# Introduction to Pathfinding II

## Dijkstra's Algorithm

Ryan Toner

# Intelligent Graph Pathfinding - Learning Objective

- We are now familiar with BFS and DFS
  - These are both unintelligent searching algorithms, as all edges are treated equally
- We want an *intelligent search* that supports weighted graphs
  - Real systems have "distances" or "costs" associated with traveling from node to node
  - Useful for true pathfinding in games, robotics, maps, etc
- Define a weighted set for graph G = {V(G), E(G)} in the following way
  - $W(G) = \{w_1, w_2 \ldots w_m\}$, where $W_i$ indicates the cost of traversing edge $e_i$
- The objective of this lecture is to introduce Dijkstra's algorithm for finding the optimal paths in a weighted graph

# Graph Search - Dijkstra’s Algorithm

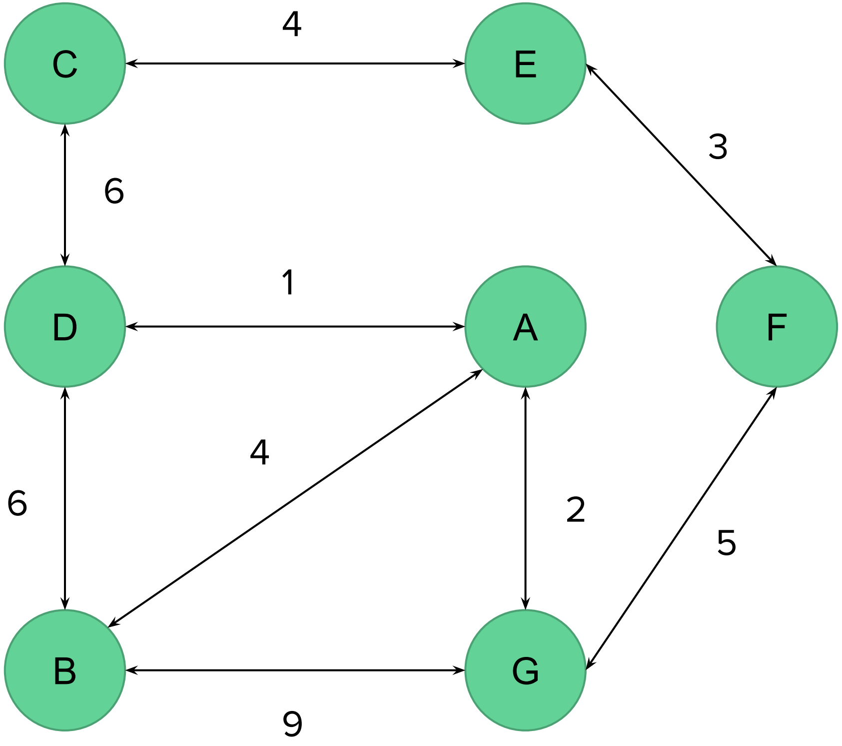
- This search will take a source node (root) and will find the optimal path to an end node (terminus).
- This search can easily be extended to find all possible paths in the graph starting from the start (simply by not specifying an ending node). This is called single-source shortest path.

High-Level Dijkstra’s Algorithm:

1. Push root node onto Priority Queue
2. Dequeue the top node (node with the lowest path total)
   a. If the top node is marked, ignore it and repeat step 2
3. Mark the top node
4. Enqueue all unmarked neighbors and calculate a new path cost for each neighbor
5. Repeat step 2-5, until queue is empty or terminus point is found

The path cost is calculated by taking the current path weight of the top node (line 2) + the edge weight connecting the top node to each neighbor

# Example Graph




- Here is our example undirected graph.
- $V(G) = \{A,B,C,D,E,F,G\}$
- $E(G) = \{$

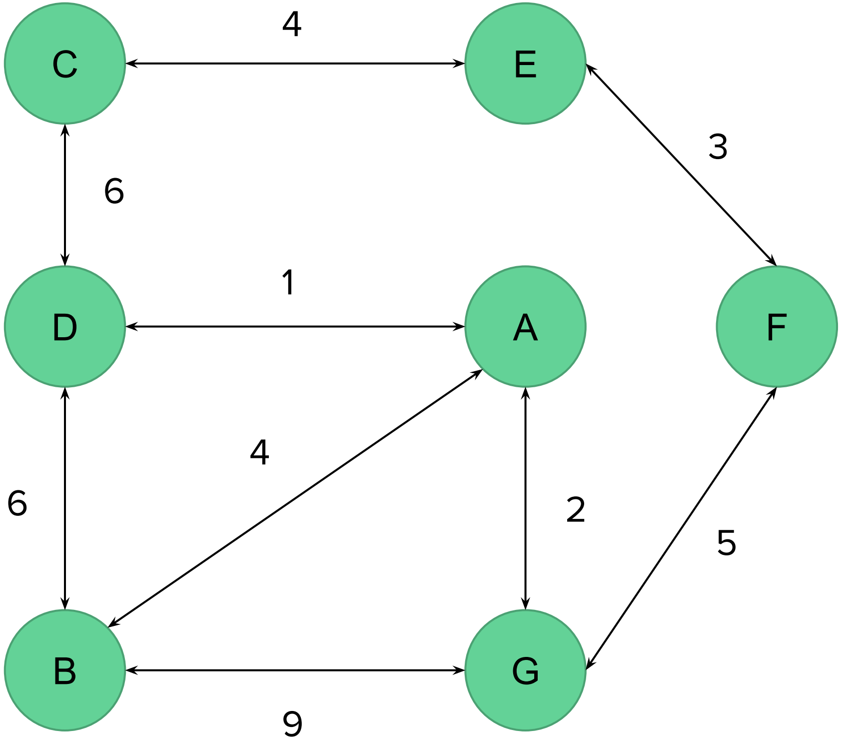
$$
\begin{aligned}
e_1 &= (A,B), & w_1 &= 4\\
e_2 &= (A,D), & w_2 &= 1\\
e_3 &= (A,G), & w_3 &= 2\\
e_4 &= (B,D), & w_4 &= 6\\
e_5 &= (B,G), & w_5 &= 9\\
e_6 &= (C,D), & w_6 &= 6\\
e_7 &= (C,E), & w_7 &= 4\\
e_8 &= (E,F), & w_8 &= 3\\
e_9 &= (F,G), & w_9 &= 5
\end{aligned}
$$

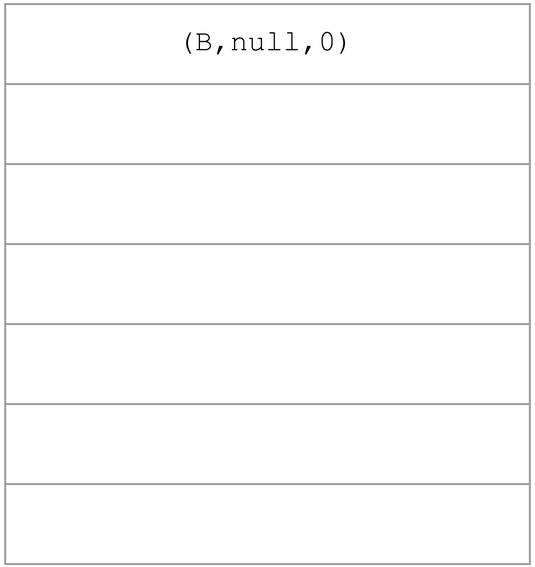
$\}$

- We will compute the shortest path from B to E using Dijkstra's Algorithm.

# Dijkstra's Algorithm

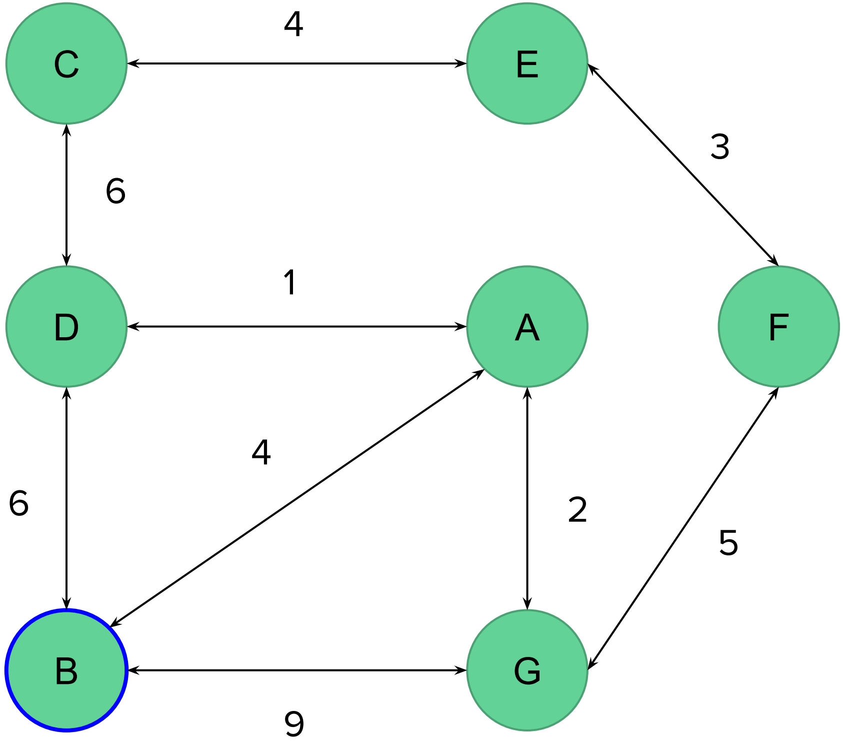
- Our Priority Queue will store tuples of the form (node, path, weight)
- Add (B, null, 0) to the Queue




Priority Queue

# Dijkstra's Algorithm




1. Dequeue (B, null, 0)
2. Mark B
3. Process the unmarked neighbors of B:
   a. Enqueue (D, {B},6)
   b. Enqueue (A, {B},4)
   c. Enqueue (G,{B},8)

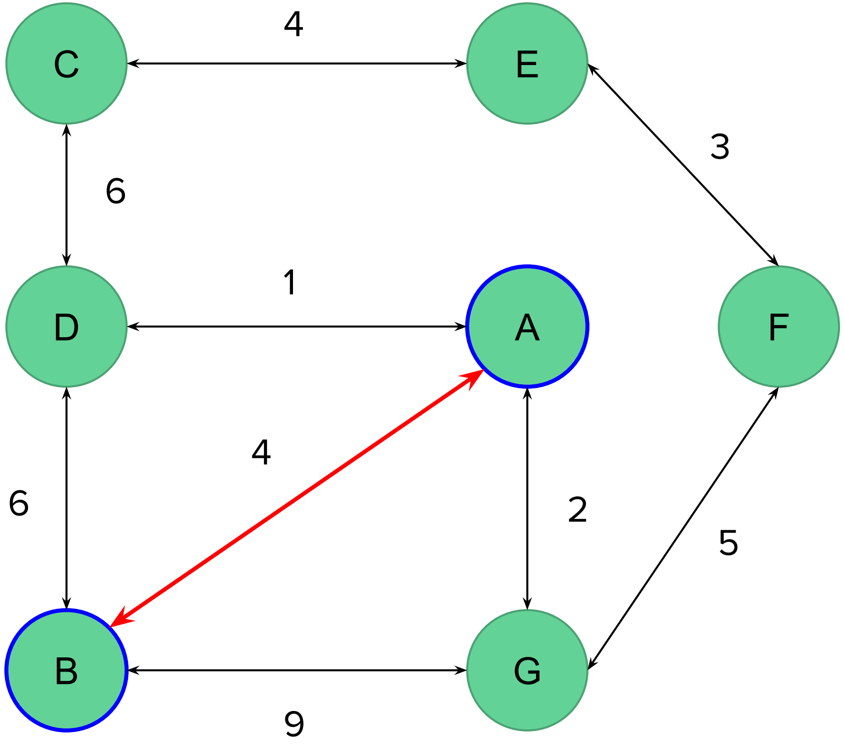
*Notice that the PriorityQueue is sorting the elements by lowest weight, since lower weight indicates higher priority!

| |
|---|
| (A,{B},4) |
| (D,{B},6) |
| (G,{B},9) |
| |
| |
| |
| |

Priority Queue

# Dijkstra's Algorithm

1. Dequeue (A, {B},4)
2. Mark A
3. Process the unmarked neighbors of A:
   a. Enqueue (D, {B,A},5)
   b. Enqueue (G, {B,A},6)

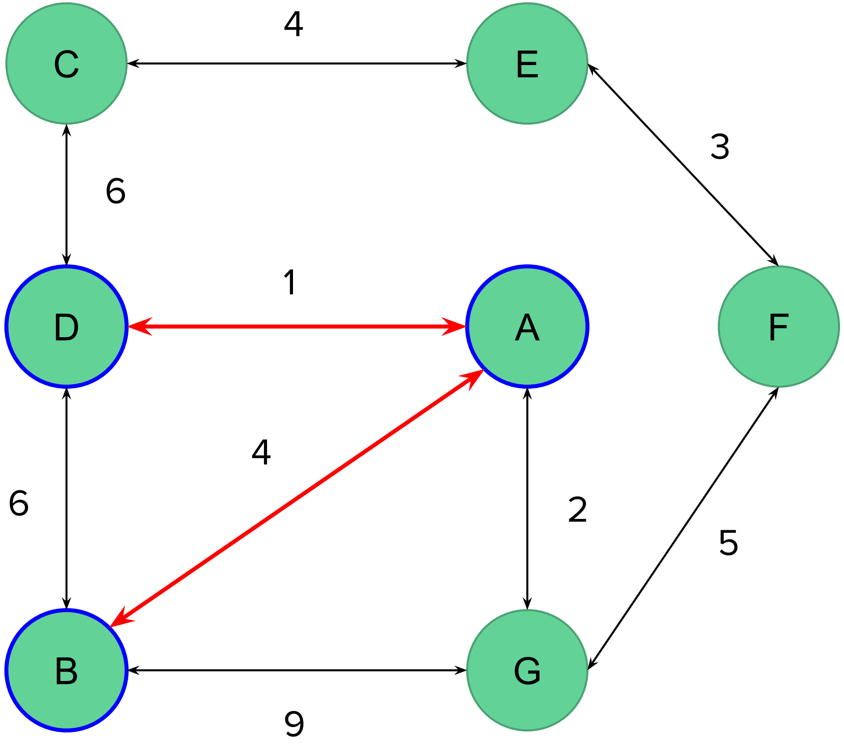



| |
|---|
| (D,{B,A},5) |
| (G,{B,A},6) |
| (D,{B},6) |
| (G,{B},8) |
| |
| |
| |

Priority Queue

# Dijkstra's Algorithm

1. Dequeue (D, {B,A},5)
2. Mark D
3. Process the unmarked neighbors of D:
    a. Enqueue (C, {B,A,D},11)

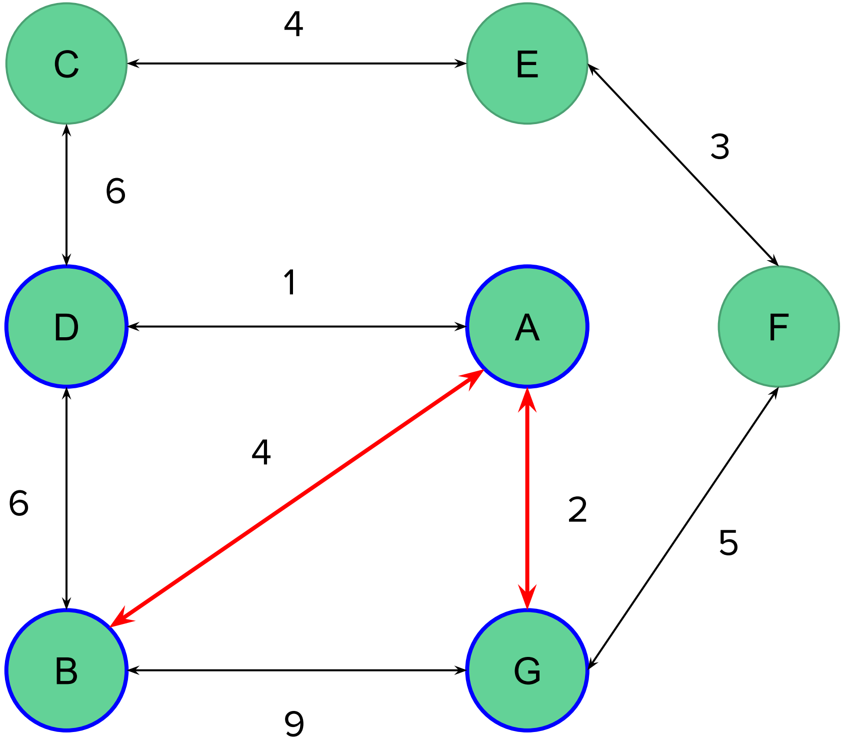



| |
|---|
| (G,{B,A},6) |
| (D,{B},6) |
| (G,{B},8) |
| (C,{B,A,D},11) |
| |
| |
| |

Priority Queue

# Dijkstra's Algorithm

1. Dequeue (G, {B,A},6)
2. Mark G
3. Process the unmarked neighbors of G:
   a. Enqueue (F, {B,A,G},11)

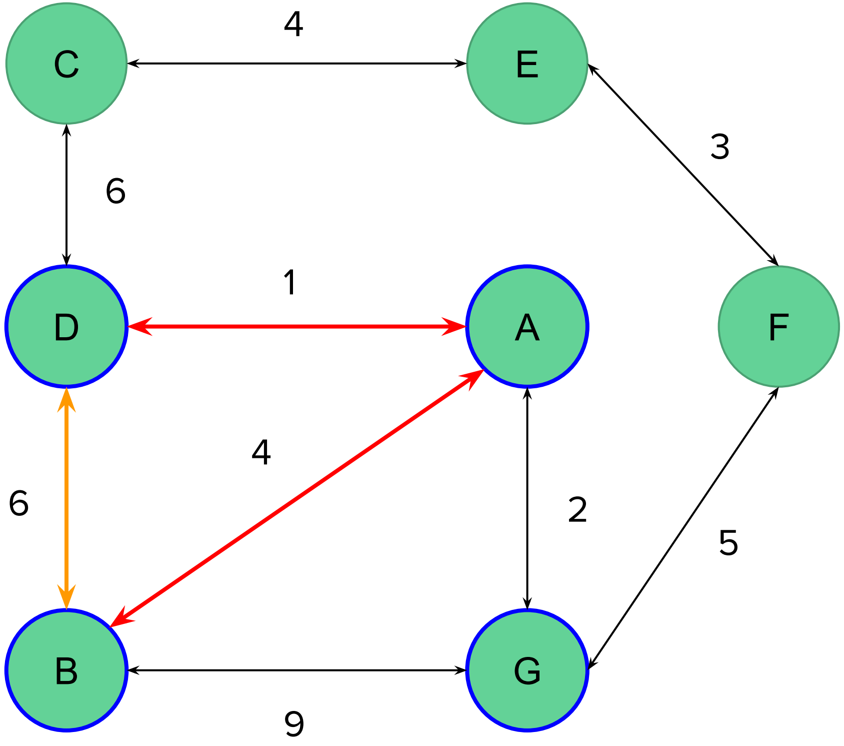



| |
|---|
| (D,{B},6) |
| (G,{B},8) |
| (F,{B,A,G},11) |
| (C,{B,A,D},11) |
| |
| |
| |

Priority Queue

# Dijkstra's Algorithm

1. Dequeue (D, {B},6)
2. D is already marked, meaning a shorter path has already been found. Discard this value.
    a. As you can see, the red path (5) processed D first, since its weight is less than the orange path, which has a weight of 6.

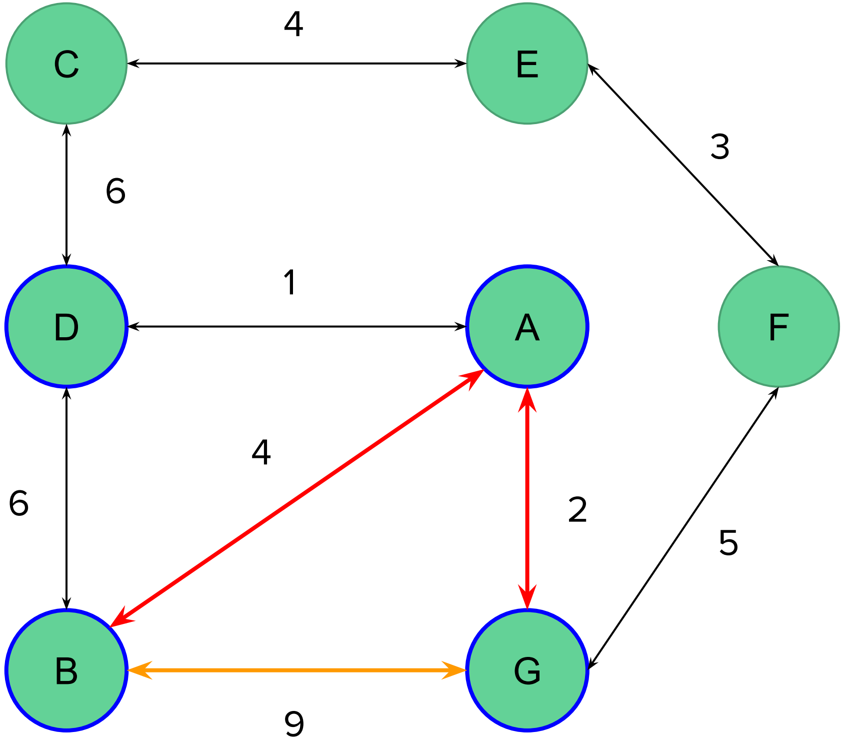



| |
|---|
| (G,{B},9) |
| (F,{B,A,G},11) |
| (C,{B,A,D},11) |
| |
| |
| |
| |

Priority Queue

# Dijkstra's Algorithm

1. Dequeue (G,{B},9)
2. G is already marked, meaning a shorter path has already been found. Discard this value.
    a. The red path has weight (6), less than the orange path weight (9)

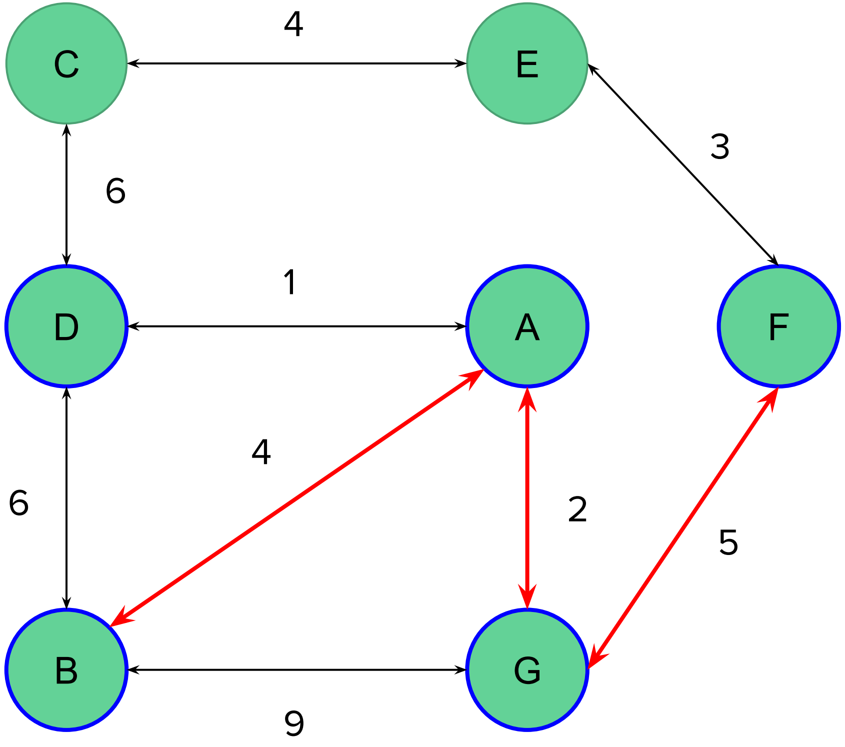



| |
|---|
| (F,{B,A,G},11) |
| (C,{B,A,C},11) |
| |
| |
| |
| |
| |

Priority Queue

# Dijkstra's Algorithm

1. Dequeue (F, {B,A,G},11)
2. Mark F
3. Process the unmarked neighbors of F:
   a. Enqueue (E, {B,A,G,F},14)

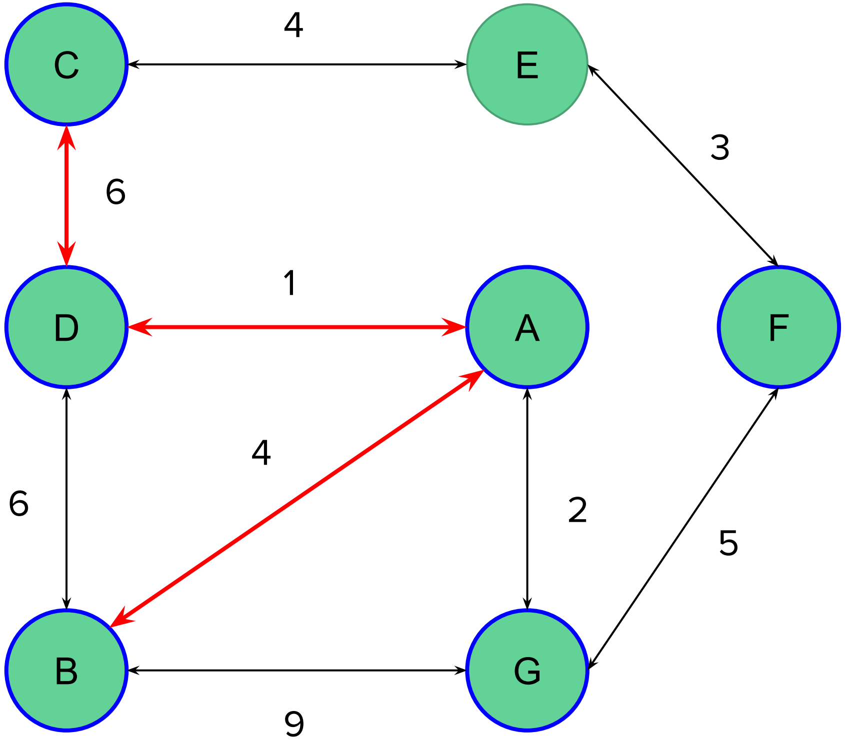



| |
|---|
| (C,{B,A,C},11) |
| (E,{B,A,G,F},14) |
| |
| |
| |
| |
| |

Priority Queue

# Dijkstra's Algorithm

1. Dequeue (C, {B,A,D},11)
2. Mark C
3. Process the unmarked neighbors of C:
    a. Enqueue (E, {B,A,D,C},15)

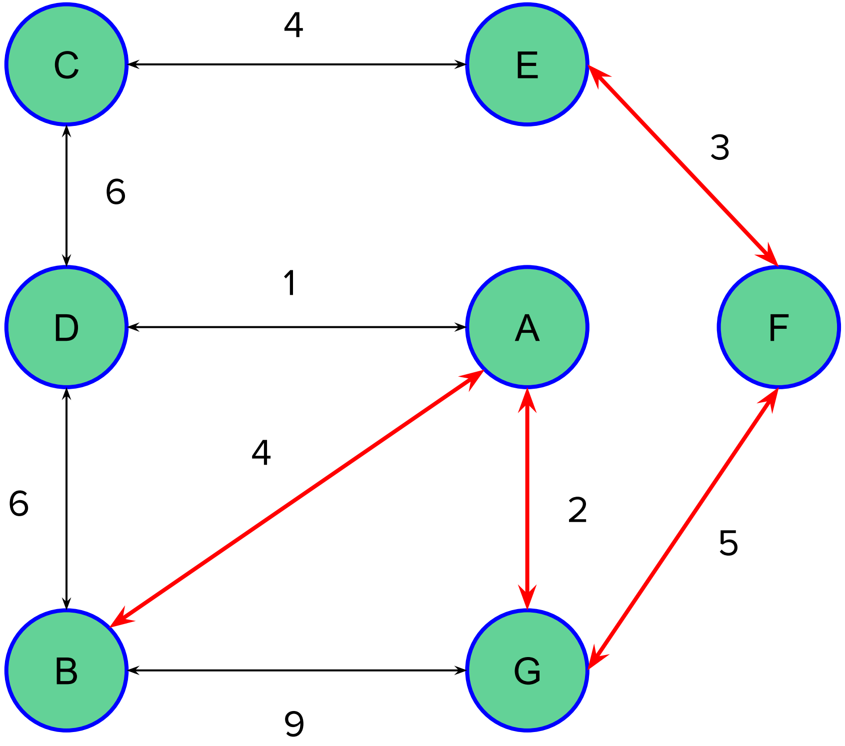



| |
|---|
| (E,{B,A,G,F},14) |
| (E,{B,A,D,C},15) |
| |
| |
| |
| |
| |

Priority Queue

# Dijkstra's Algorithm

1. Dequeue (E, {B,A,G,F},14)
2. Mark E
3. We have found our shortest path from B to E!
4. The final path is {B,A,G,F,E} with total path cost 14




| |
|---|
| (E,{B,A,D,C},15) |
| |
| |
| |
| |
| |
| |

Priority Queue

# Overview - Dijkstra's Algorithm

- Dijkstra's Algorithm is a true intelligent ("informed") search
  - Works on both directed and undirected graphs
  - Always picks to work with the best path by nature of using a Priority Queue
- Always returns the shortest path (as long as it exists)
- Also called Uniform-Cost search
  - *this **does not** mean all weights are equal!
  - It *does mean* that it only calculates the cost for edges that have been found
    - And the algorithm uses a marked set to never recalculate optimized paths
- Consider using Dijkstra's algorithm to solve grid puzzles like mazes
  - BFS may perform similarly, but cannot be generalized to weighted graphs
  - DFS is uninformed and cannot consider weights ➔ suboptimal solutions
  - Dijkstra gets us **optimal** solutions **fast** in any graph structure!

# Thinking about BFS again - BFS ⊂ Dijkstra

- BFS can be thought of as a subset of Dijkstra's Algorithm
  - BFS is simply Dijkstra’s algorithm where all edge weights are equal!
- Consider which algorithm is appropriate for each scenario:
  - Weighted graphs → use Dijkstra
  - Unweighted graphs → use BFS
  - Find all nodes reachable from a source node → use BFS
  - Find optimal paths to all other nodes from a source node → use Dijkstra