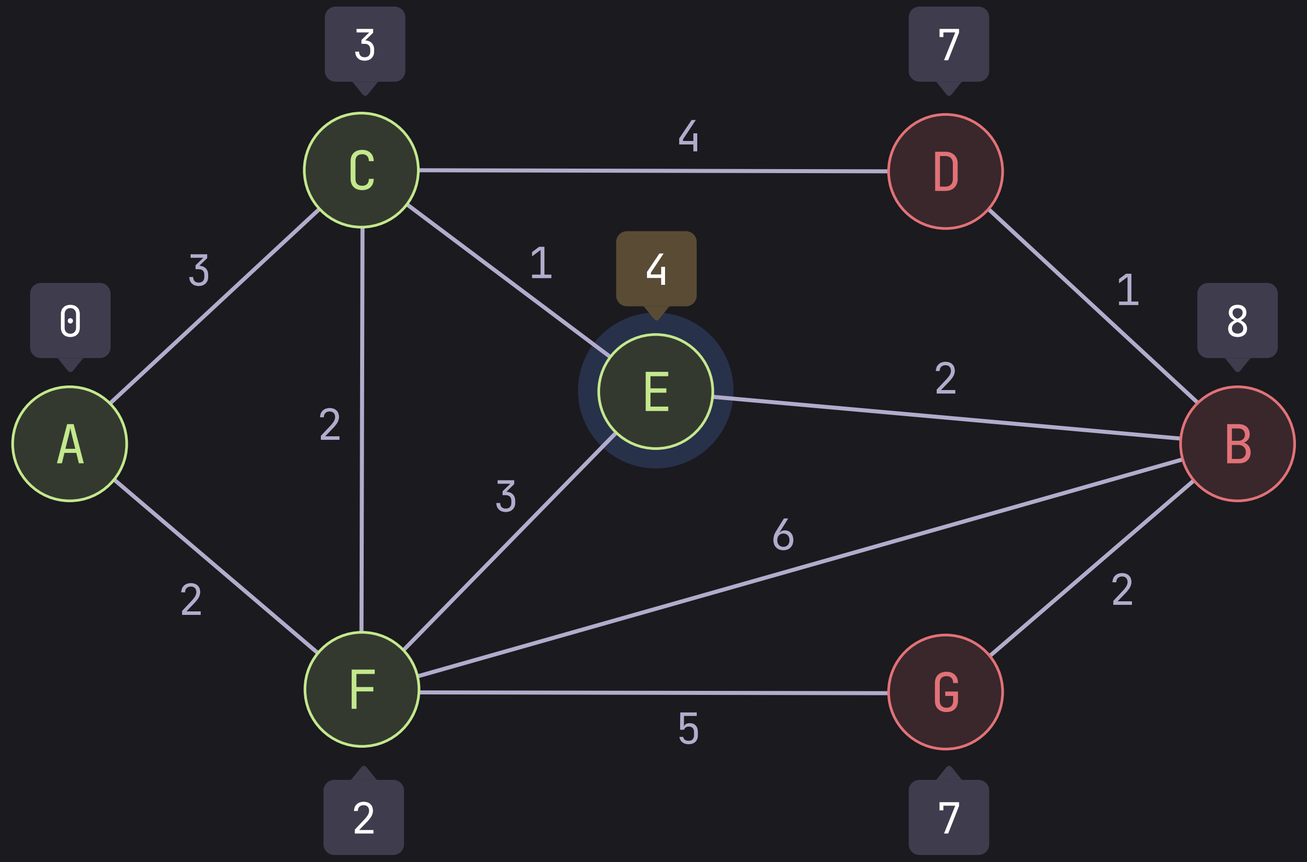
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

Vertex E has the shortest estimate so we will visit E

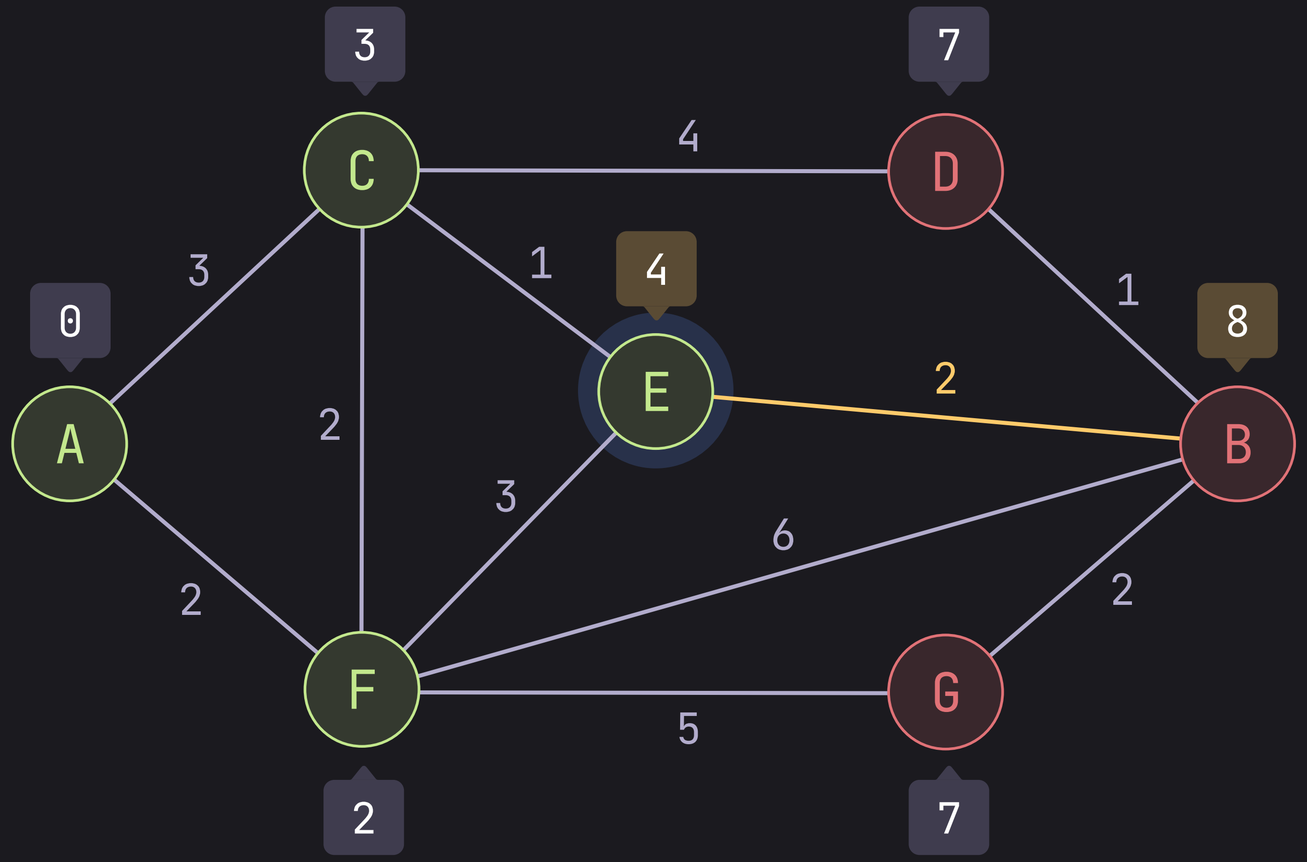
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

Vertex E has the shortest estimate so we will visit E

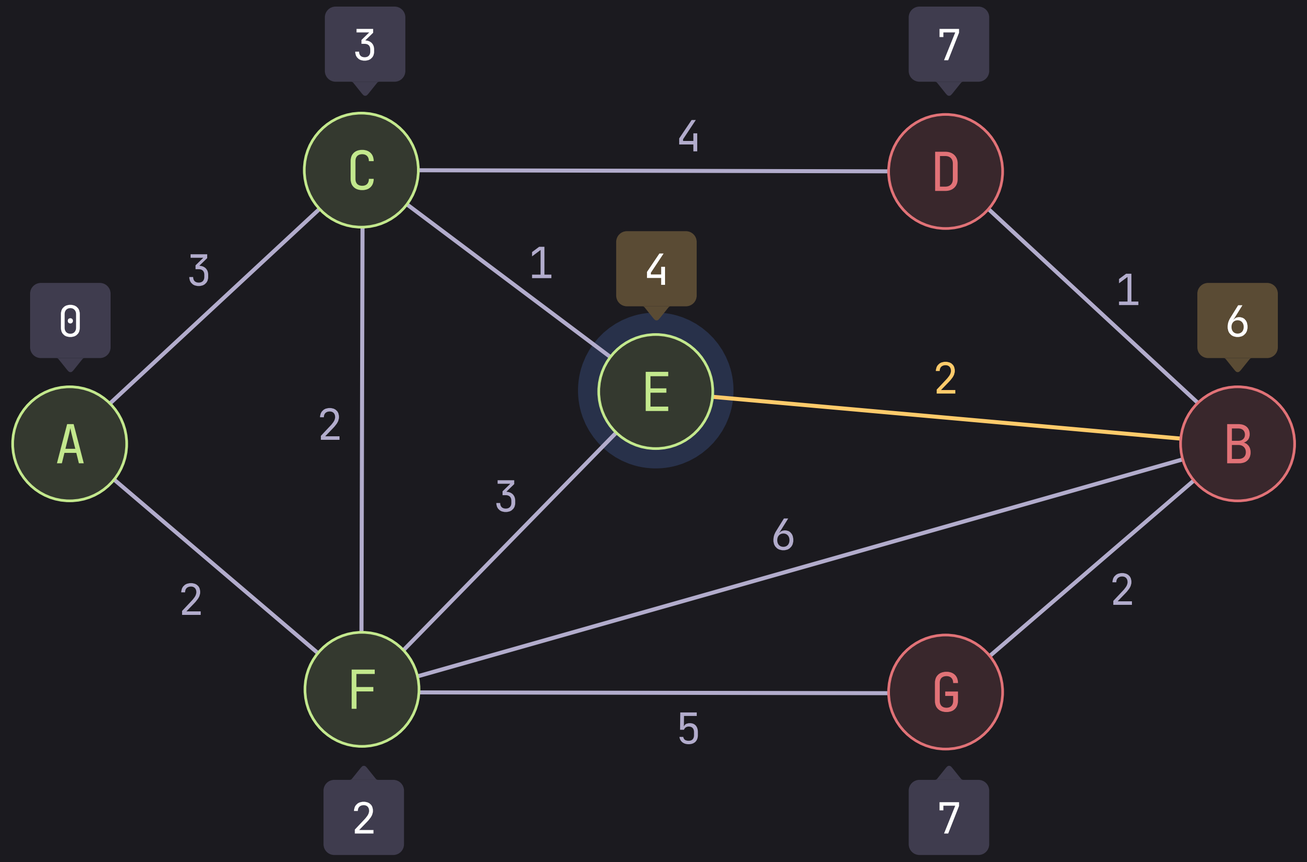
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

We can get to vertex B through E in 4 + 2 = 6 minutes

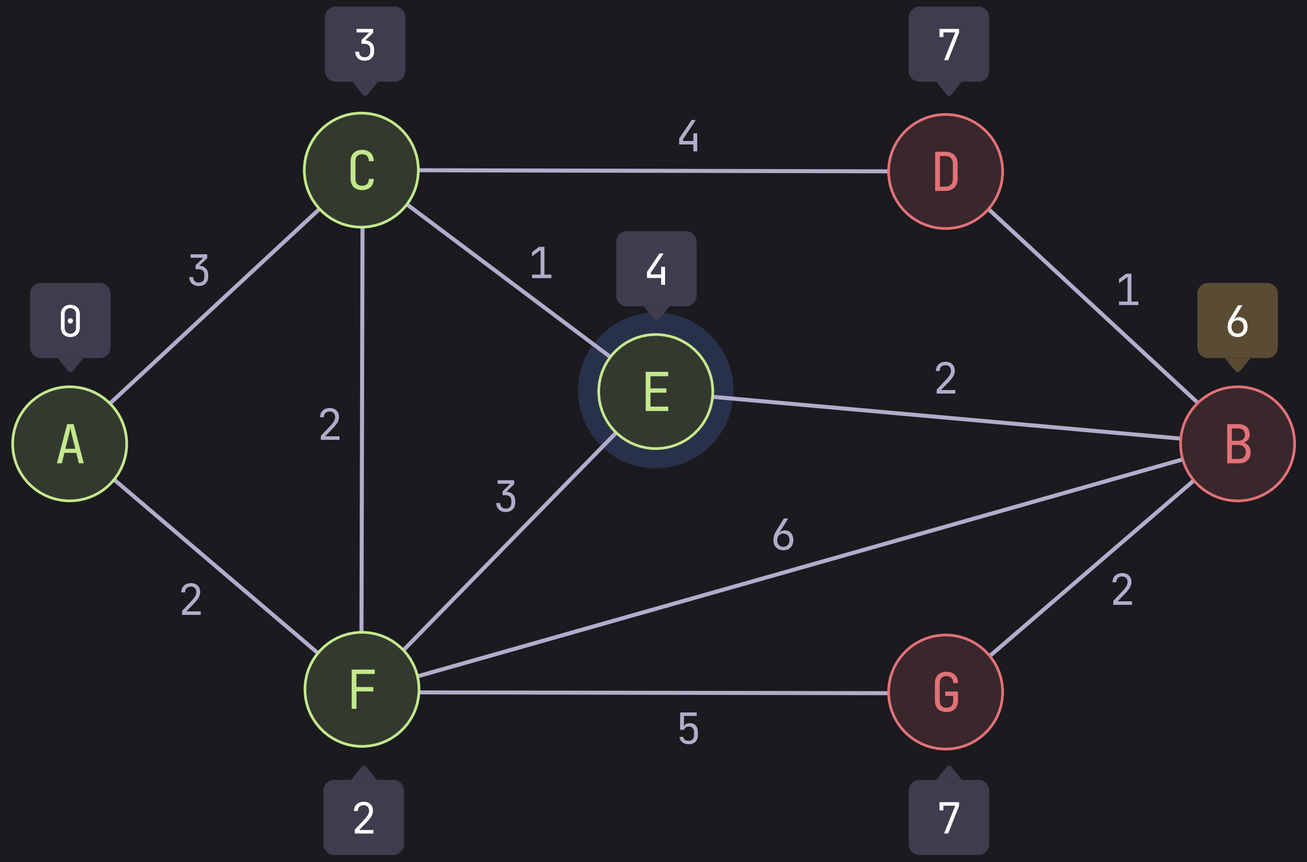
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

As 6 minutes is better than 8 minutes, we update the estimate

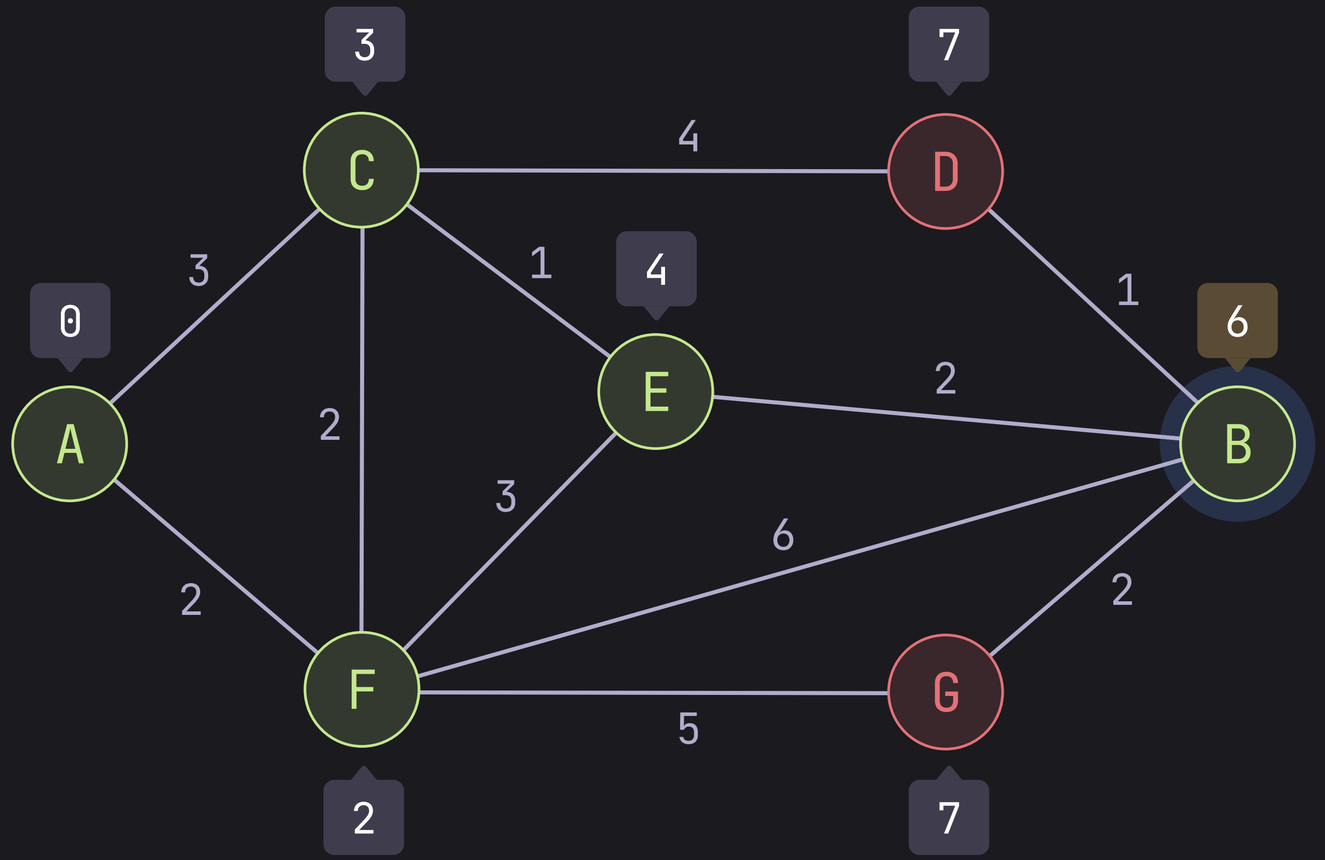
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

Vertex B has the shortest estimate so we will visit B

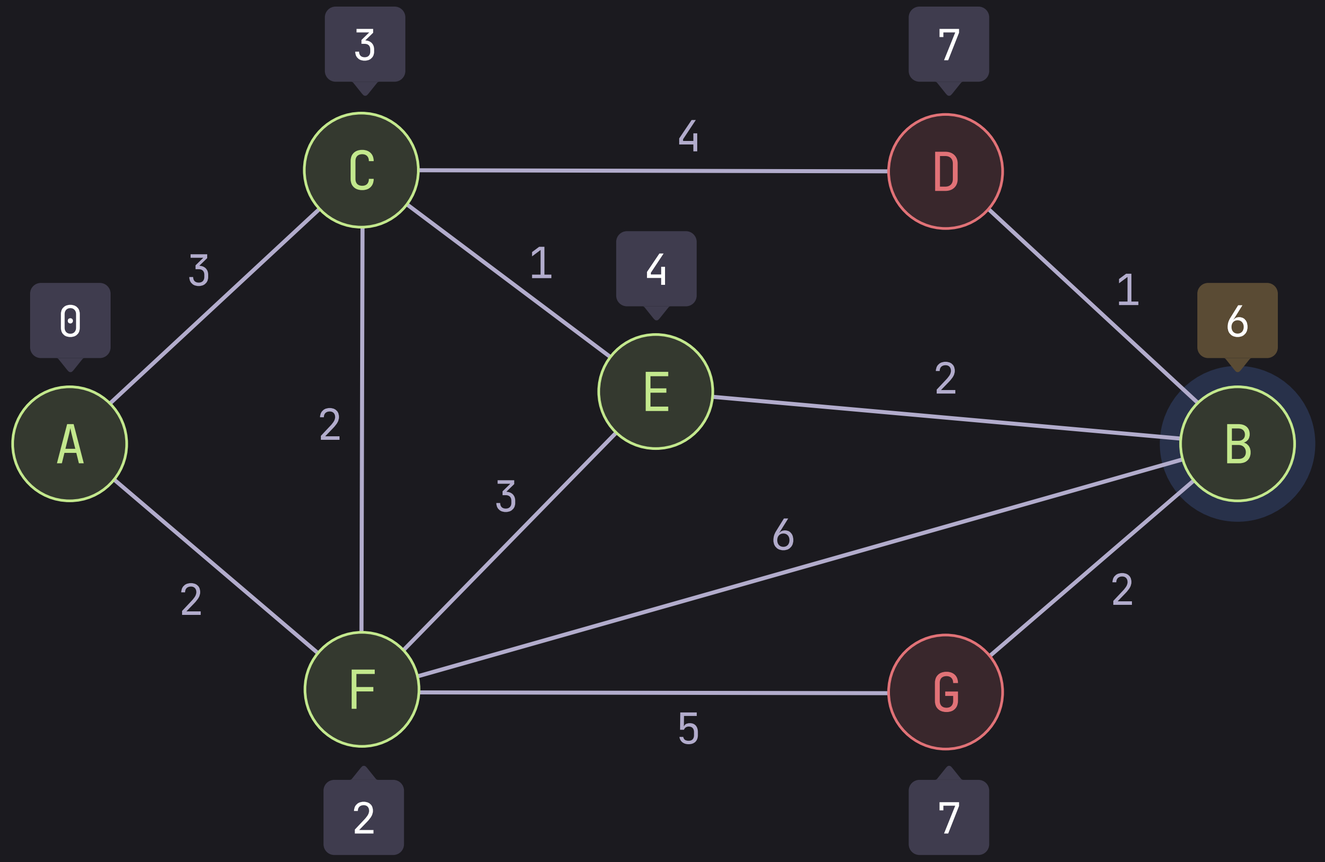
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

Now we have reached the target node, so we are done

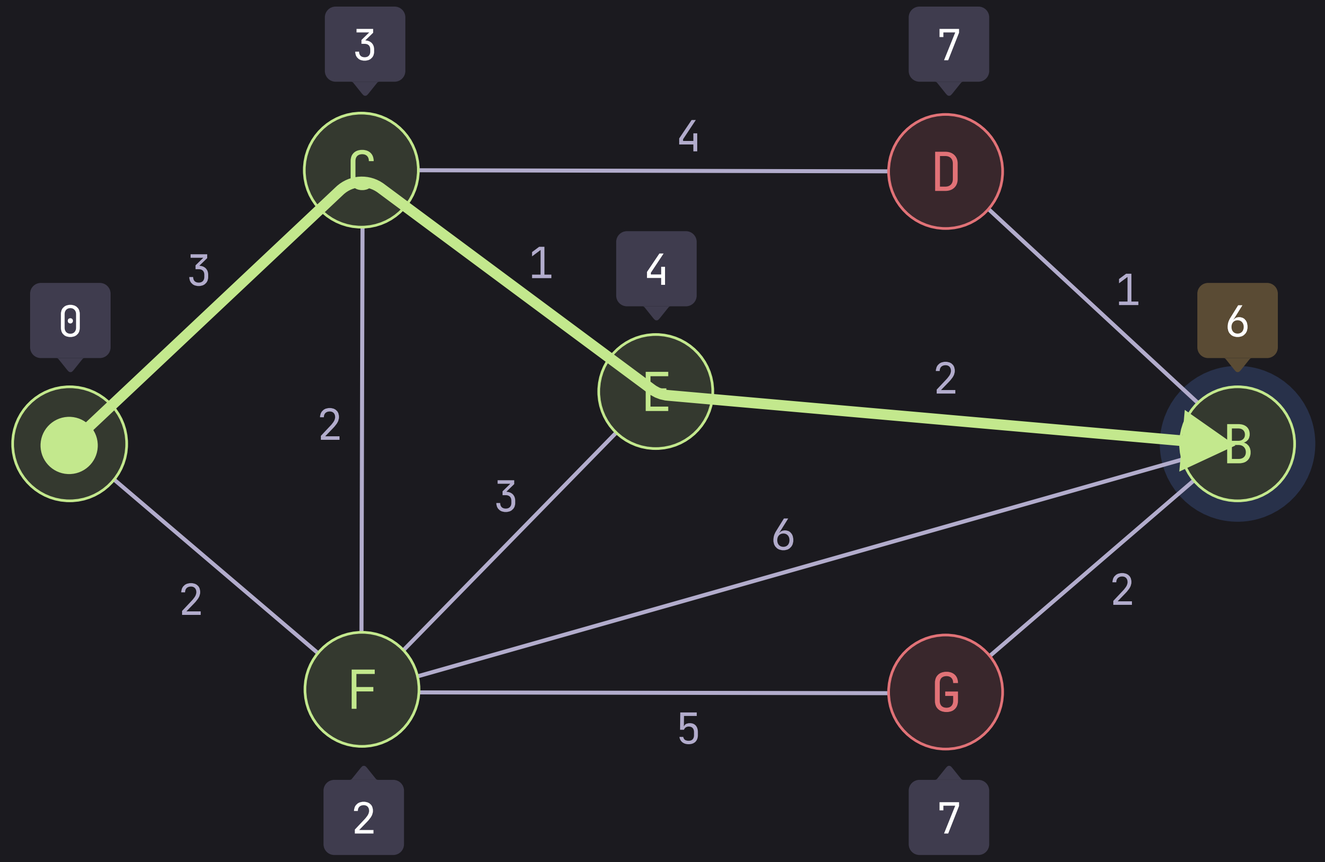
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

This means that we can get from A to B in 6 minutes

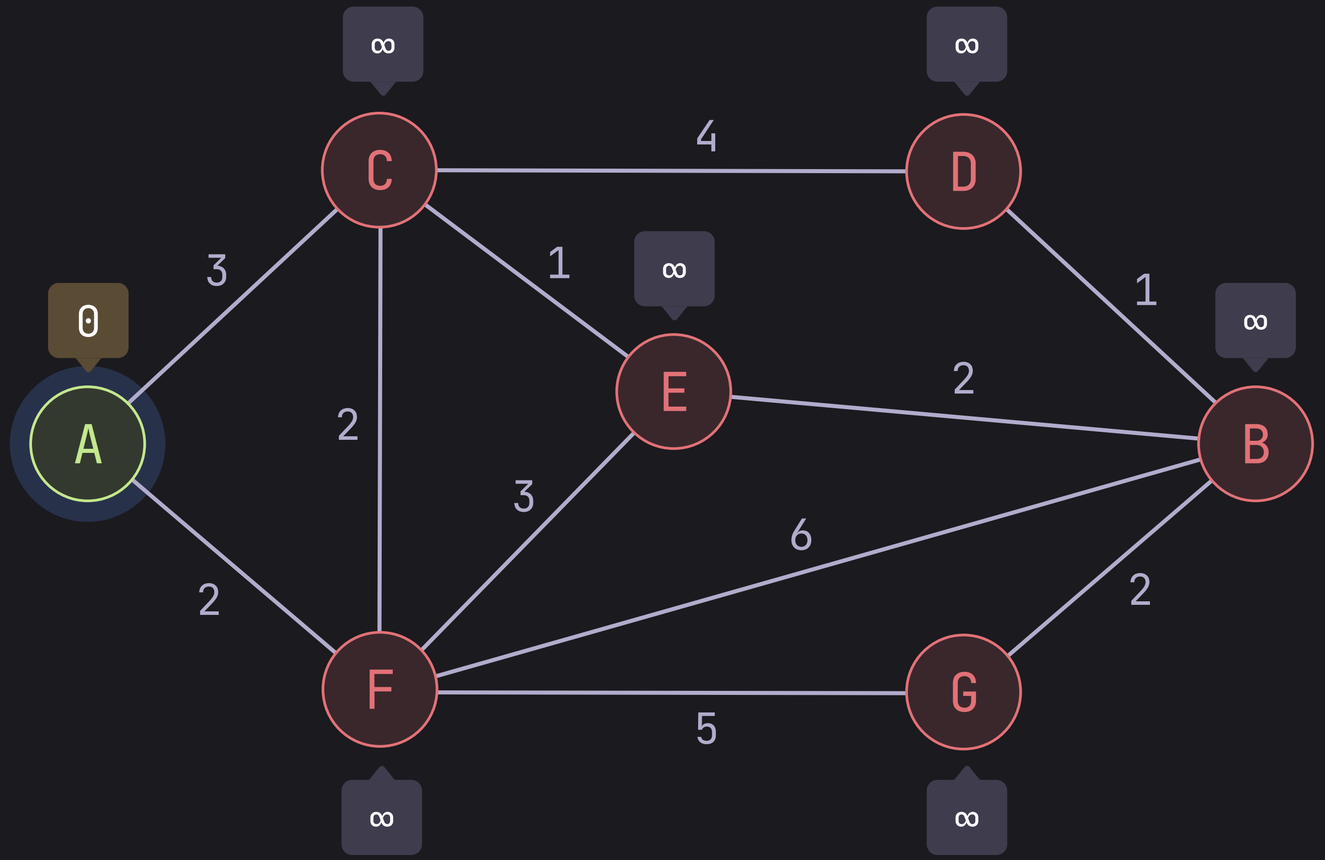
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour
2. Choose next vertex to visit

But what we really wanted was the **shortest path** from A to B

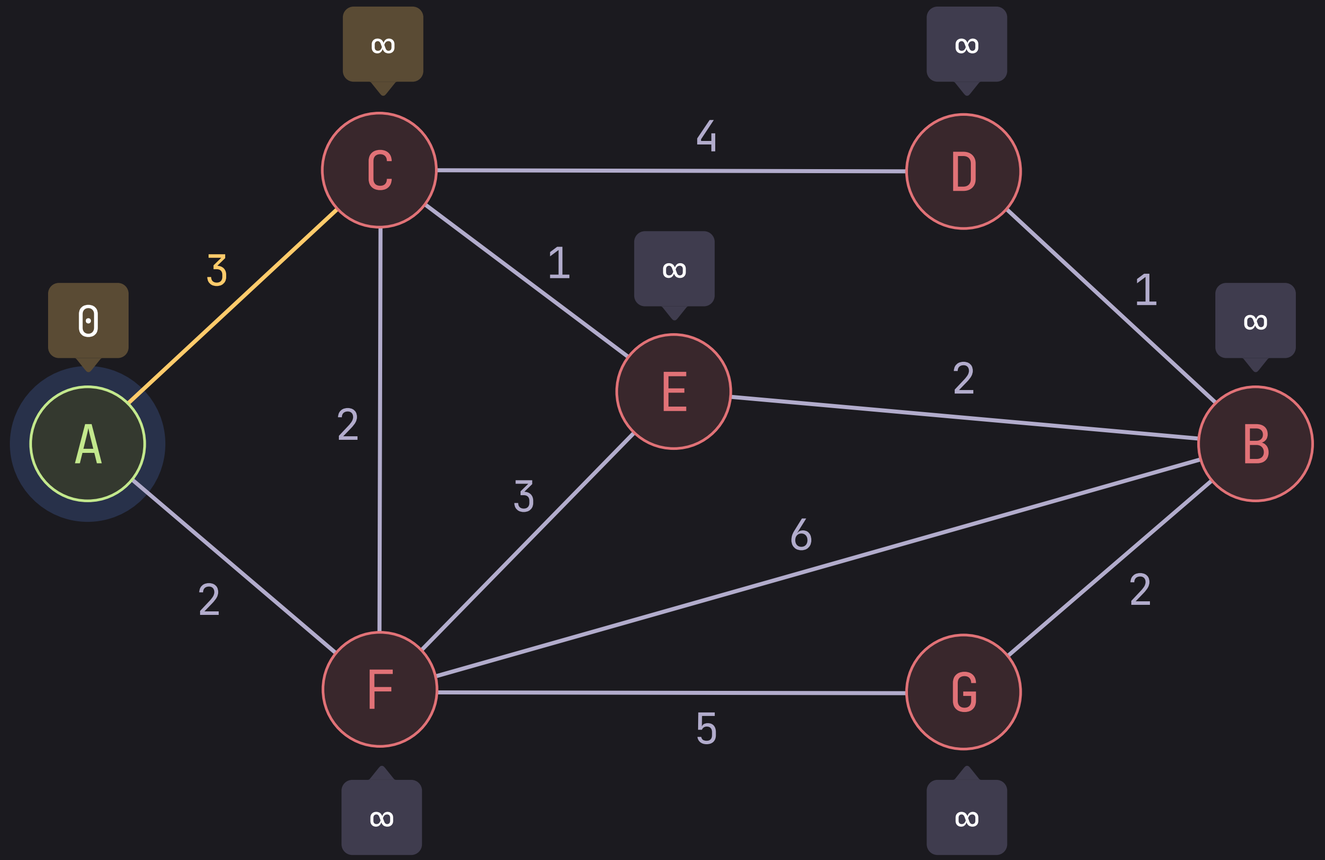
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

To to this we need to update our algorithm slightly

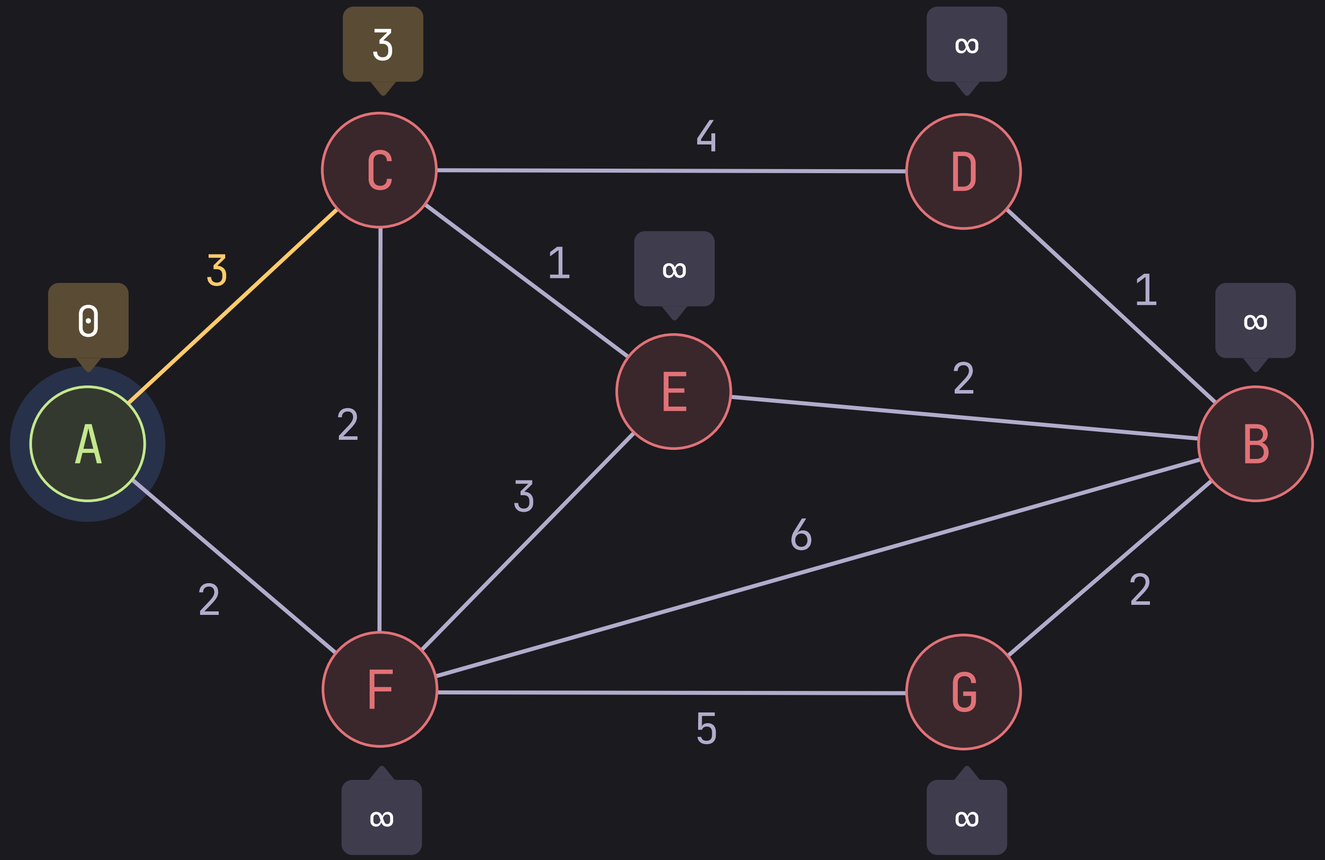
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

We can get to vertex C through A in 0 + 3 = 3 minutes

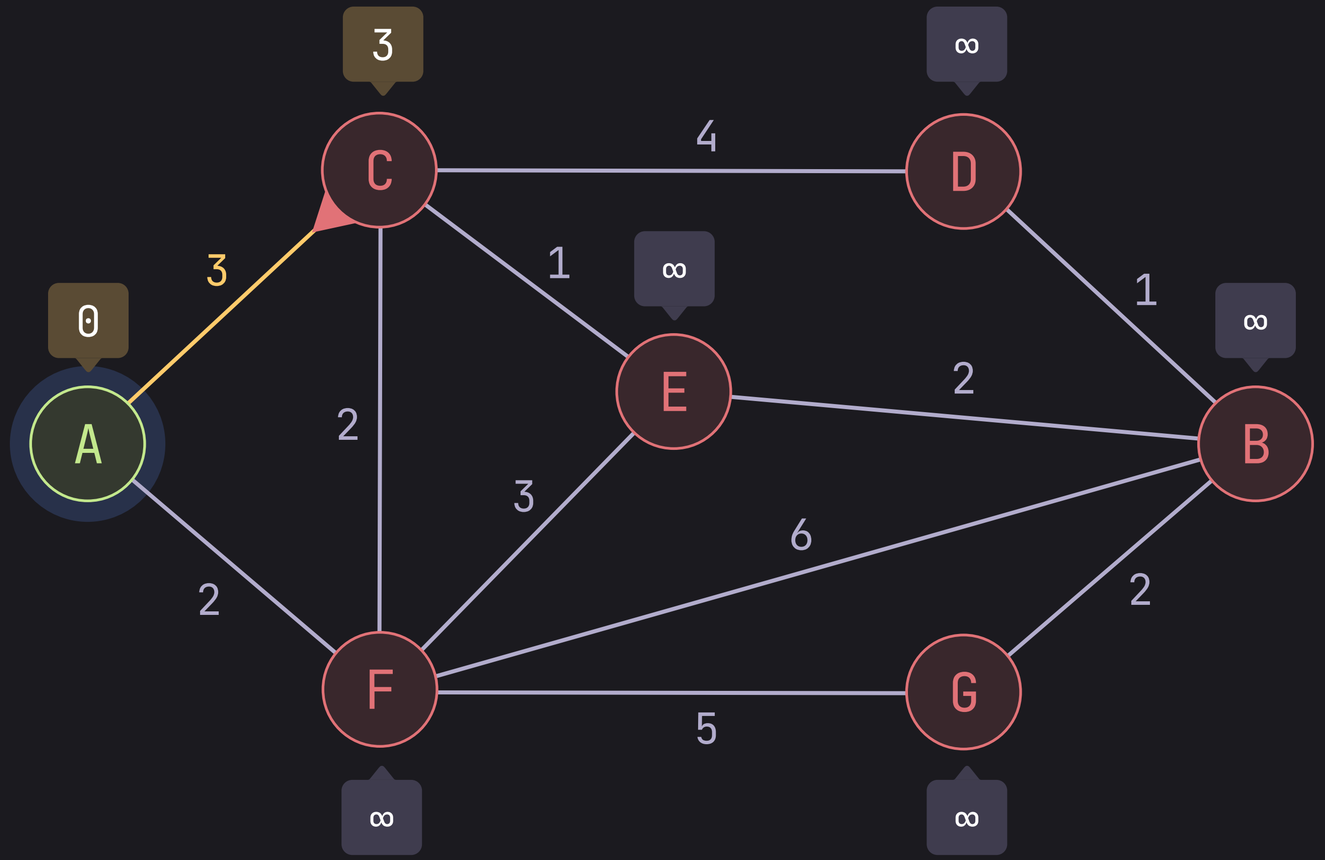
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

As 3 minutes is better than ∞ minutes, we update the estimate

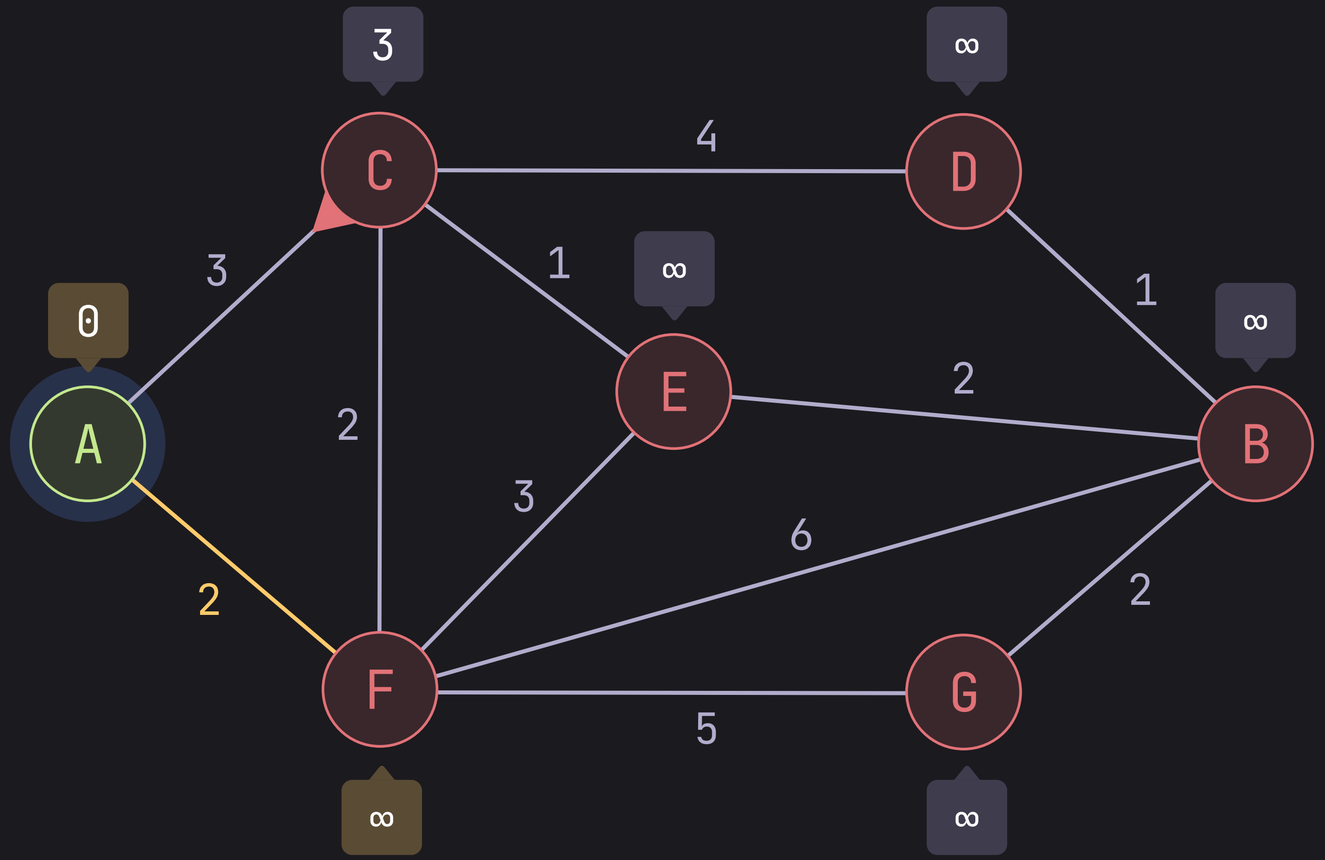
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

Because we updated the value we update the previous vertex

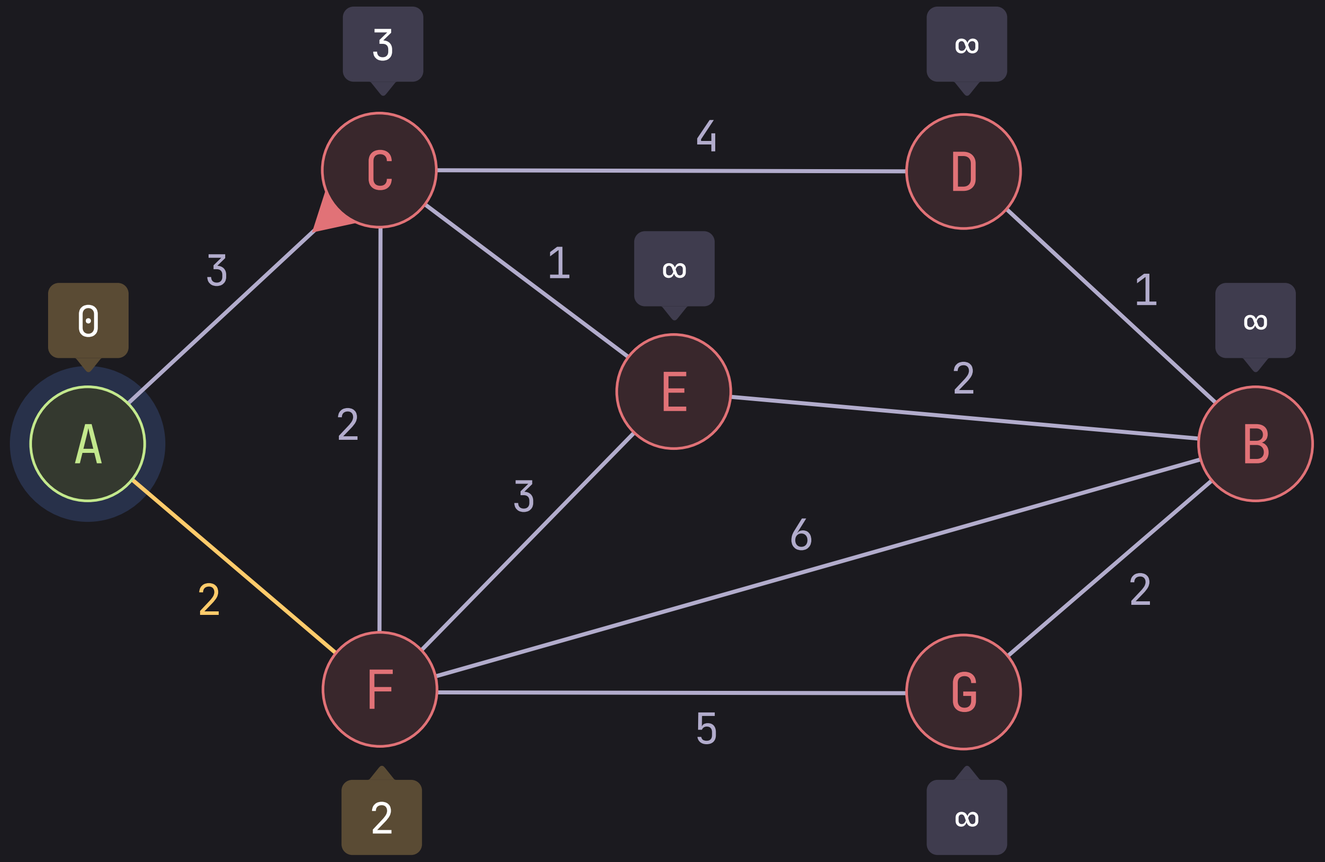
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

We can get to vertex F through A in 0 + 2 = 2 minutes

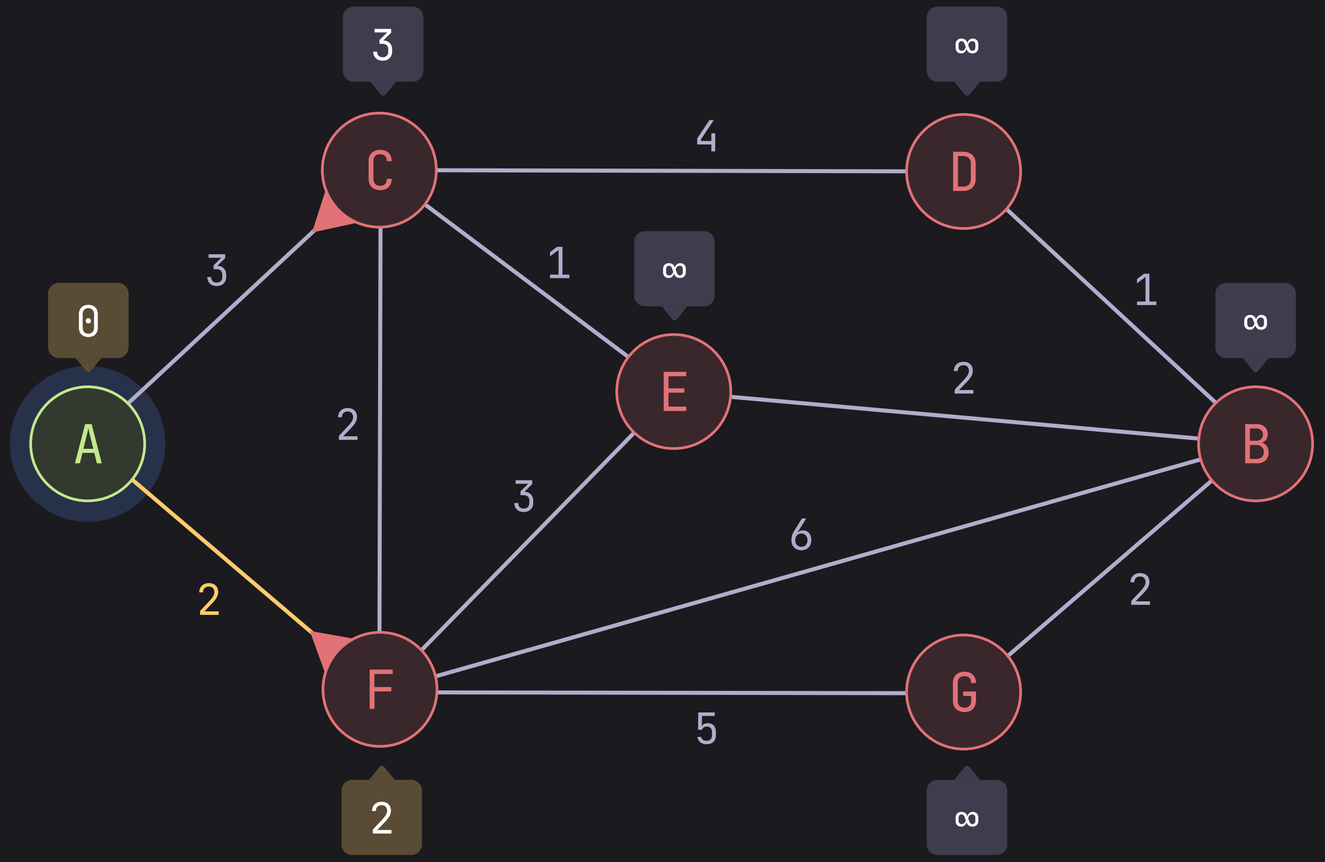
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

As 2 minutes is better than ∞ minutes, we update the estimate

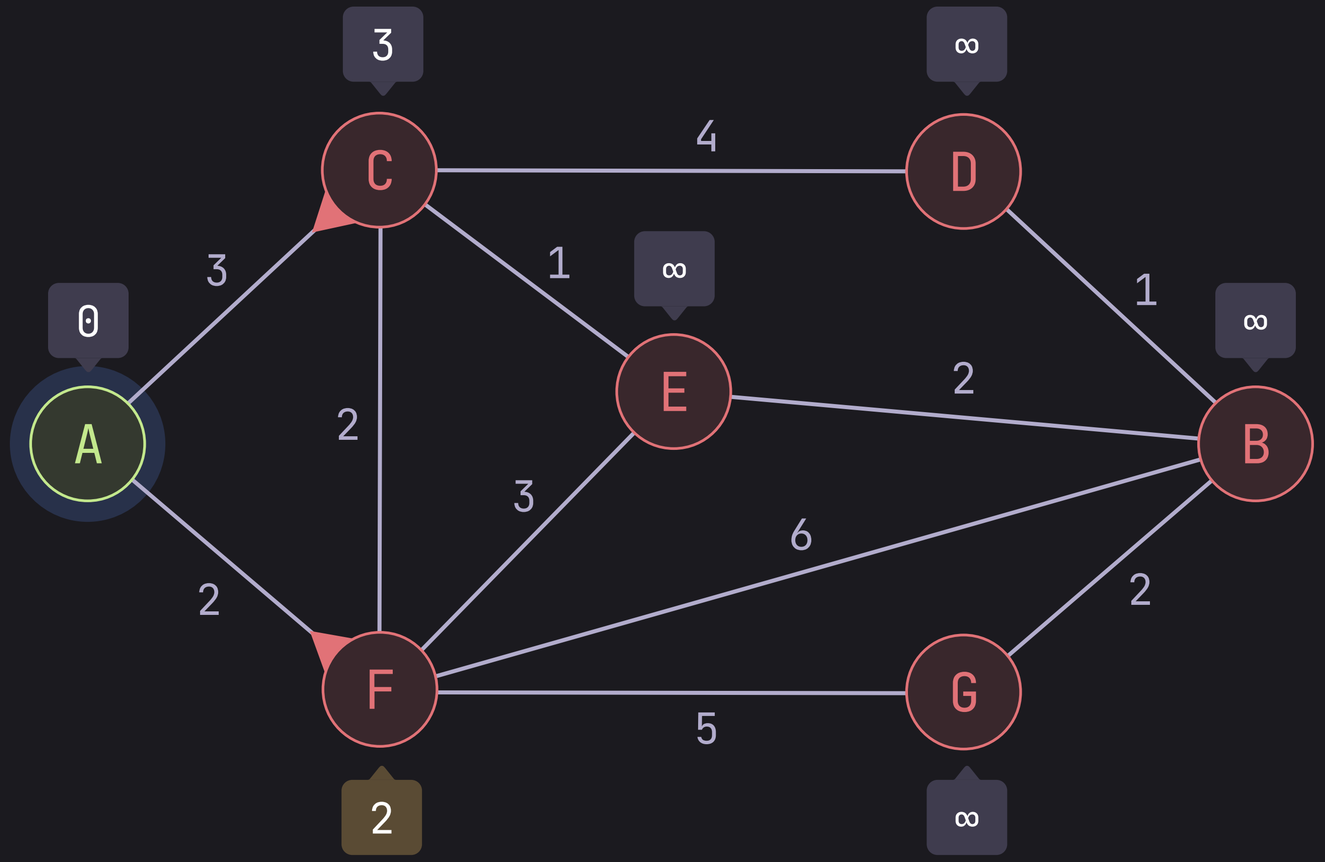
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

Because we updated the value we update the previous vertex

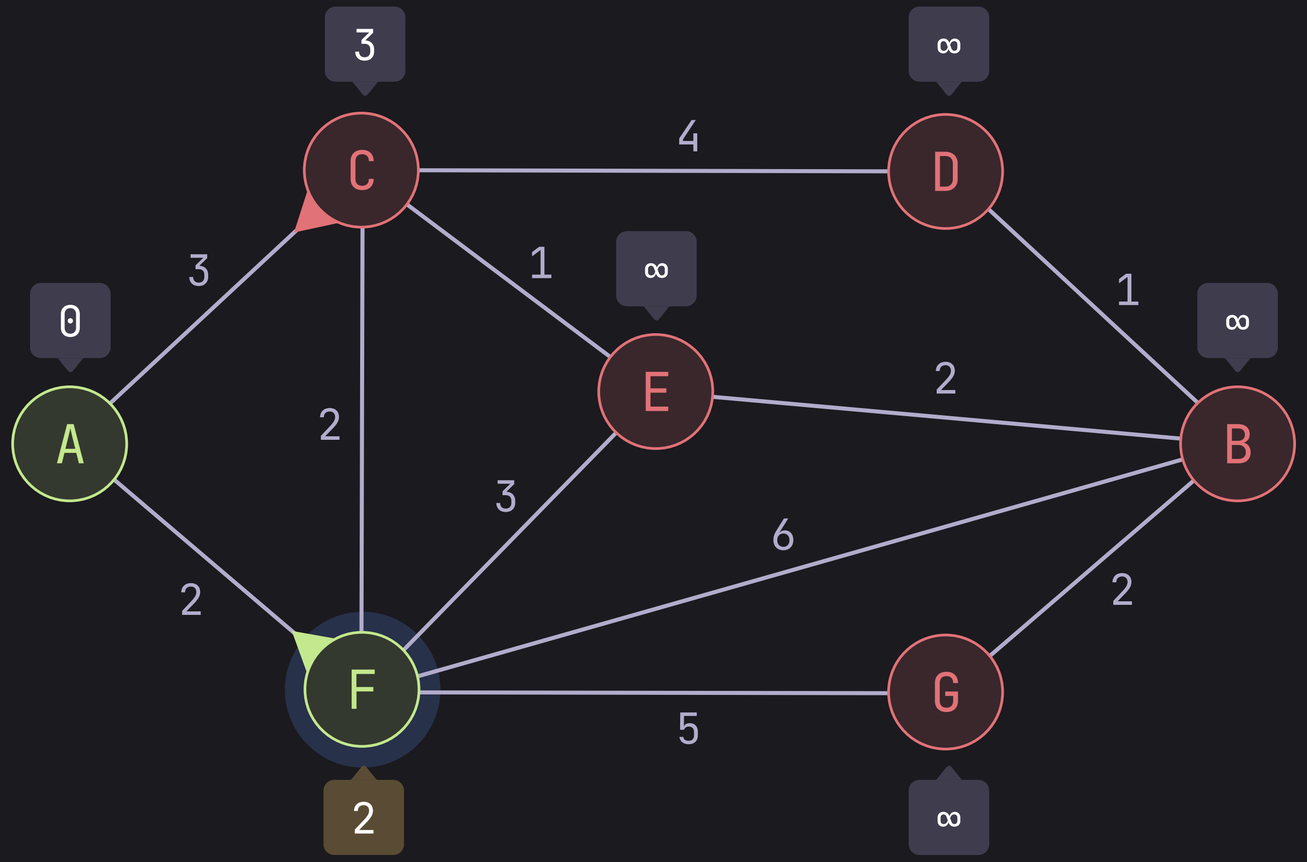
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

Vertex F has the shortest estimate so we will visit F

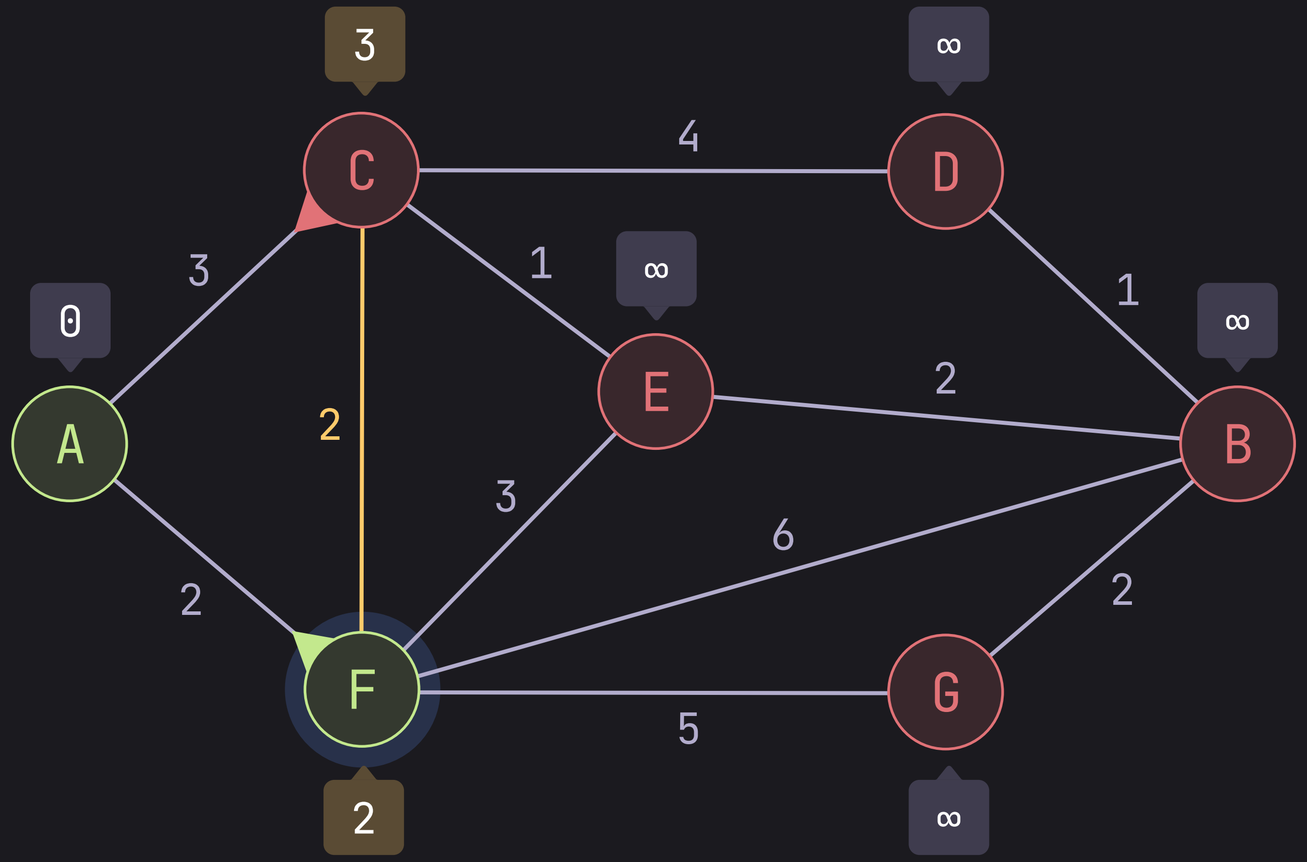
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

Vertex F has the shortest estimate so we will visit F

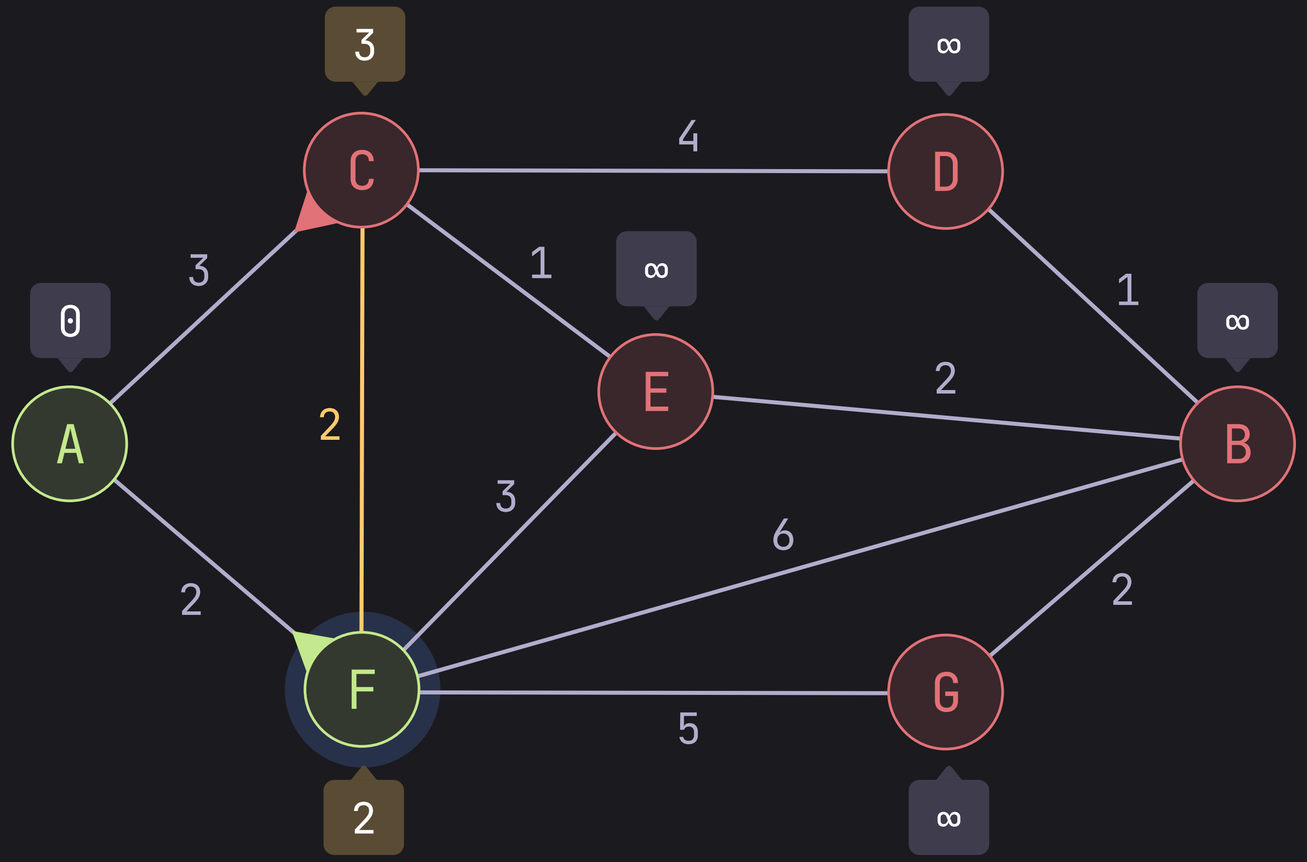
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

We can get to vertex C through F in 2 + 2 = 4 minutes

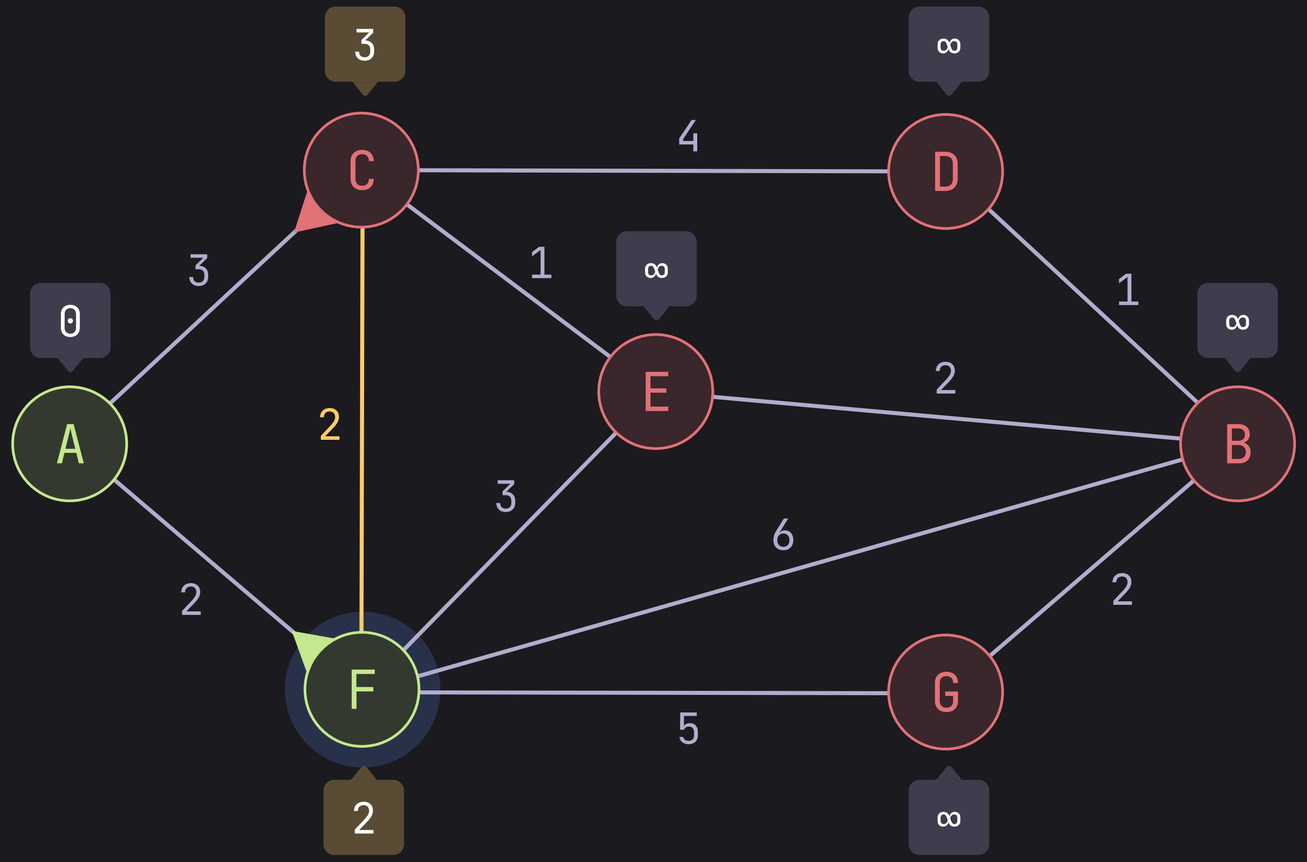
Dijkstra’s Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

As 4 minutes is worse than 3 minutes we don’t update the estimate

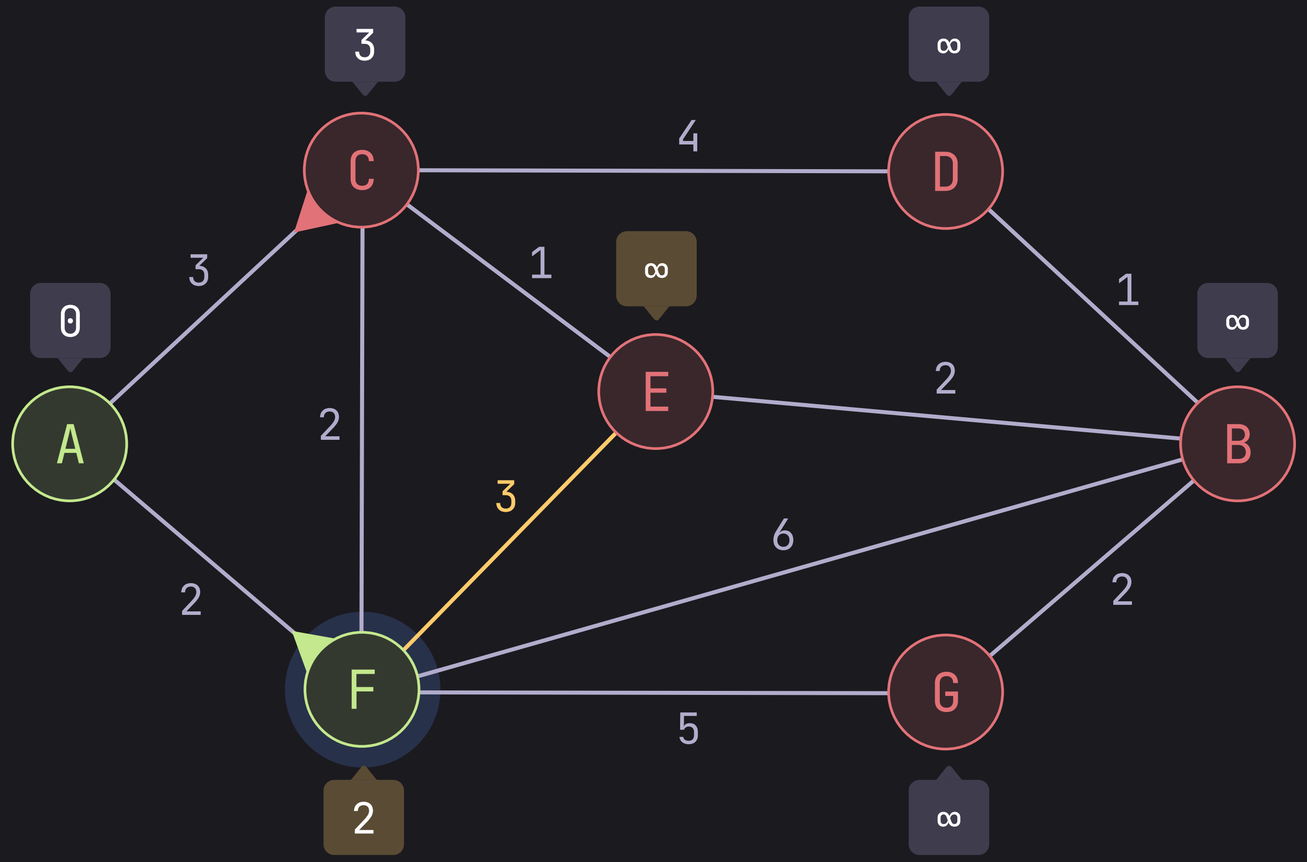
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

So we also do not update the previous vertex

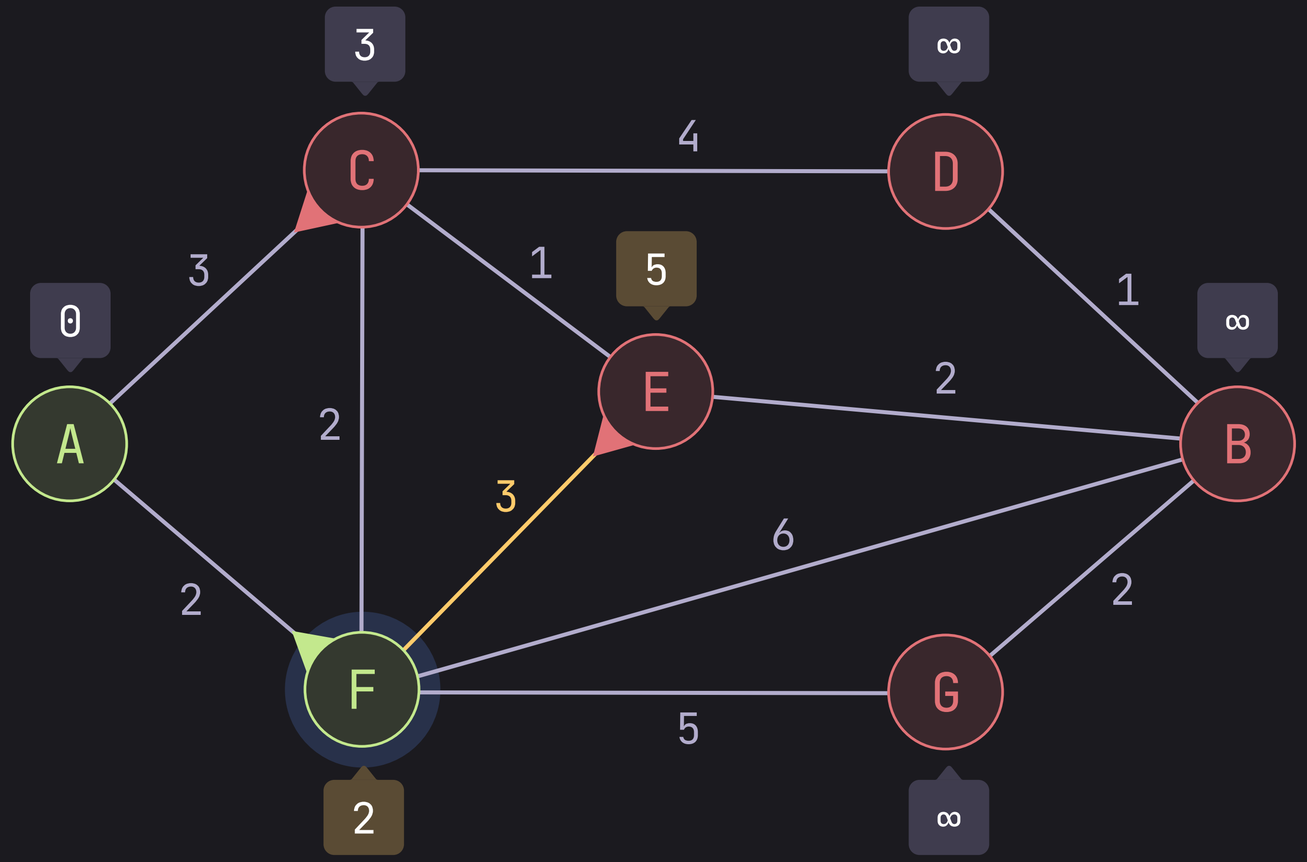
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

We can get to vertex E through F in 2 + 3 = 5 minutes

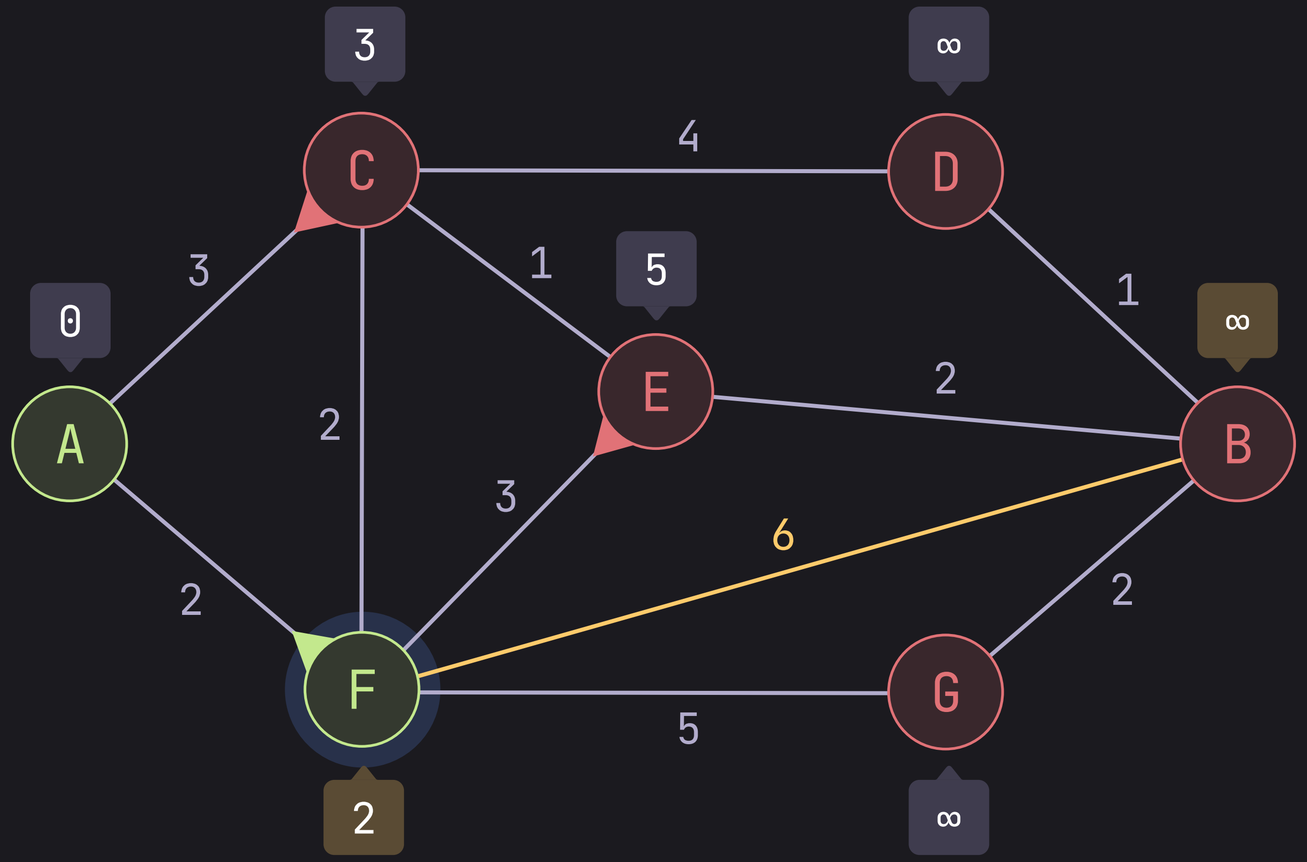
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

As 5 minutes is better than ∞ minutes we update the estimate and we set the previous vertex

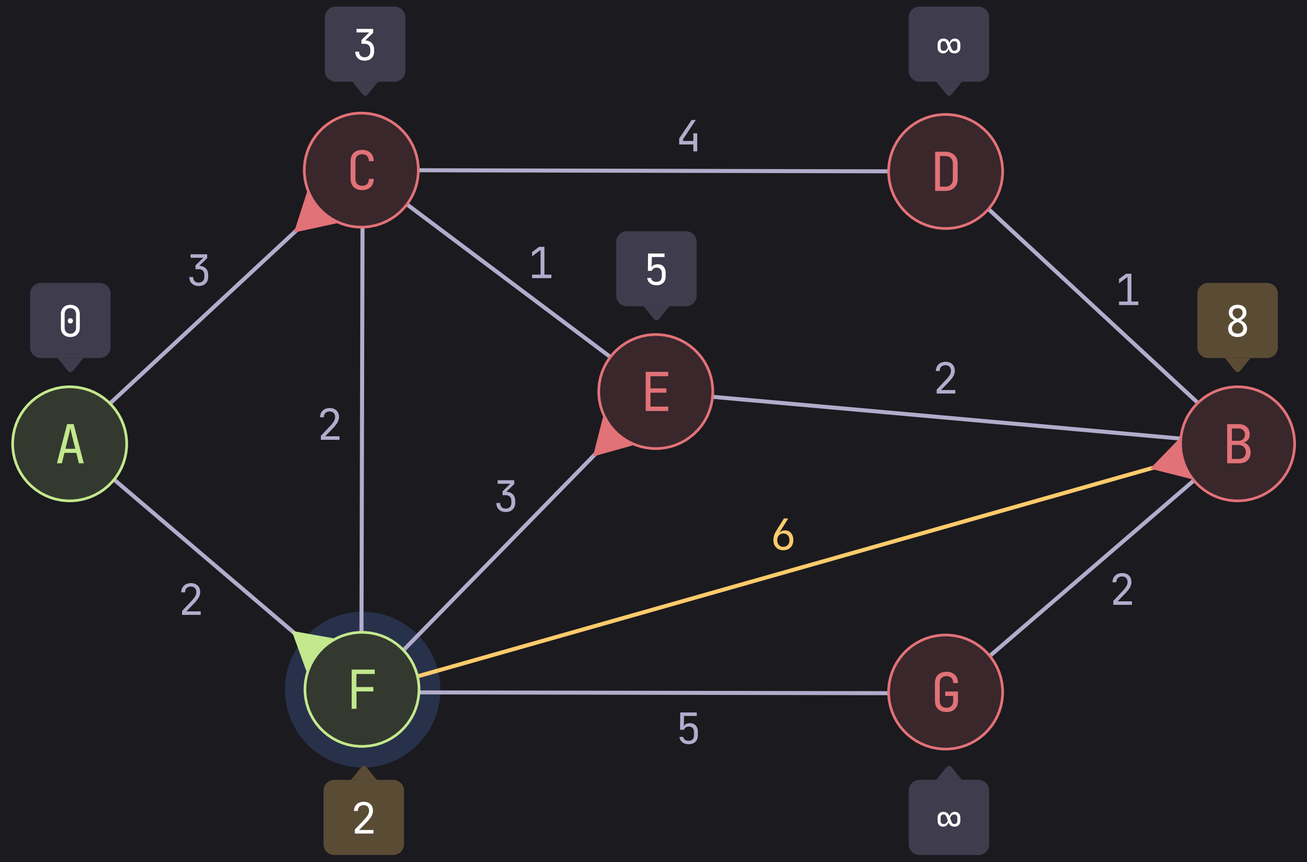
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

We can get to vertex B through F in 2 + 6 = 8 minutes

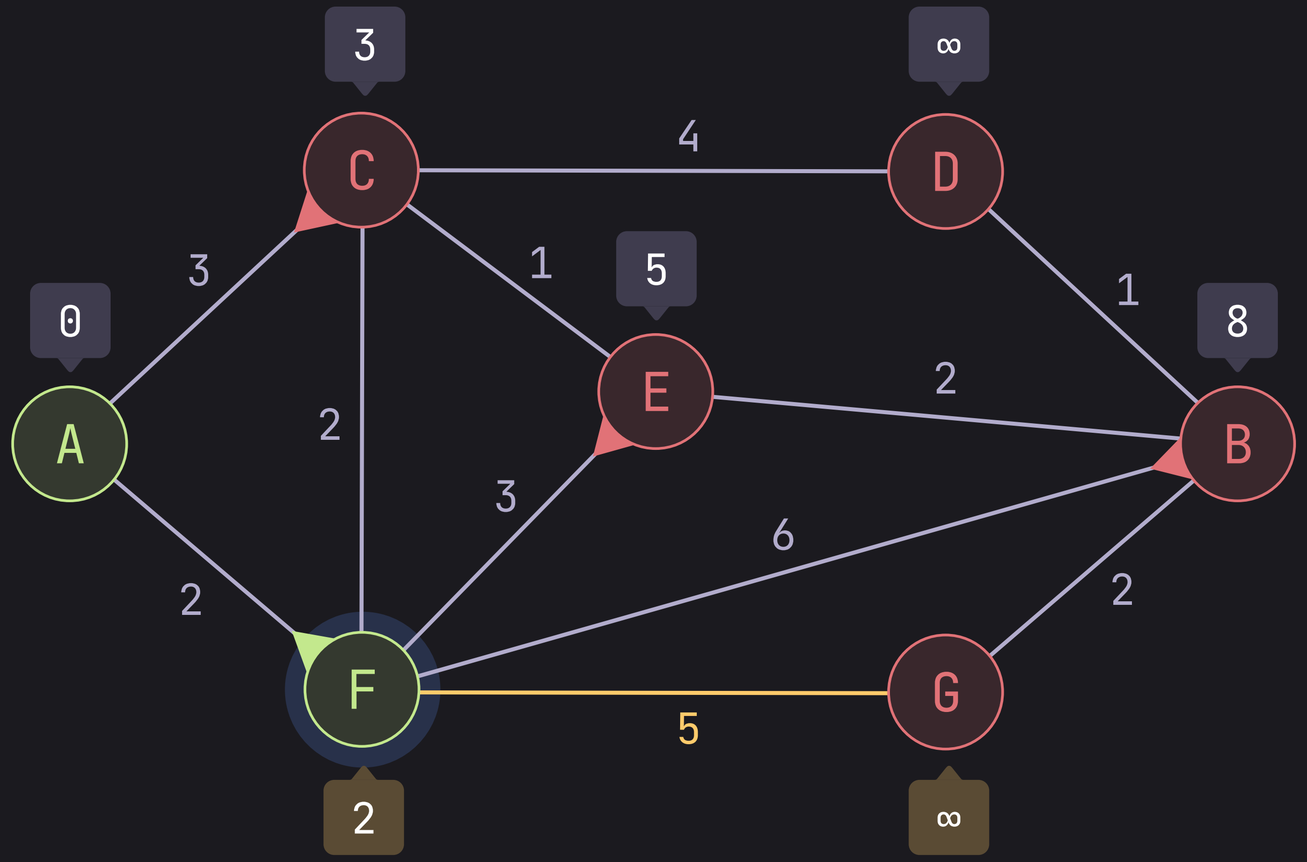
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

As 8 minutes is better than ∞ minutes we update the estimate and we set the previous vertex

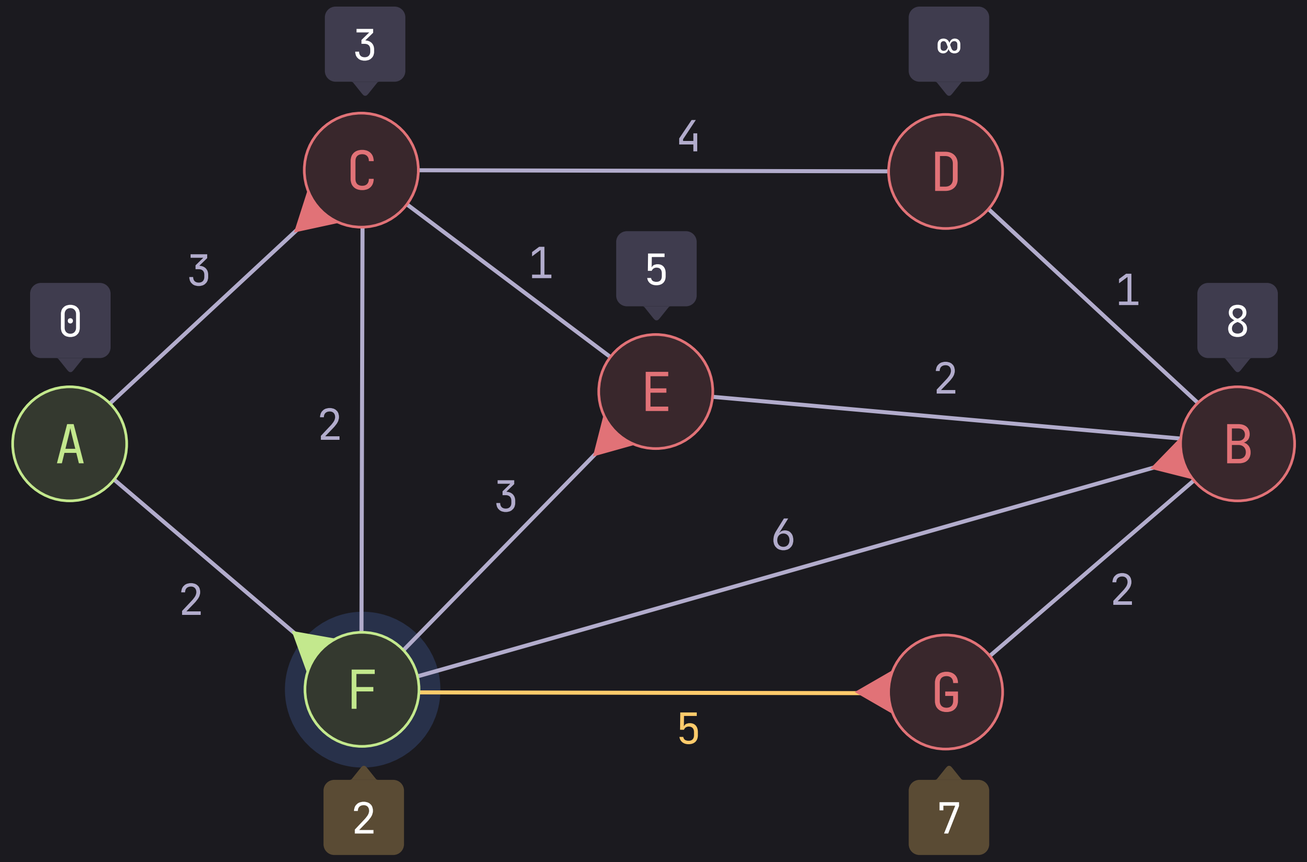
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

We can get to vertex G through F in 2 + 5 = 7 minutes

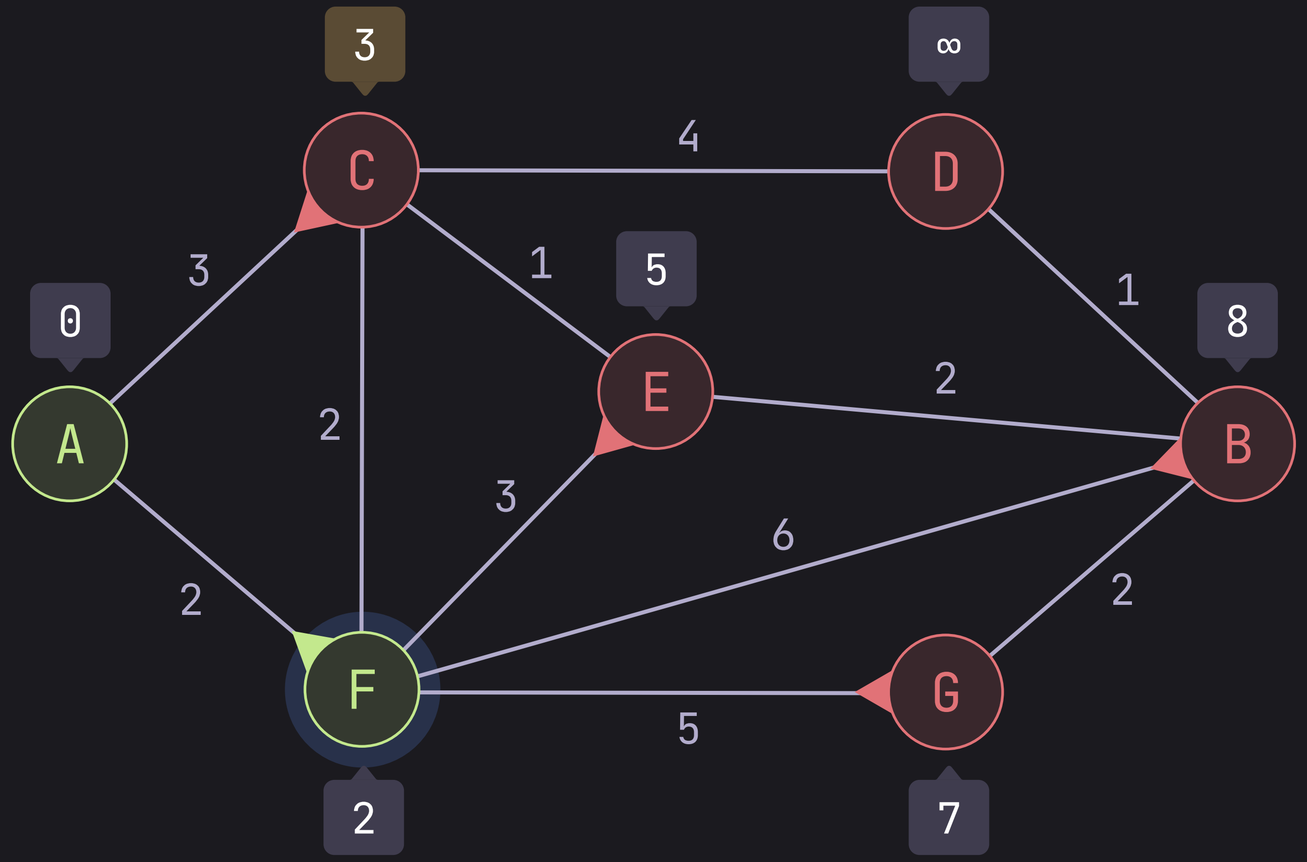
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

As 7 minutes is better than ∞ minutes we update the estimate and we set the previous vertex

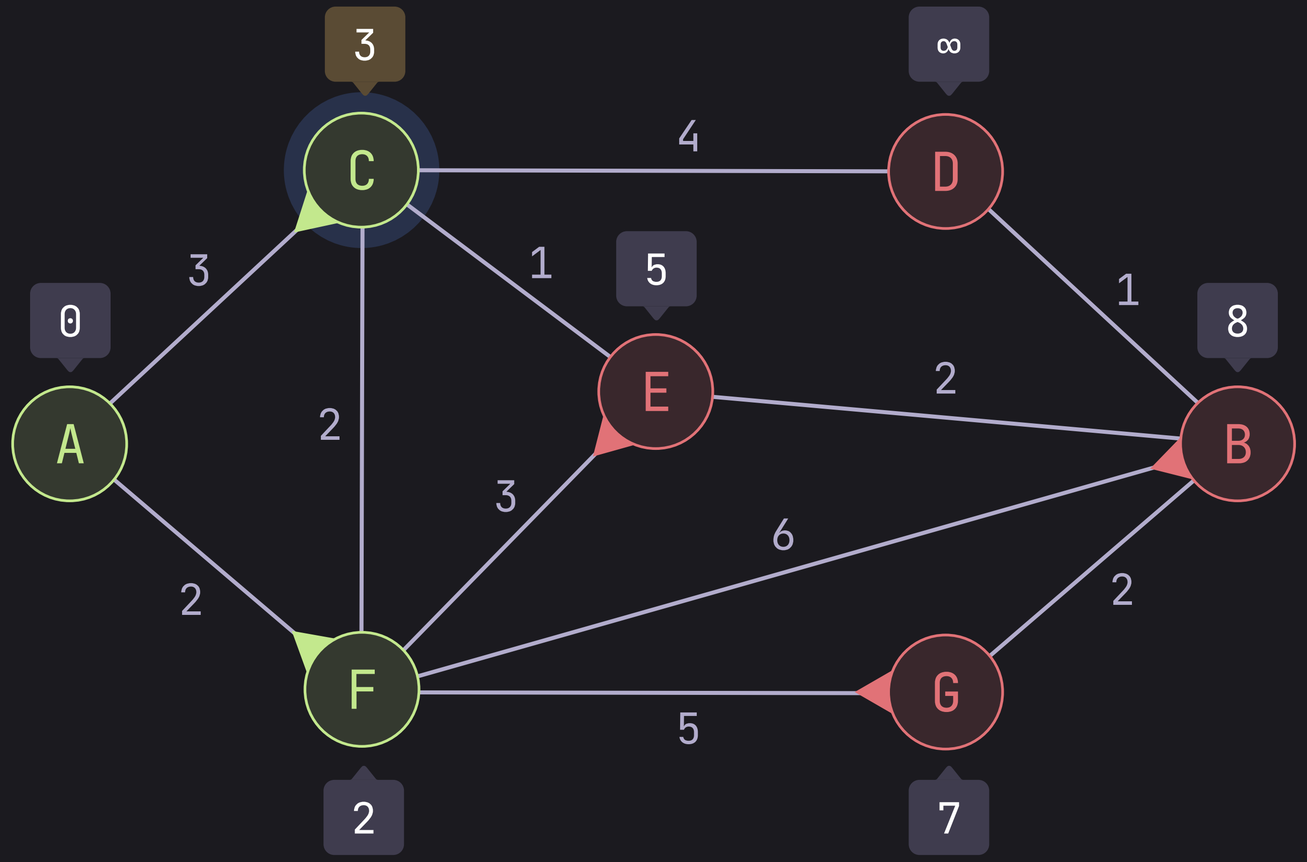
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

Vertex C has the shortest estimate so we will visit C

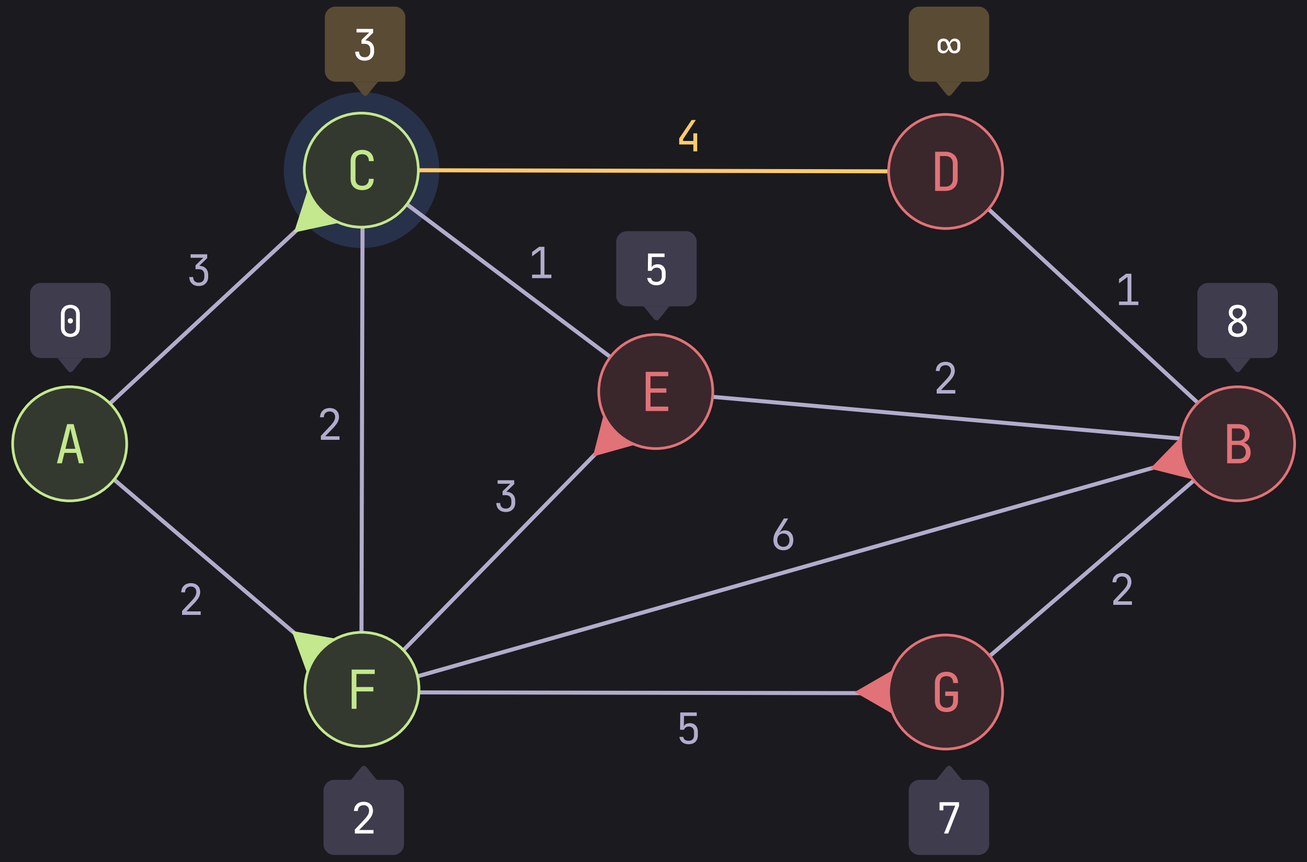
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

Vertex C has the shortest estimate so we will visit C

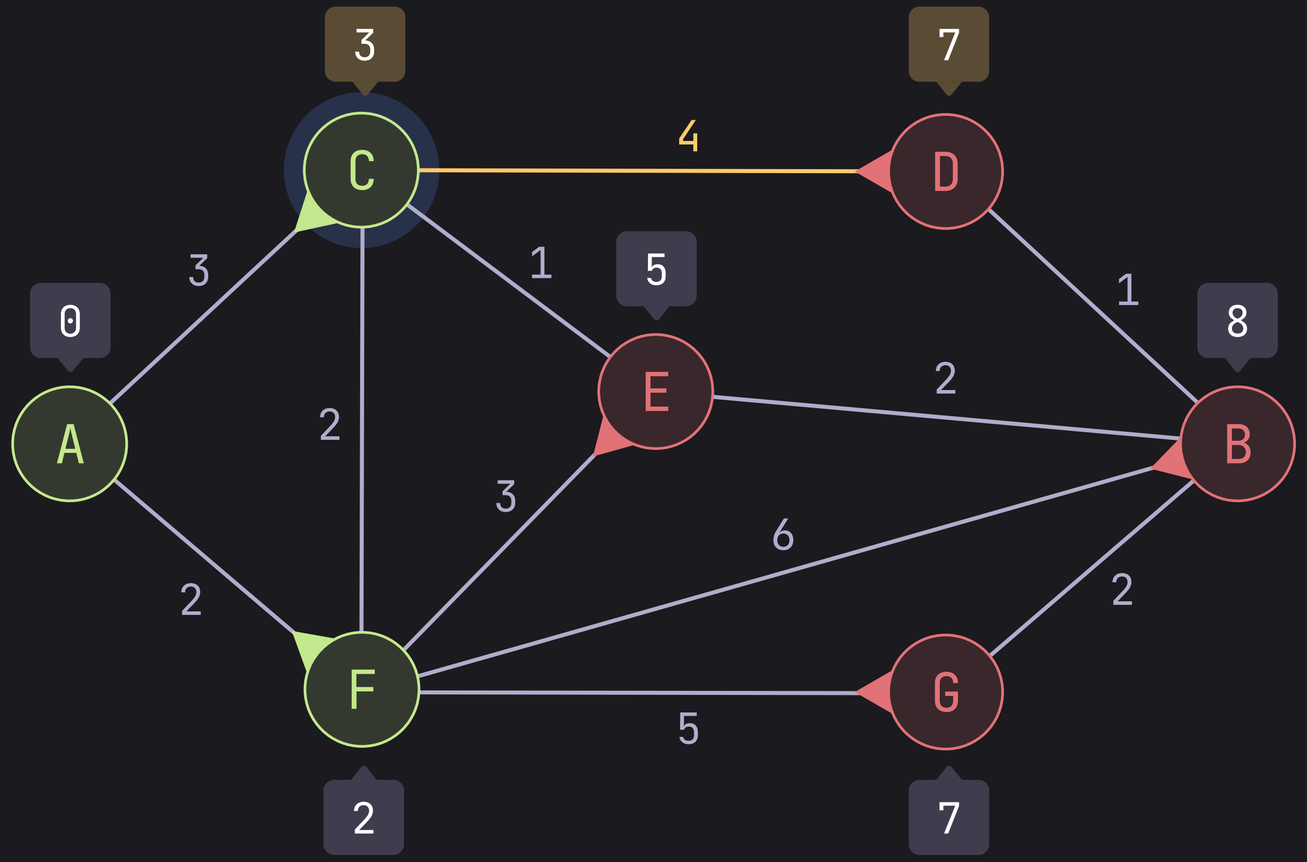
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

We can get to vertex D through C in 3 + 4 = 7 minutes

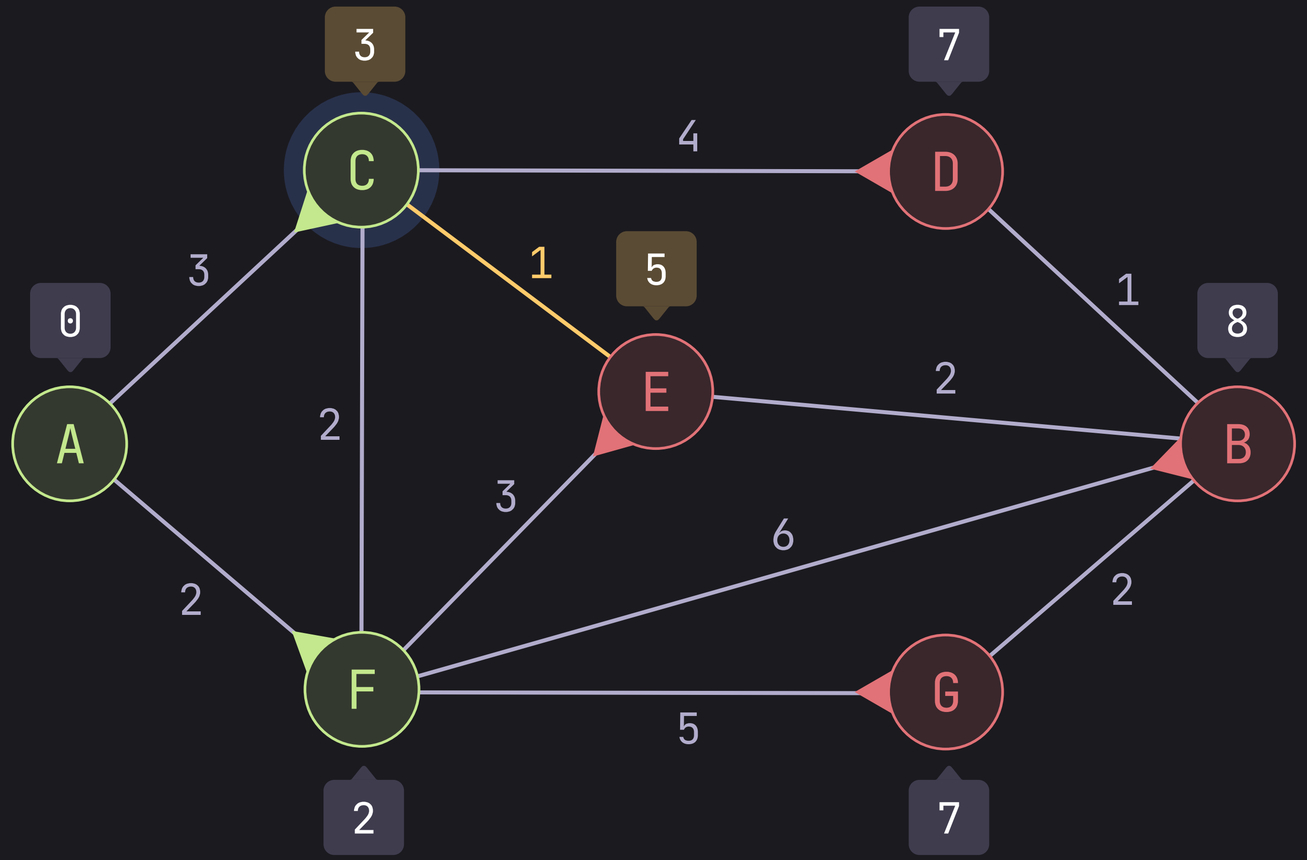
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

As 7 minutes is better than ∞ minutes we update the estimate and we set the previous vertex

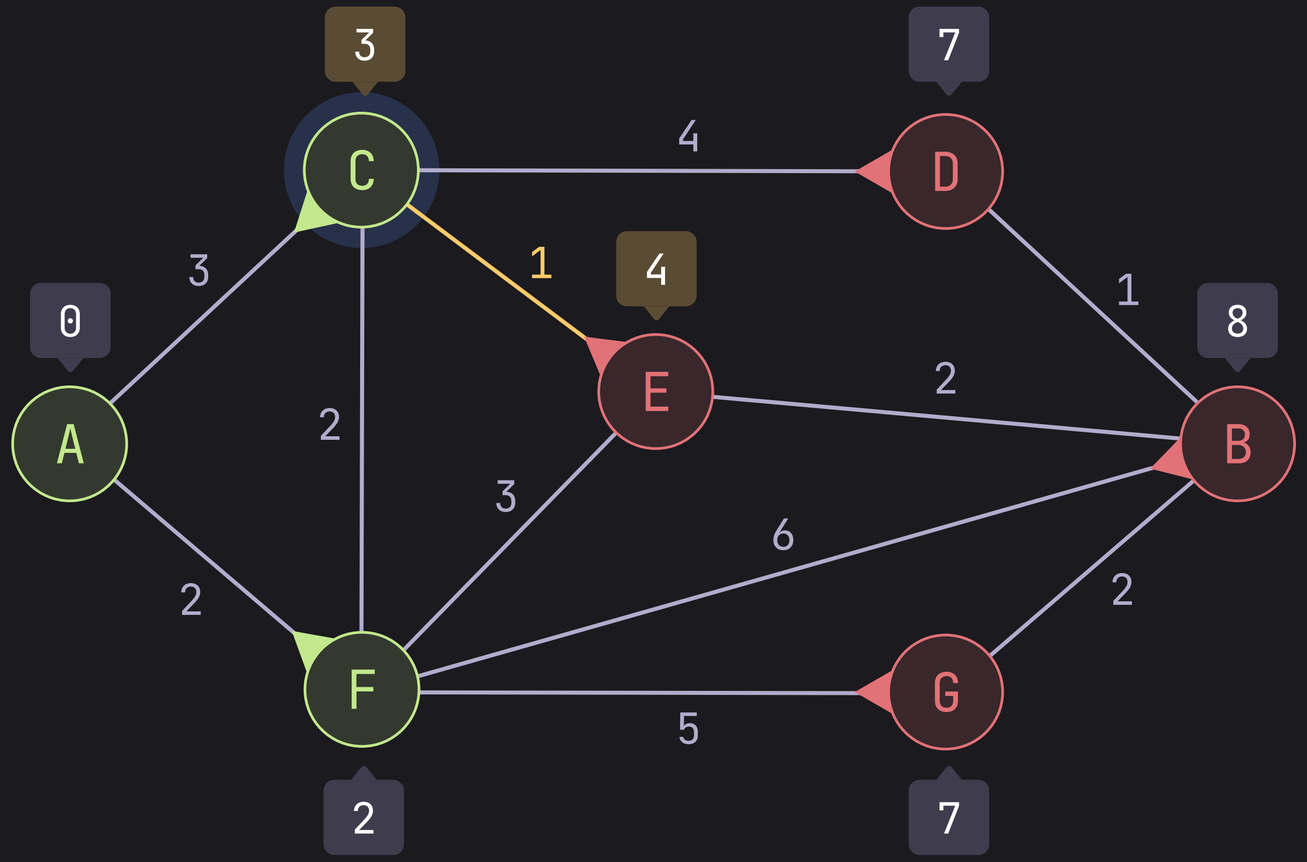
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

We can get to vertex E through C in 3 + 1 = 4 minutes

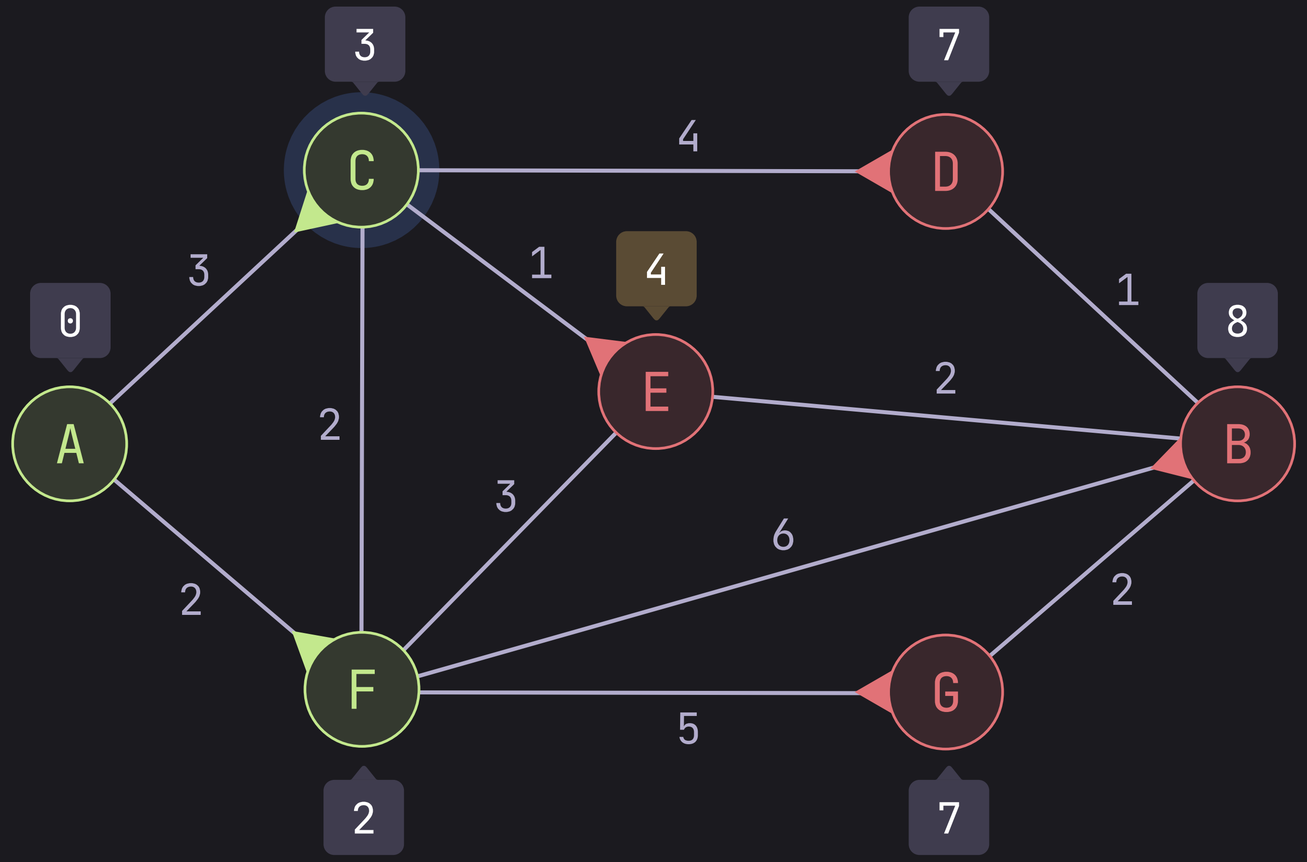
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

As 4 minutes is better than 5 minutes we update the estimate and we set the previous vertex

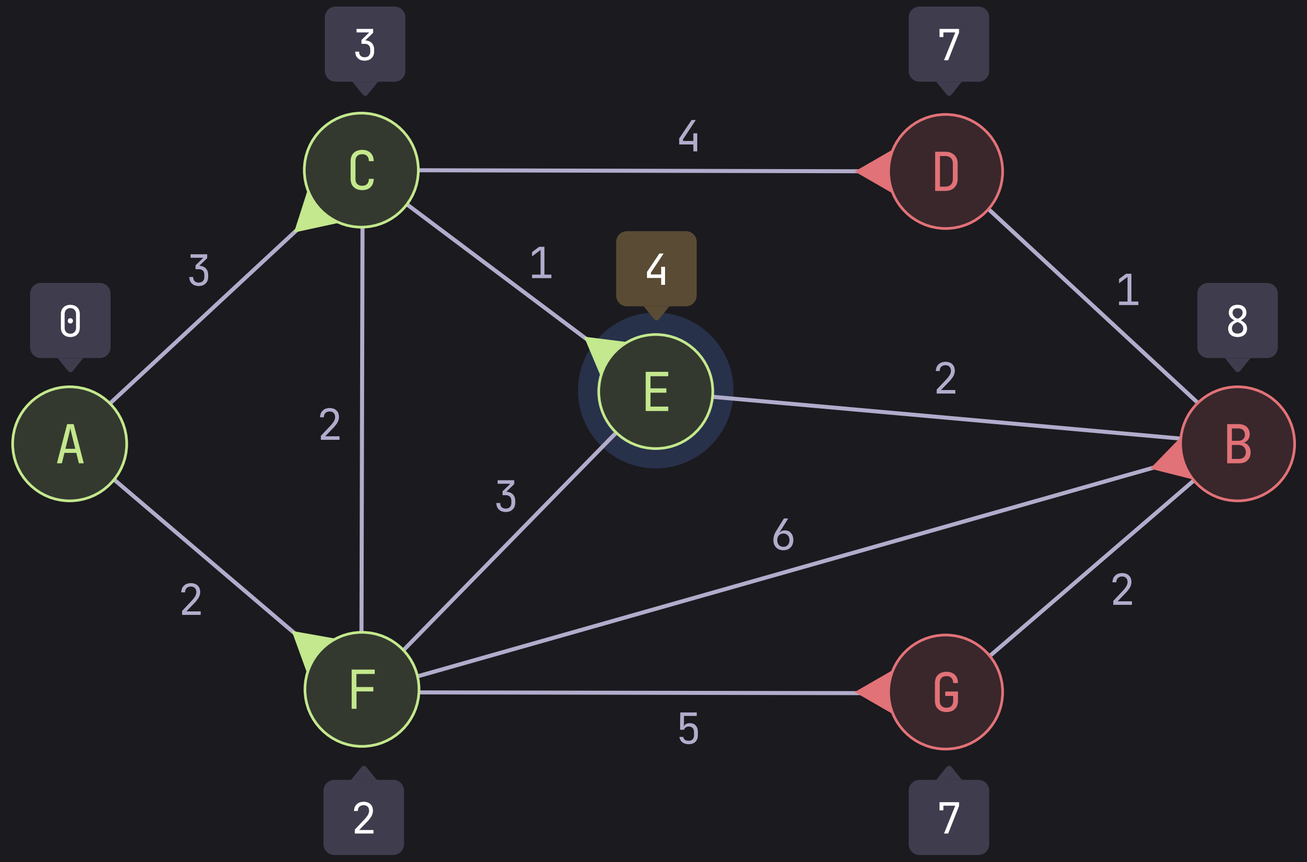
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

Vertex E has the shortest estimate so we will visit E

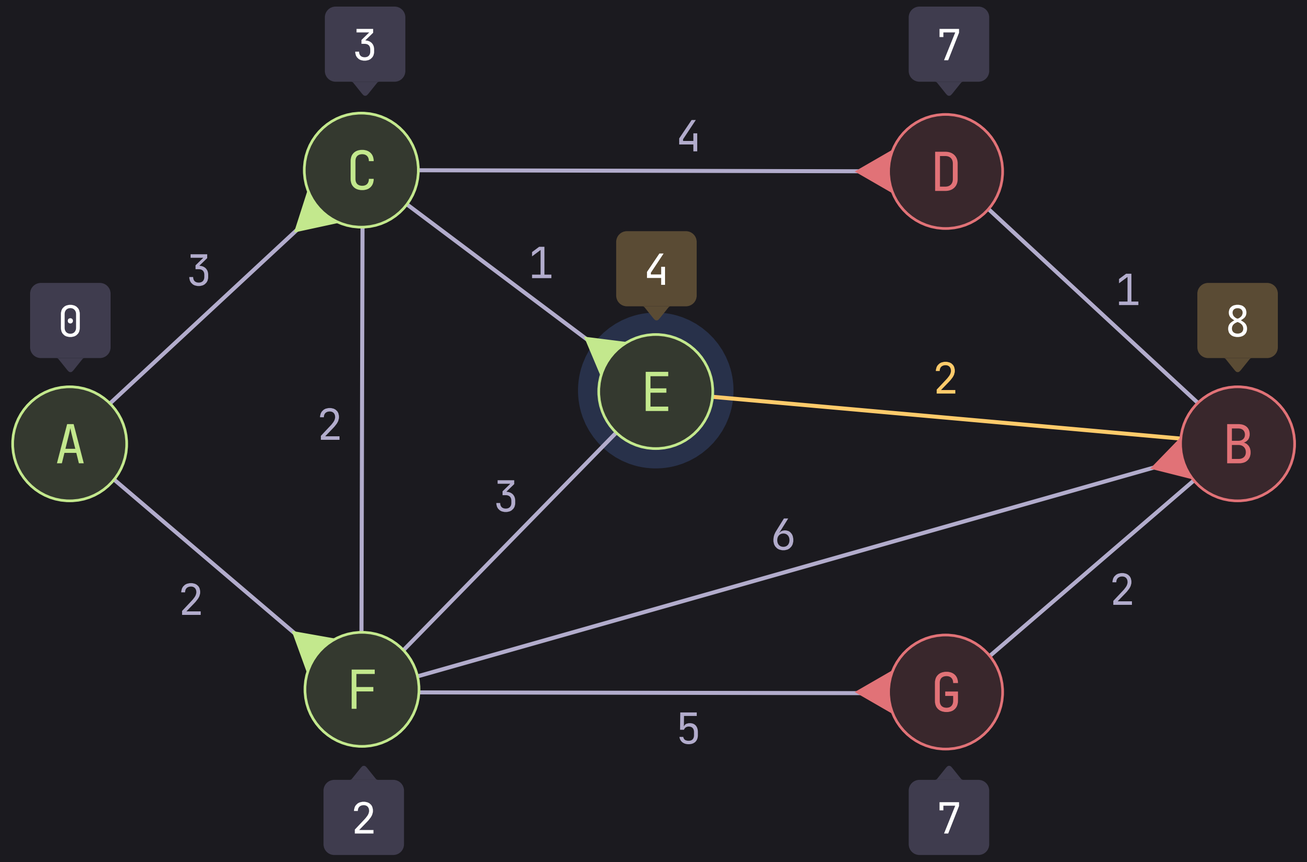
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

Vertex E has the shortest estimate so we will visit E

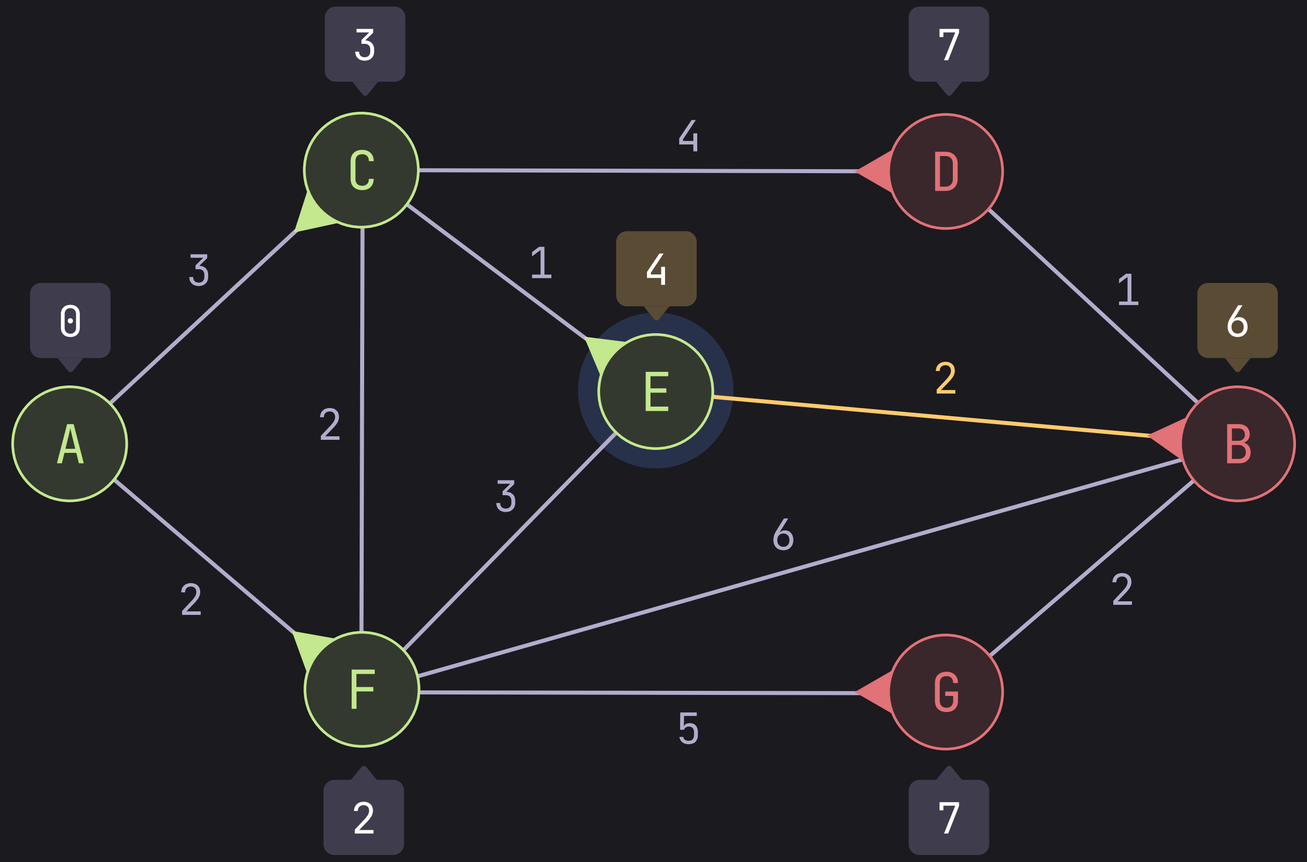
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

We can get to vertex B through E in 4 + 2 = 6 minutes

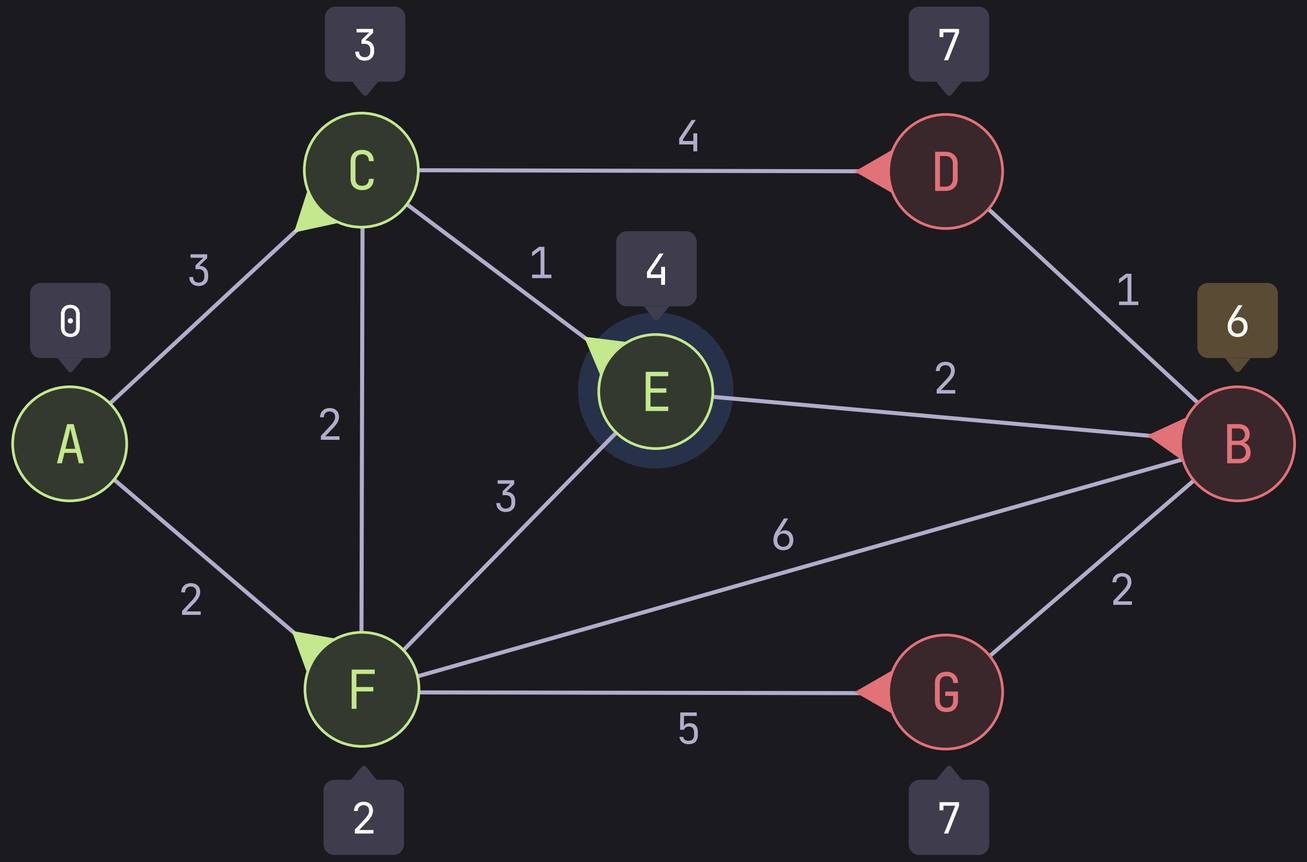
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

As 6 minutes is better than 8 minutes we update the estimate and we set the previous vertex

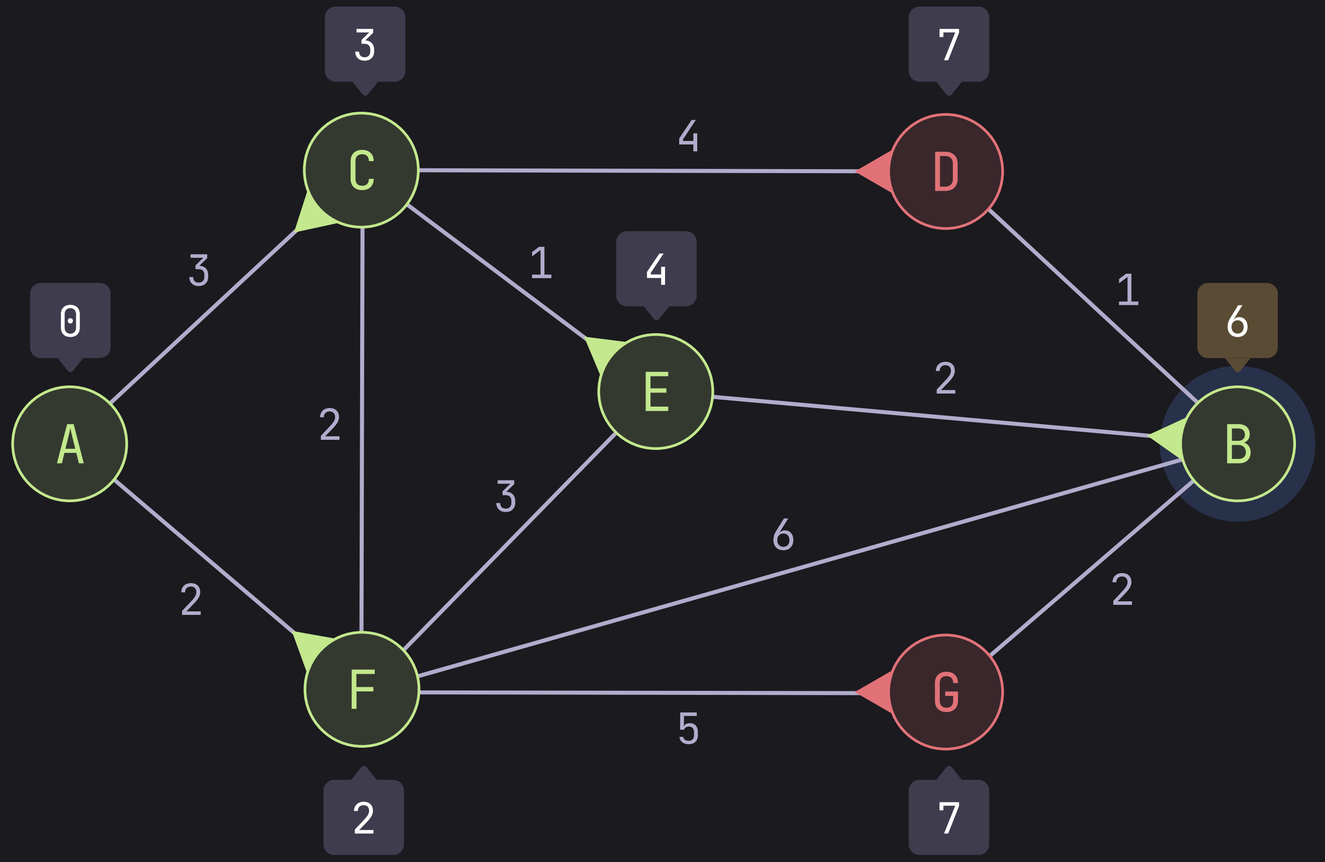
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

Vertex B has the shortest estimate so we will visit B

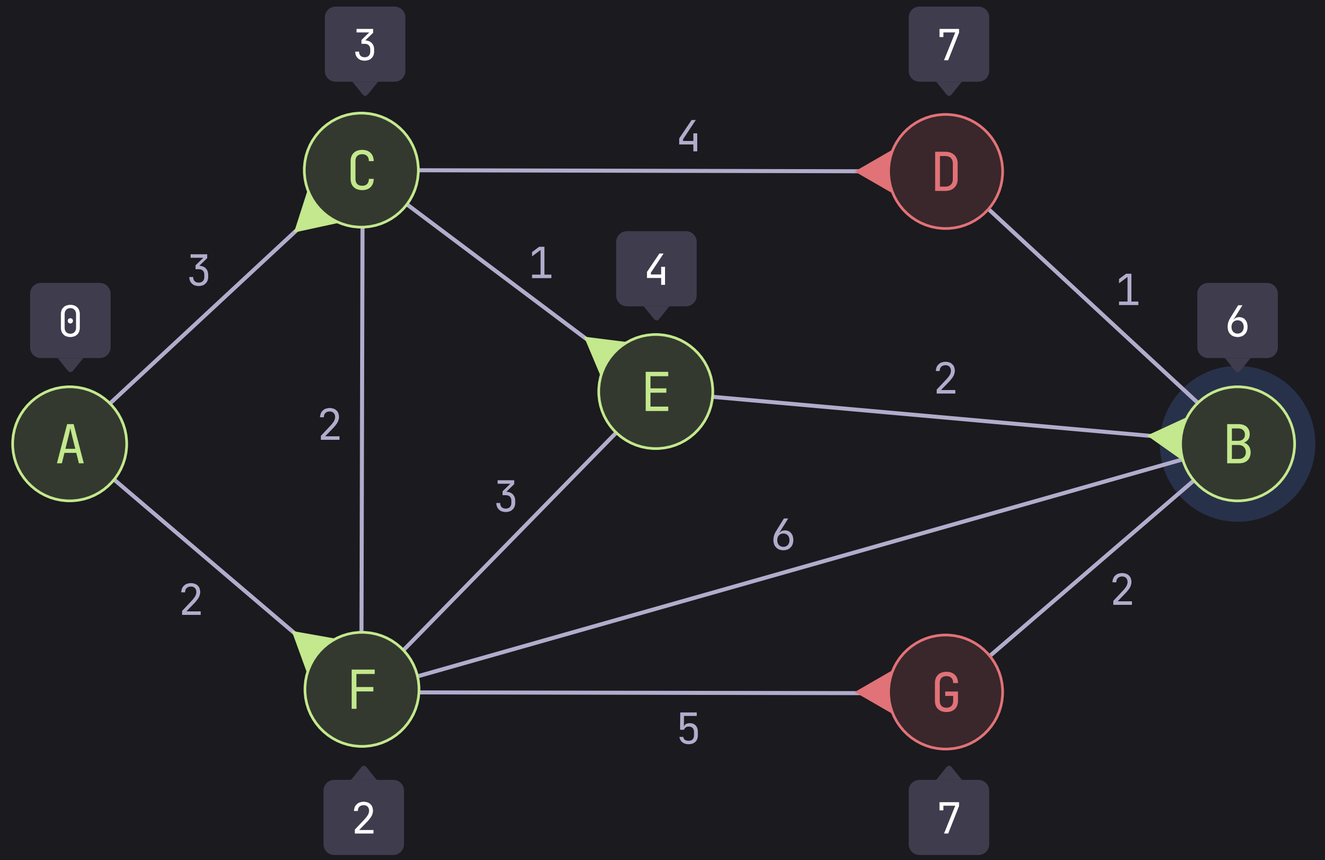
Dijkstra's Algorithm



1. Update estimates for each unvisited neighbour and set previous vertex
2. Choose next vertex to visit

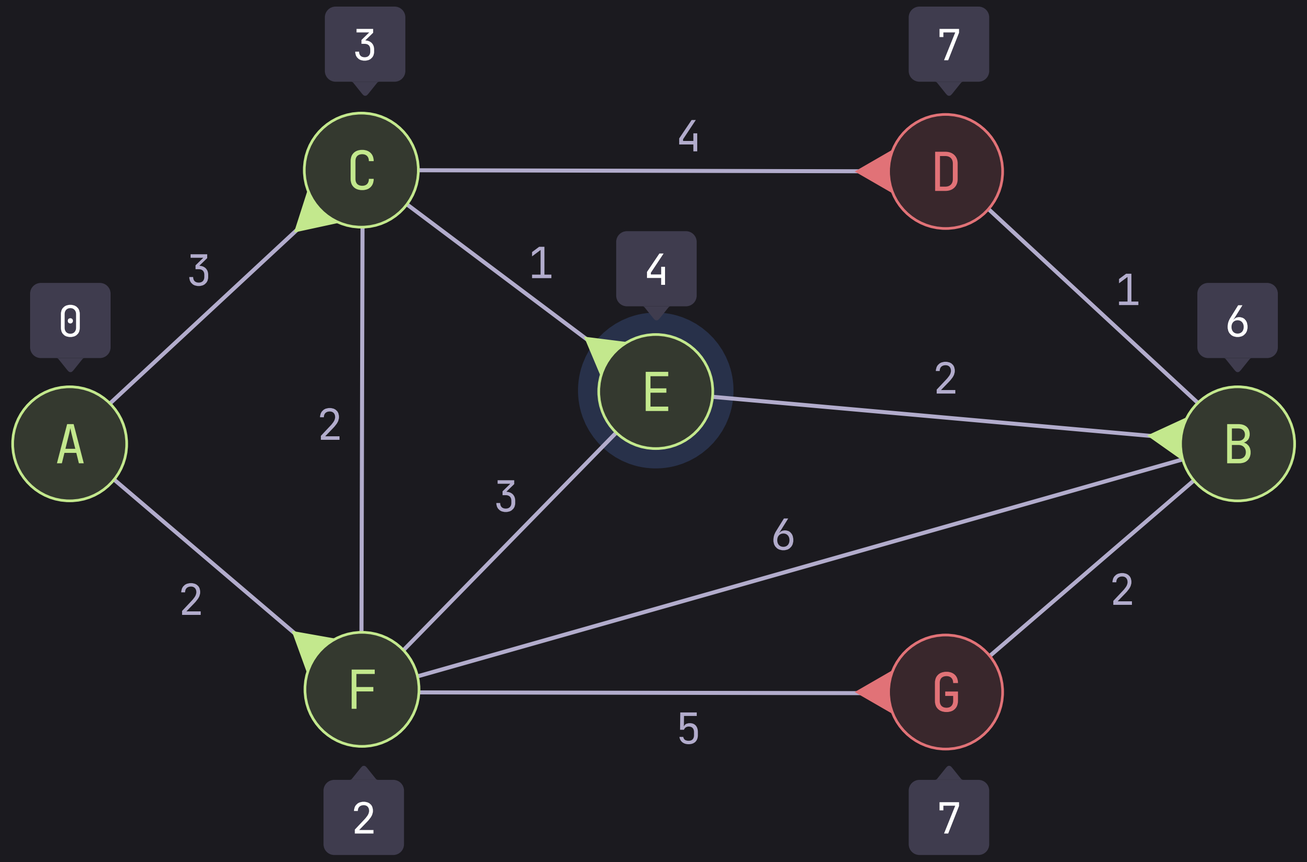
Now we have reached the target node, so we are done

Dijkstra's Algorithm



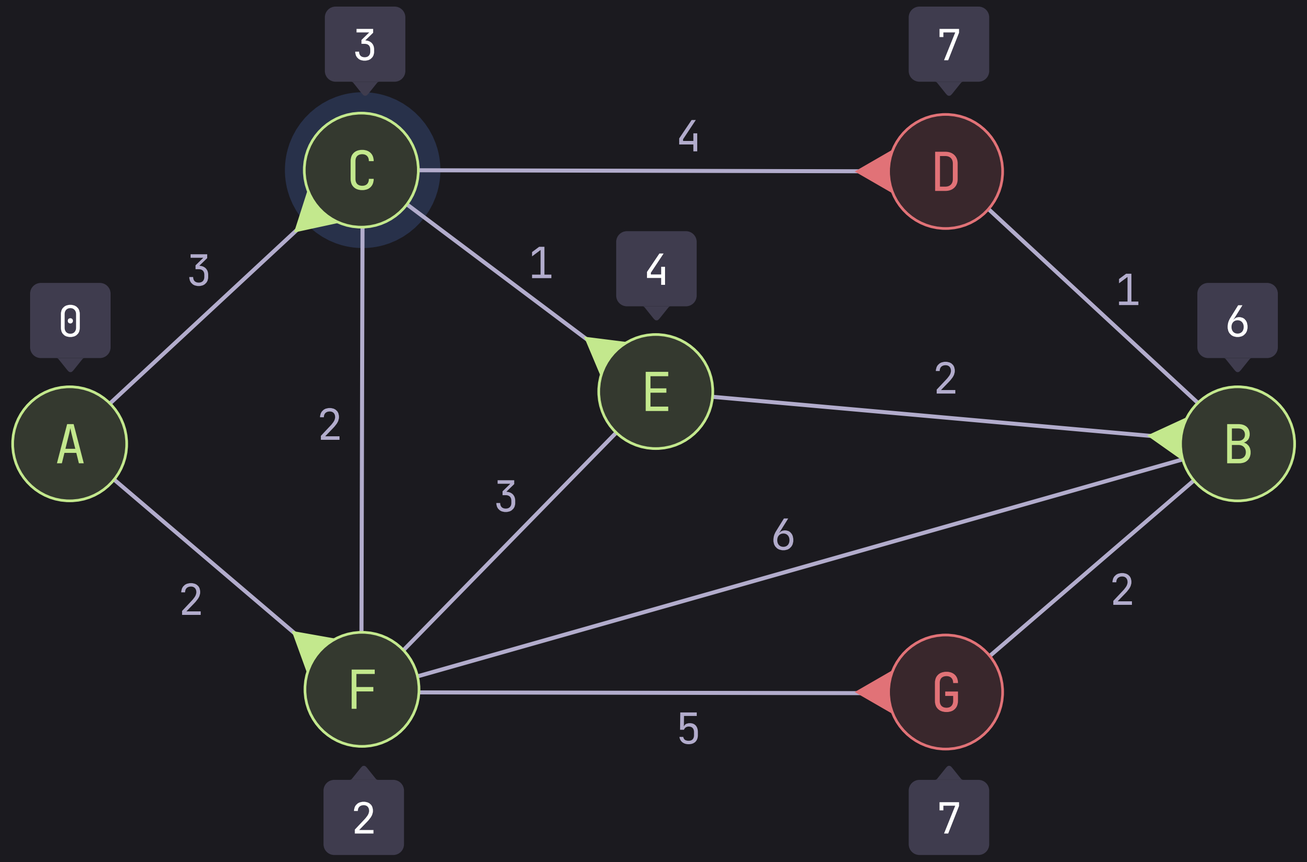
Now to get the path we just follow the arrows back to the start

Dijkstra's Algorithm



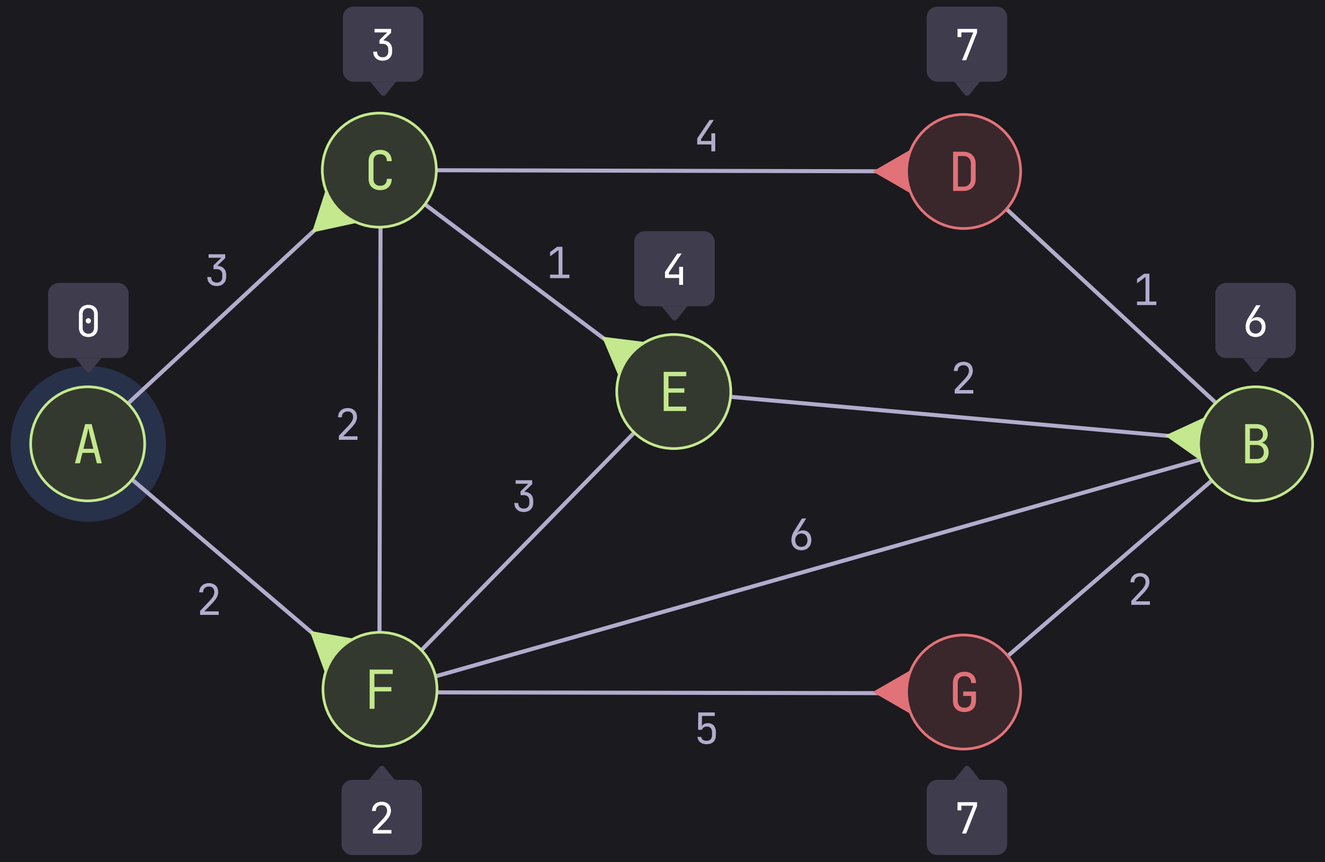
Now to get the path we just follow the arrows back to the start

Dijkstra's Algorithm



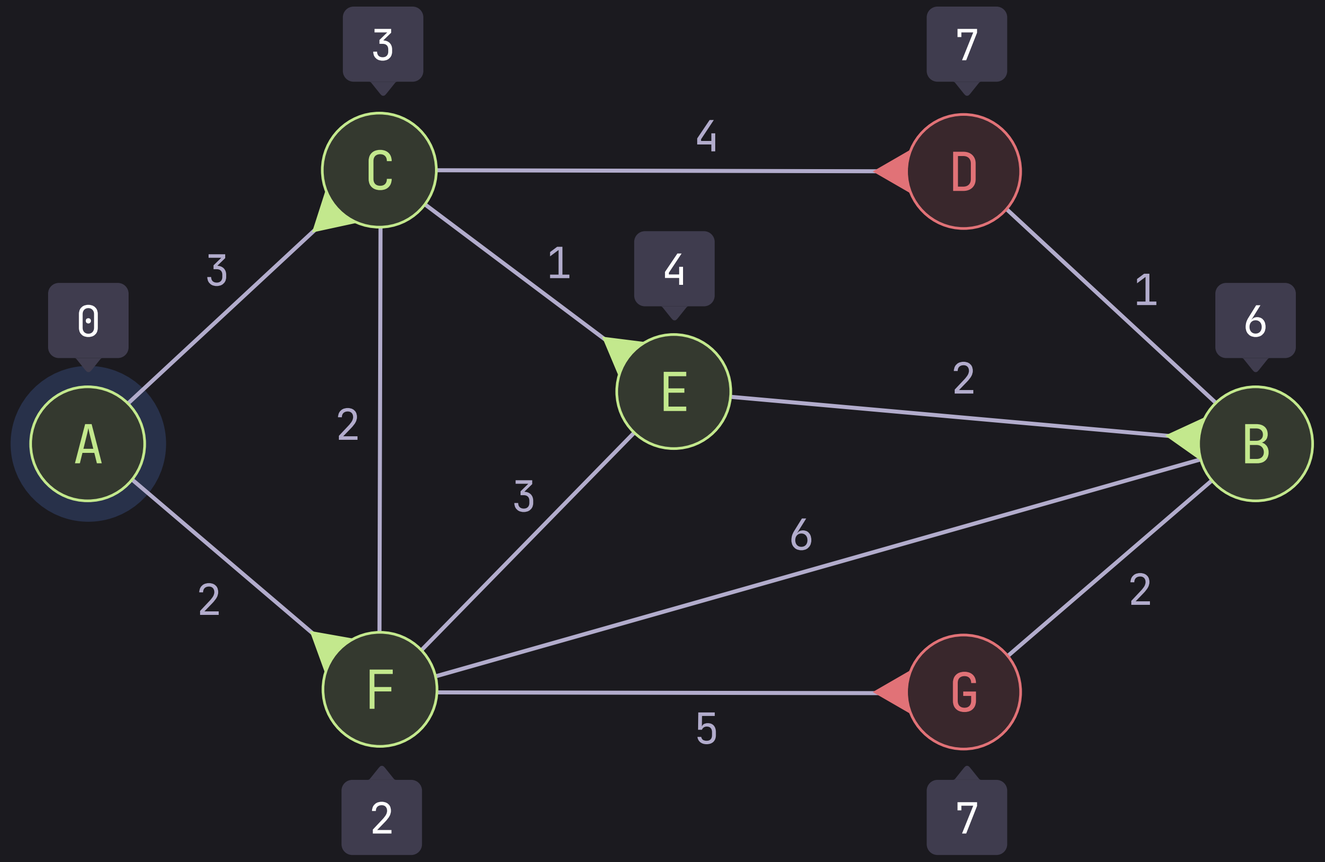
Now to get the path we just follow the arrows back to the start

Dijkstra's Algorithm



Now to get the path we just follow the arrows back to the start

Dijkstra’s Algorithm



Finally we reverse the order to get the path from A to B

Dijkstra’s Algorithm

How can we keep track of the visited vertex?

Dijkstra’s Algorithm

How can we keep track of the visited vertex?

- Either a set of visited nodes, or a vector of booleans

Dijkstra’s Algorithm

How can we keep track of the visited vertex?

- Either a set of visited nodes, or a vector of booleans

What data structure lets us efficiently pick the node with the shortest estimate?

Dijkstra’s Algorithm

How can we keep track of the visited vertex?

- Either a set of visited nodes, or a vector of booleans

What data structure lets us efficiently pick the node with the shortest estimate?

- Min priority queue

Dijkstra’s Algorithm

How can we keep track of the visited vertex?

- Either a set of visited nodes, or a vector of booleans

What data structure lets us efficiently pick the node with the shortest estimate?

- Min priority queue

How can we keep track of the previous vertex?

Dijkstra’s Algorithm

How can we keep track of the visited vertex?

- Either a set of visited nodes, or a vector of booleans

What data structure lets us efficiently pick the node with the shortest estimate?

- Min queue/heap

How can we keep track of the previous vertex?

- Hashmap (or an array if the vertices are just numbers)

Dijkstra’s Algorithm

How can we keep track of the visited vertex?

- Either a set of visited nodes, or a vector of booleans

What data structure lets us efficiently pick the node with the shortest estimate?

- Min priority queue

How can we keep track of the previous vertex?

- Hashmap (or an array if the vertices are just numbers)

What does this algorithm remind you of?

Dijkstra’s Algorithm

How can we keep track of the visited vertex?

- Either a set of visited nodes, or a vector of booleans

What data structure lets us efficiently pick the node with the shortest estimate?

- Min queue/heap

How can we keep track of the previous vertex?

- Hashmap (or an array if the vertices are just numbers)

What does this algorithm remind you of?

- Breadth First Search

```
function singleSourceShortestPaths(start, adjacencylist):
    minQueue = new MinQueue()
    visited = new Set()
    bestDistanceTo = new Vector<float>(adjacencylist.size(), infinity)

    bestDistanceTo[start] = 0
    minQueue.push((0, start));

    while not minQueue.empty():
        (distance, current) = minQueue.pop()
        if visited.contains(current):
            continue
        visited.insert(current)

        for (neighbour, weight) in adjacencylist[current]:
            distanceViaCurrent = bestDistanceTo[current] + weight
            if distanceViaCurrent < bestDistanceTo[neighbour]:
                bestDistanceTo[neighbour] = distanceViaCurrent
                minQueue.push((distanceViaCurrent, neighbour))

    return bestDistanceTo
```