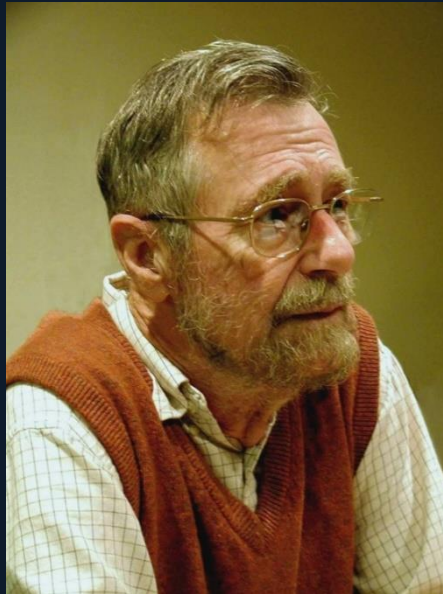


# Dijkstra's Shortest Path



# Lecture Contents

- Why do we need the shortest path?
- Graphs for Pathfinding
- Breadth First Search review
- Dijkstra's Algorithm

# Why do we need the shortest path?

- Many techniques require the shortest path in a graph, including finding the shortest path between two locations in a game level or finding the least amount of moves required in a board-game to win from the current arrangement of game pieces.

# Graphs For Pathfinding

- One of the most common uses for graphs in games is pathfinding.
  - Being able to calculate how to get from point A to B in a game level.
- Why not just move from A to B?
  - There could be obstacles or complex geometry blocking a straight line path between the two.

# An Example



- This is Steppes of War
  - A map in Starcraft 2

# An Example



- The walls are hard to see so I've drawn over them.

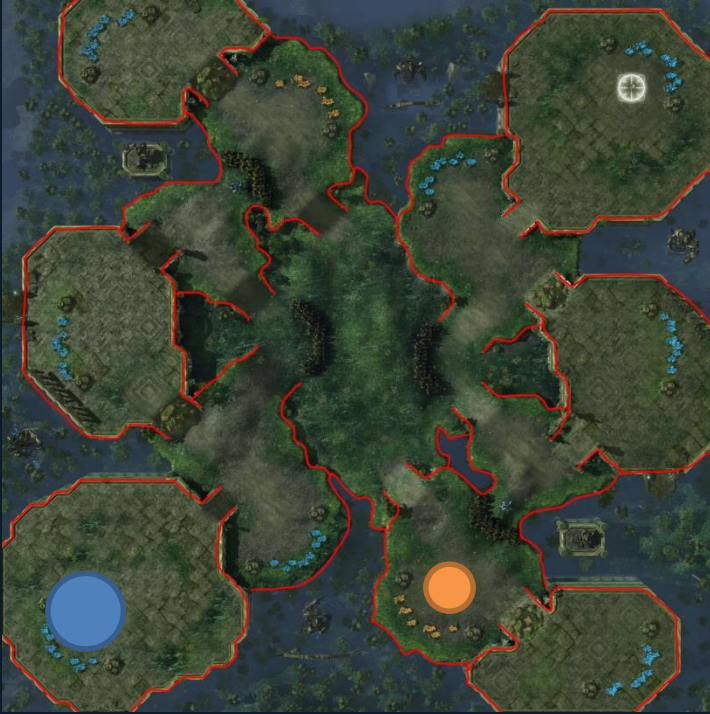


# An Example



- The walls are hard to see so I've drawn over them.
- Lets say the player selected a unit here.

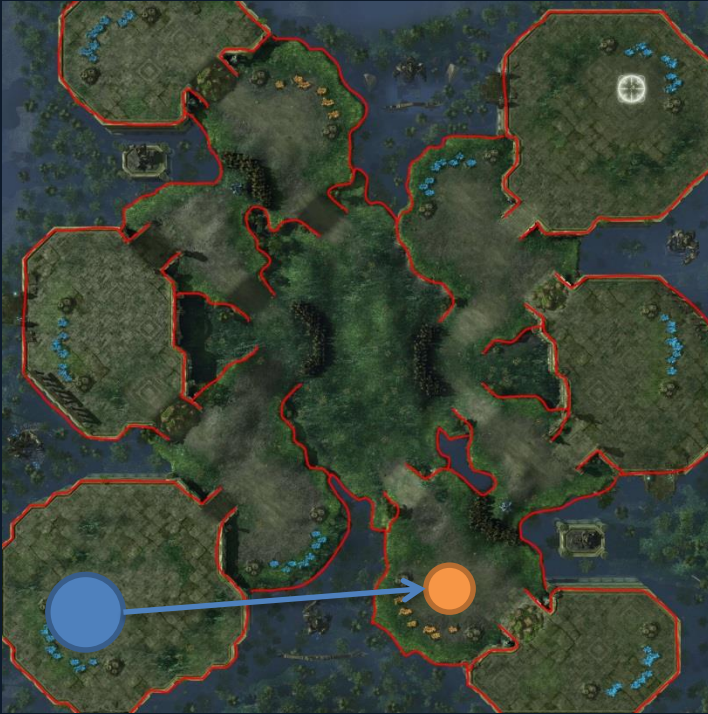
# An Example



- The walls are hard to see so I've drawn over them.
- Lets say the player selected a unit here.
- And told them to move over here.



# An Example



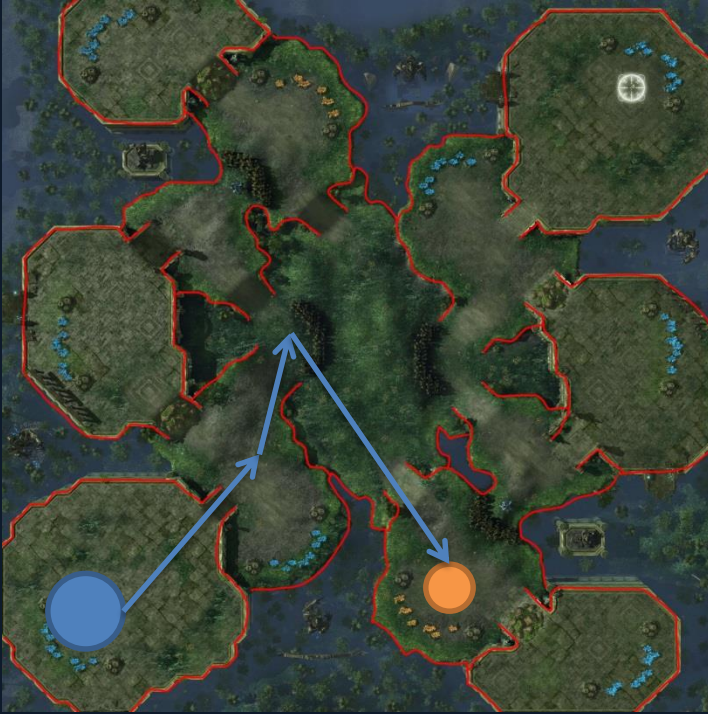
- If we just told the unit to move towards the goal.

# An Example



- It would run into the wall here.

# An Example



- The correct path looks more like this.

# Graphs For Pathfinding

- We solve the problem by filling the map with nodes.
  - Nodes represent a position and are only connected to another node if there is a straight line path between them.
- To find the path, we now need to find the shortest path, along the edges from the starting node to the goal node.

# Breadth First Search Review

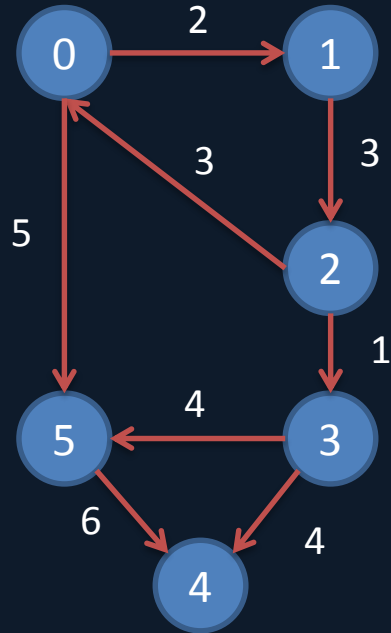
- Breadth First Search (BFS) fans out from the starting node in all directions until all connected nodes are processed.
- Uses a queue to generate the order of the nodes it traverses.

# Dijkstra's Algorithm

- Dijkstra's Algorithm is a modification of BFS that allows us to find the shortest distance from any two nodes on a weighted graph.
  - For an unweighted graph, you can just treat all edges as having a weight of 1.
- It makes two changes from vanilla BFS.
  - It adds a running count of the distance from the start to each node it traverses.
  - It uses a priority queue to pick the next node.

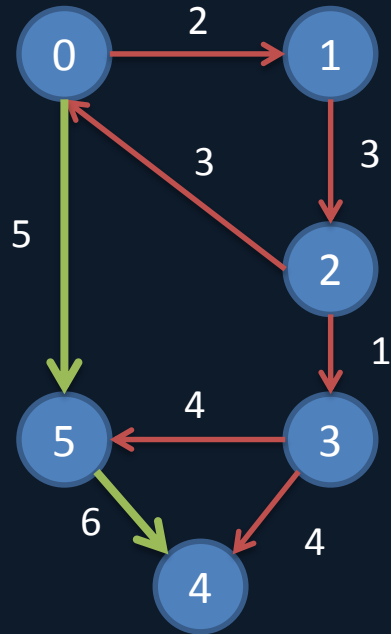


# Dijkstra's Algorithm



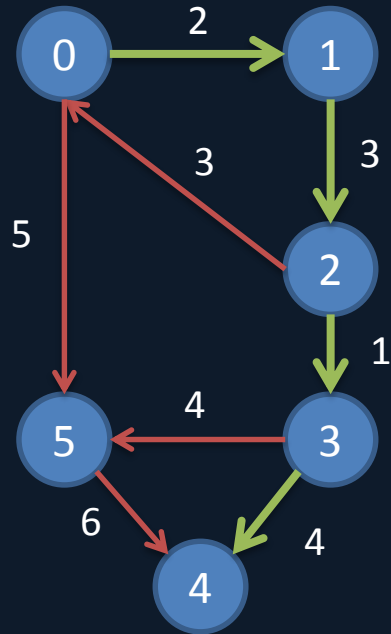
- Given this weighted graph, what paths can get us from node 0 to node 4?
- And from those paths which is the shortest?
- Just looking at the graph, we can tell that there are 2 main options

# Dijkstra's Algorithm



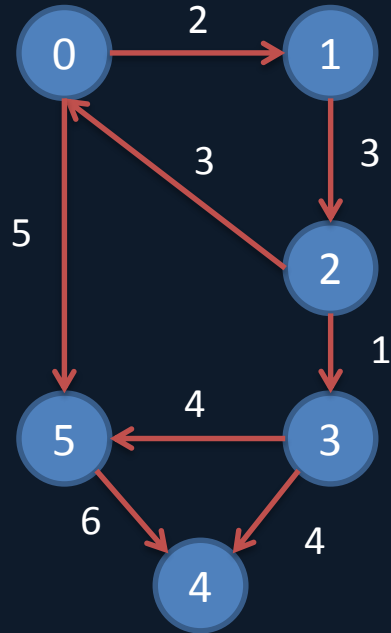
- Given this weighted graph, what paths can get us from node 0 to node 4?
- And from those paths which is the shortest?
- Just looking at the graph, we can tell that there are 2 main options

# Dijkstra's Algorithm



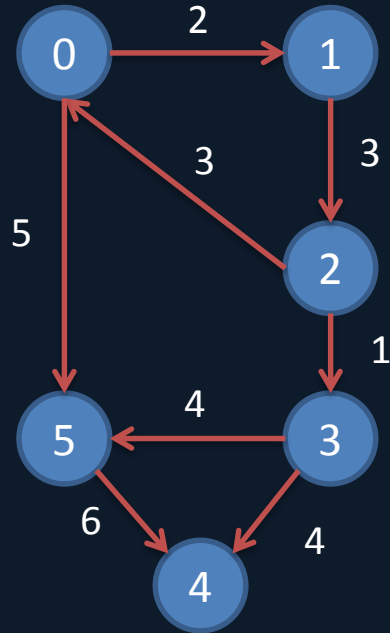
- Given this weighted graph, what paths can get us from node 0 to node 4?
- And from those paths which is the shortest?
- Just looking at the graph, we can tell that there are 2 main options

# Dijkstra's Algorithm



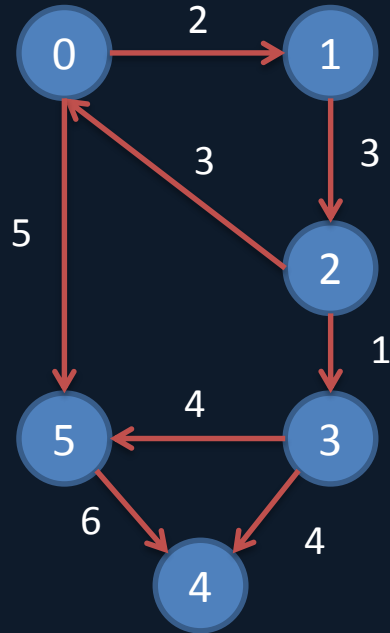
- The first path 0->5->4 costs a total of
  - $6 + 5 = 11$
- The second path 0->1->2->3->4 costs
  - $2 + 3 + 1 + 4 = 10$
- So the second path is the fastest, even though it takes us through more nodes

# Dijkstra's Algorithm



- So how do we get a computer to figure this out?

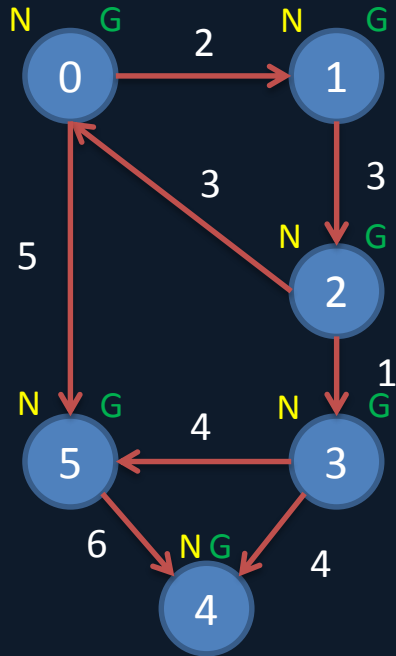
# Dijkstra's Algorithm



- First we need to add two new pieces of information to each node.
  - The previous node we came from
  - The “G score”, or total traversal cost to get to this node

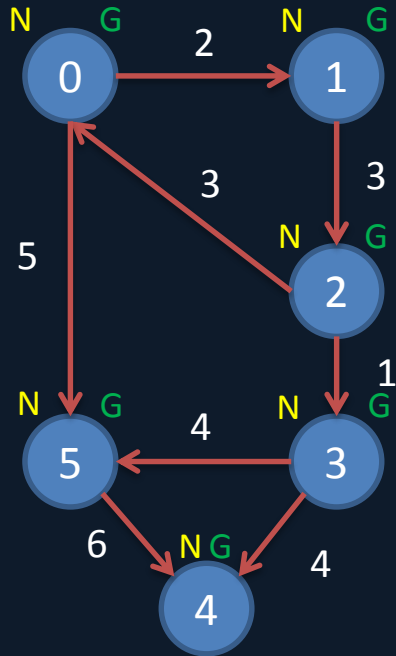


# Dijkstra's Algorithm



- First we need to add two new pieces of information to each node.
  - The previous node we came from
  - The “G score”, or total traversal cost to get to this node

# Dijkstra's Algorithm

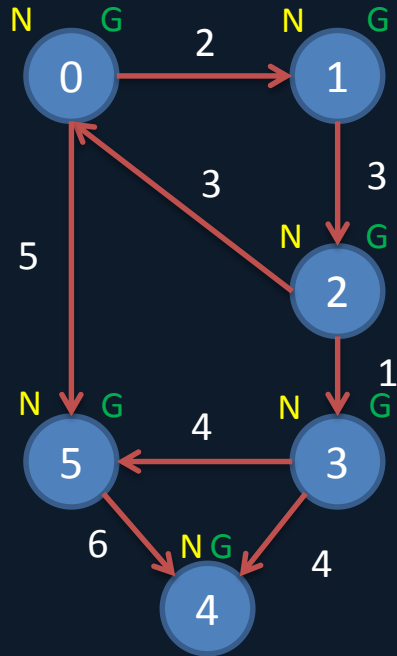


- I mentioned a priority queue before.
- With the regular queue in BFS, when getting the top node it will return the one pushed first.
- The priority queue will return the node with the lowest G-score.

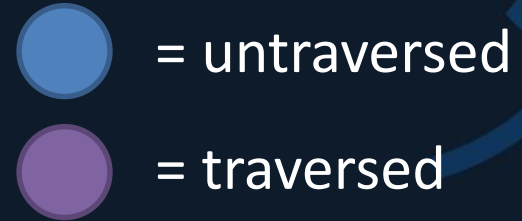
# Dijkstra's Algorithm

- Dijkstra's algorithm is:
  - Set all **N**s to null, set all **G**s to infinity
  - Push start node onto priority queue, set its **N** to itself and its **G** to 0
  - While queue not empty
    - Get the current node off the end of the queue and remove it.
    - Mark it as traversed
    - Loop through its edges
      - If end node not traversed
        - » Calculate current node's **G** + the edge cost
        - » If cost is less than existing **G** cost in end node
          - Set end node's **N** to the current node
          - Set end node's **G** to the current node's **G** + the edge cost
          - If end node not in the queue
            - Push end node onto the queue

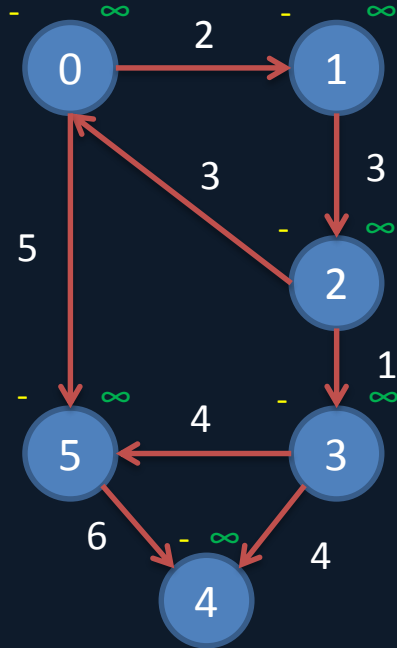
# Dijkstra's Algorithm





Priority  
Queue:



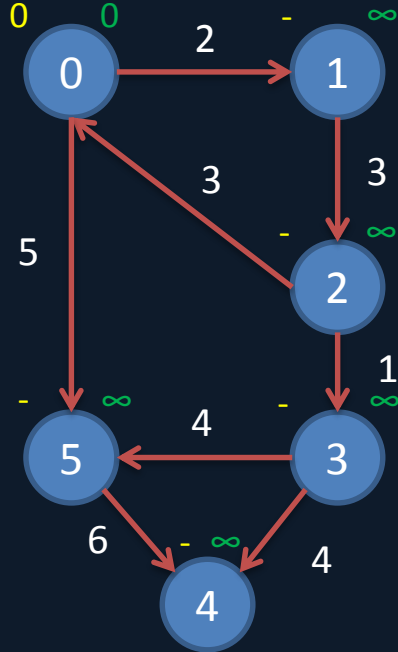
# Dijkstra's Algorithm



Priority  
Queue:



 = untraversed  
 = traversed

# Dijkstra's Algorithm



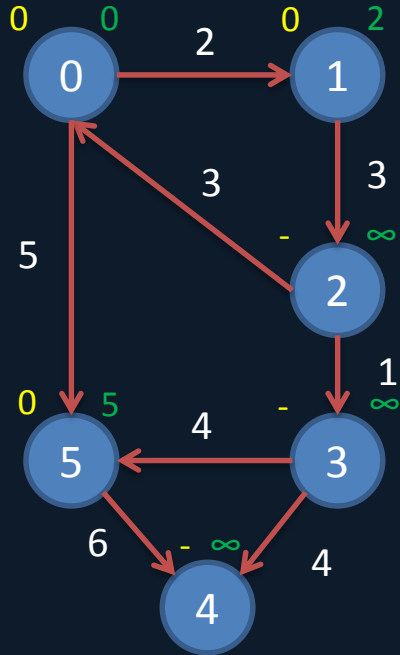
Priority  
Queue:

0

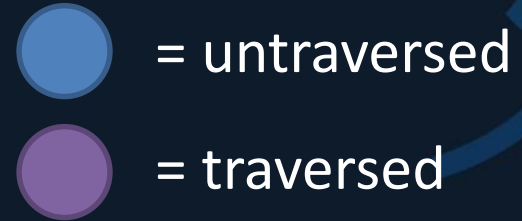
 = untraversed  
 = traversed



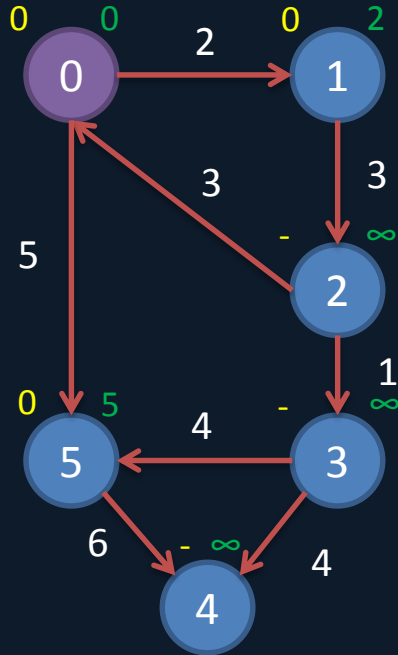
# Dijkstra's Algorithm



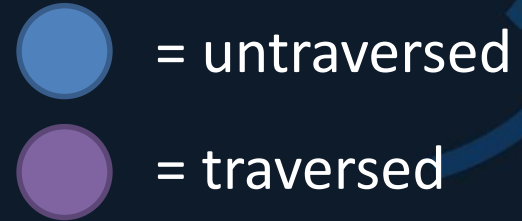
Priority  
Queue:



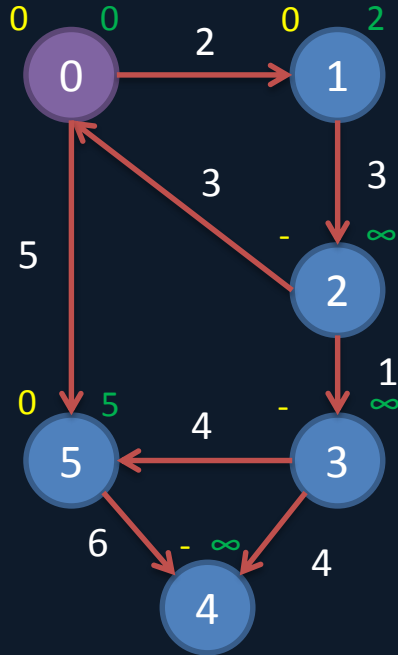
# Dijkstra's Algorithm



Priority  
Queue:





# Dijkstra's Algorithm

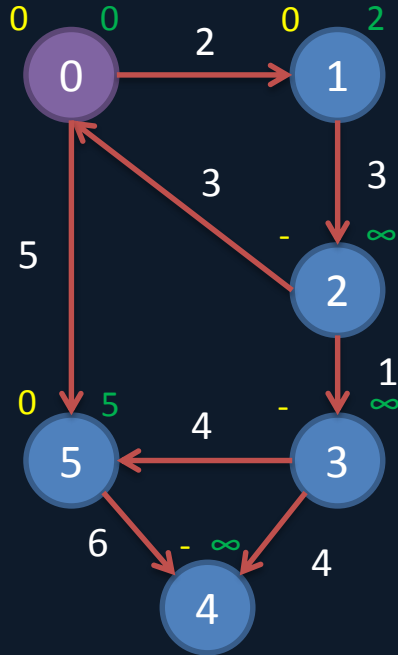


Priority  
Queue:

1



 = untraversed  
 = traversed

# Dijkstra's Algorithm

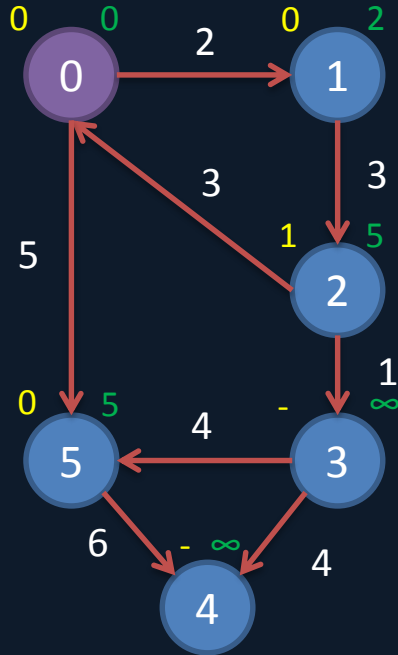


Priority  
Queue:

1
5

 = untraversed  
 = traversed

# Dijkstra's Algorithm

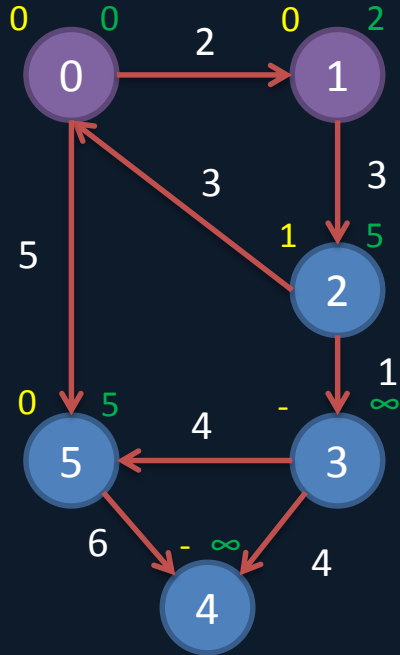


Priority  
Queue:

5

● = untraversed  
● = traversed

# Dijkstra's Algorithm



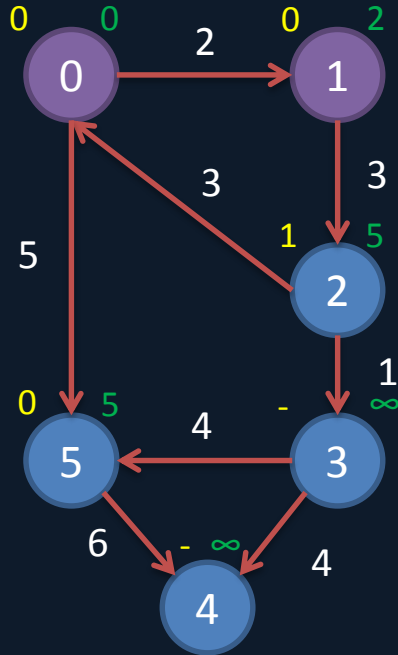
Priority  
Queue:

5

● = untraversed  
● = traversed



# Dijkstra's Algorithm

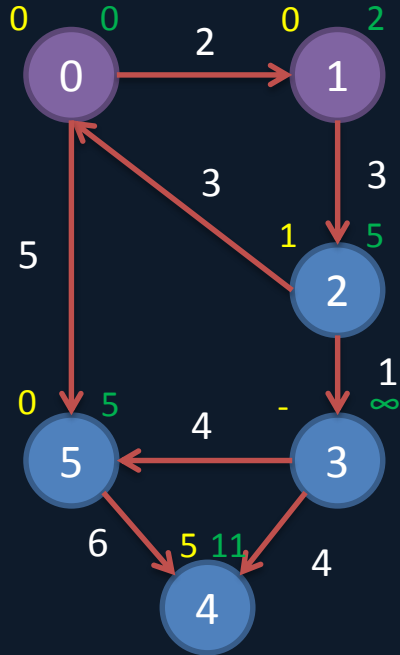


Priority  
Queue:

5
2

● = untraversed  
● = traversed

# Dijkstra's Algorithm

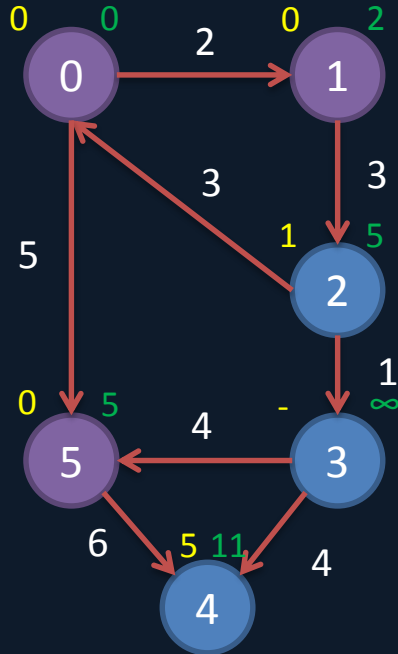


Priority  
Queue:

2

● = untraversed  
● = traversed

# Dijkstra's Algorithm

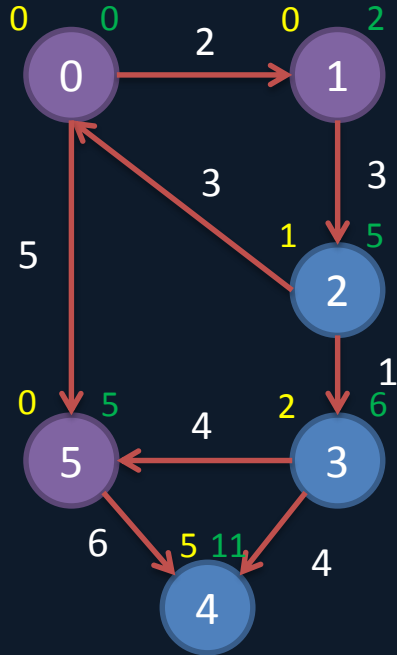


Priority  
Queue:

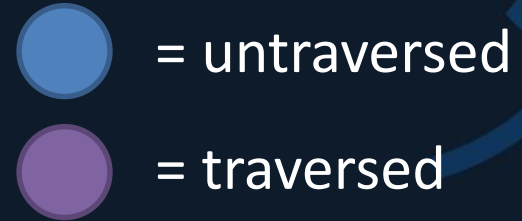
2

● = untraversed  
● = traversed

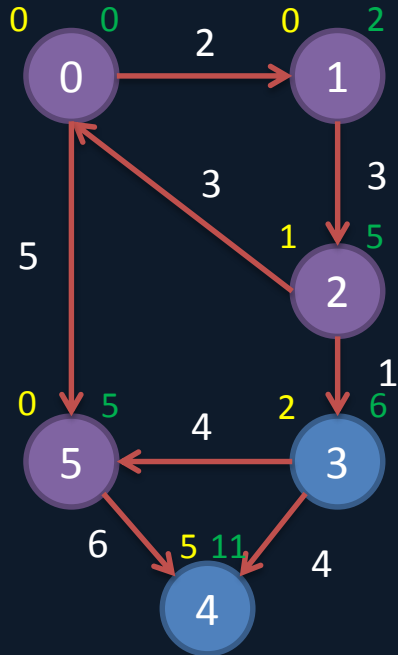
# Dijkstra's Algorithm



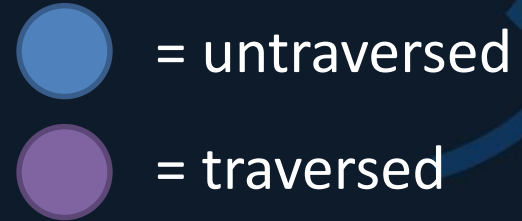
Priority  
Queue:



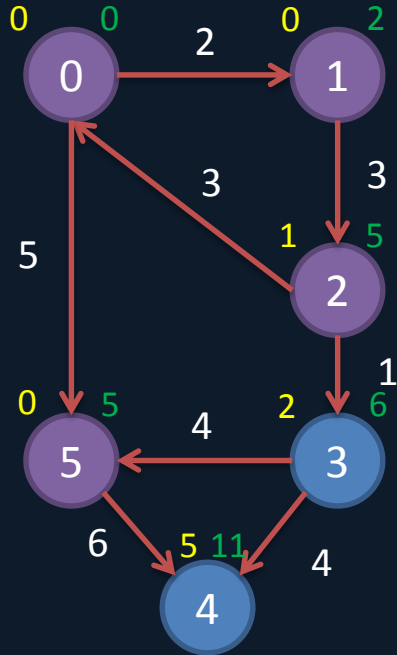
# Dijkstra's Algorithm



Priority  
Queue:



# Dijkstra's Algorithm

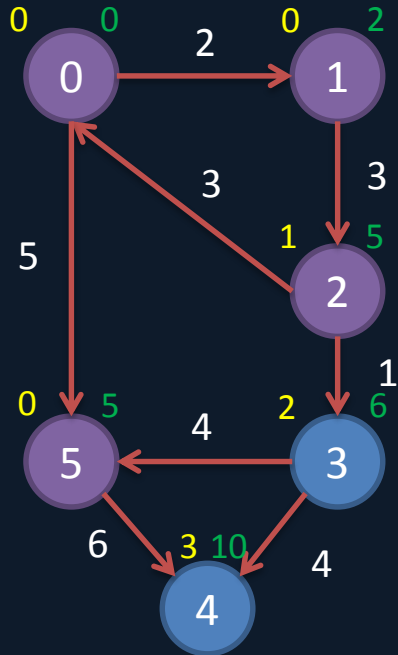


Priority  
Queue:



3

● = untraversed  
● = traversed

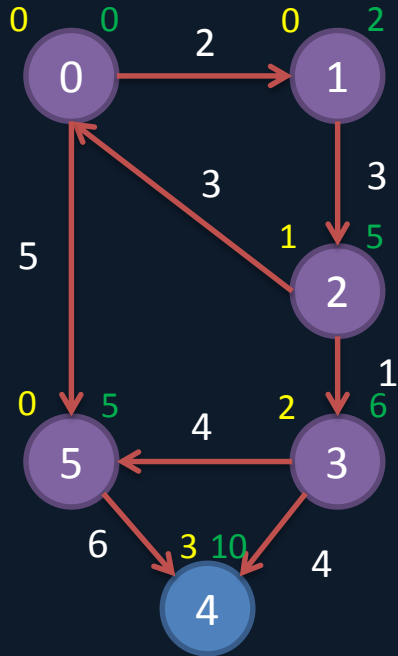
# Dijkstra's Algorithm





Priority  
Queue:

 = untraversed  
 = traversed

# Dijkstra's Algorithm

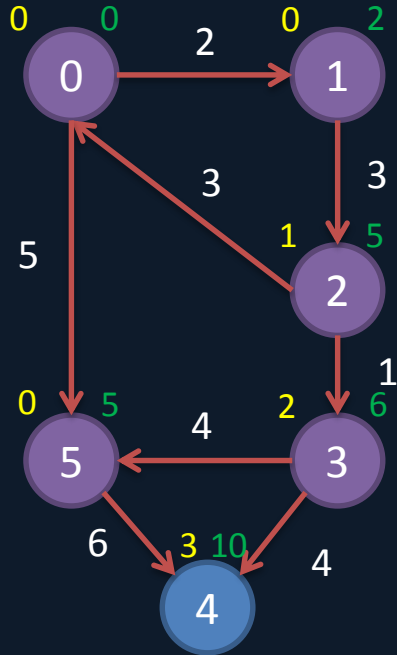


Priority  
Queue:

 = untraversed  
 = traversed



# Dijkstra's Algorithm

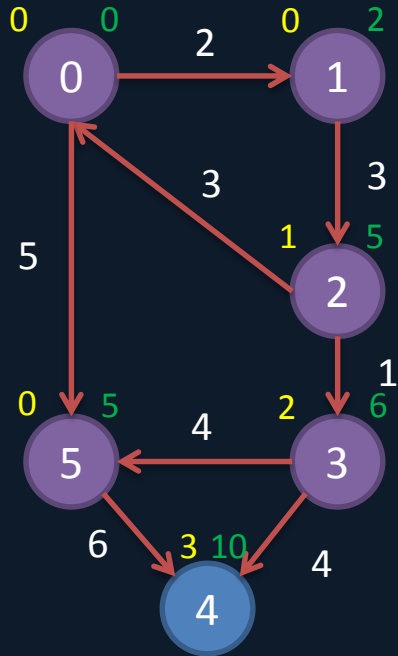


Priority  
Queue:



4

● = untraversed  
● = traversed

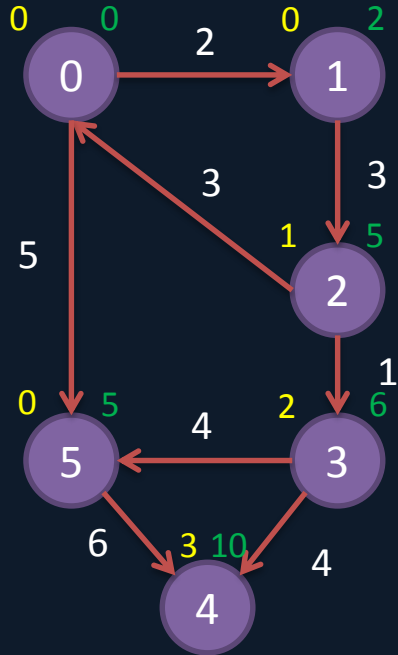
# Dijkstra's Algorithm



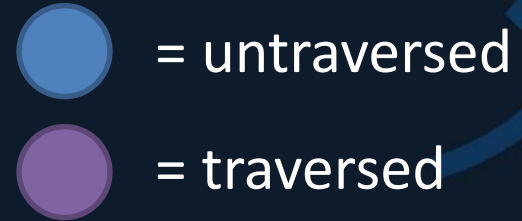
Priority  
Queue:

 = untraversed  
 = traversed

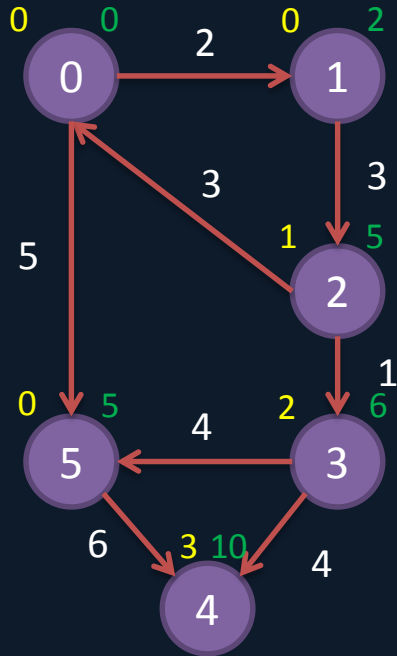
# Dijkstra's Algorithm



Priority  
Queue:



# Dijkstra's Algorithm



- Now that the algorithm has run, if we start at any node (not just 4) and follow the previous pointers back, we get the shortest path from that node back to node 0.
- Starting at 4, it points back to 3 then to 2, then 1 and finally 0.
- If we reverse this list we get
  - 0->1->2->3->4
  - The shortest path!

# Dijkstra's Algorithm

- Its important to note that Dijkstra's does calculate the shortest path from the start node to *every other node in the graph*.
- This means for small graphs, we could just pre-calculate Dijkstra's for every node and we would know the shortest path from every node to every other node.
  - For anything other than small graphs, this takes a prohibitively large amount of memory to store all the results.
  - It also means you can't dynamically change the map, or the weights without recalculating every node again.
- Dijkstra's is also quite expensive to run for larger graphs.
  - To solve all of these problems we turn to a new algorithm, called A\*

# Questions?

