Problem: Find the shortest path between two nodes in a graph.

Constraints

- Is this a directional graph?
  - Yes
- Could the graph have cycles?
  - Yes
  - Note: If the answer were no, this would be a DAG.
    - DAGs can be solved with a [topological sort]
- Are the edges weighted?
  - Yes
  - Note: If the edges were not weighted, we could do a BFS
- Are the edges all non-negative numbers?
  - Yes
  - Note: Graphs with negative edges can be done with Bellman-Ford
    - Graphs with negative cost cycles do not have a defined shortest path
- Do we have to check for non-negative edges?
  - No
- Can we assume this is a connected graph?
  - Yes
- Can we assume the inputs are valid?
  - No
- Can we assume we already have a graph class?
  - Yes
- Can we assume we already have a priority queue class?
  - Yes
- Can we assume this fits memory?
  - Yes

Test Cases

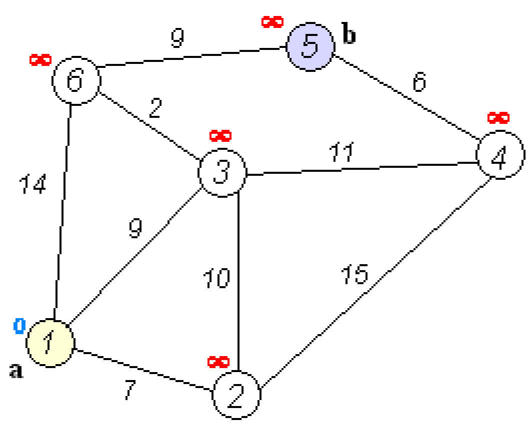
The constraints state we don't have to check for negative edges, so we test with the general case.

- graph.add_edge('a', 'b', weight=5)
- graph.add_edge('a', 'c', weight=3)
- graph.add_edge('a', 'e', weight=2)
- graph.add_edge('b', 'd', weight=2)
- graph.add_edge('c', 'b', weight=1)
- graph.add_edge('c', 'd', weight=1)
- graph.add_edge('d', 'a', weight=1)
- graph.add_edge('d', 'g', weight=2)
- graph.add_edge('d', 'h', weight=1)

- graph.add_edge('e', 'a', weight=1)
- graph.add_edge('e', 'h', weight=4)
- graph.add_edge('e', 'i', weight=7)
- graph.add_edge('f', 'b', weight=3)
- graph.add_edge('f', 'g', weight=1)
- graph.add_edge('g', 'c', weight=3)
- graph.add_edge('g', 'i', weight=2)
- graph.add_edge('h', 'c', weight=2)
- graph.add_edge('h', 'f', weight=2)
- graph.add_edge('h', 'g', weight=2)
- shortest_path = ShortestPath(graph)
- result = shortest_path.find_shortest_path('a', 'i')
- self.assertEqual(result, ['a', 'c', 'd', 'g', 'i'])
- self.assertEqual(shortest_path.path_weight['i'], 8)

Algorithm

Wikipedia's animation:



Initialize the following:

- previous = {} # Key: node key, val: prev node key, shortest path
    - Set each node's previous node key to None
- path_weight = {} # Key: node key, val: weight, shortest path
    - Set each node's shortest path weight to infinity
- remaining = PriorityQueue() # Queue of node key, path weight
    - Add each node's shortest path weight to the priority queue
- Set the start node's path_weight to 0 and update the value in remaining
- Loop while remaining still has items
    - Extract the min node (node with minimum path weight) from remaining
    - Loop through each adjacent node in the min node
        - Calculate the new weight:

        - **Adjacent node's edge weight + the min node's path_weight**
    - **If the newly calculated path is less than the adjacent node's current path_weight:**
        - **Set the node's previous node key leading to the shortest path**
        - **Update the adjacent node's shortest path and update the value in the priority queue**
- **Walk backwards to determine the shortest path:**
    - **Start at the end node, walk the previous dict to get to the start node**
- **Reverse the list and return it**

Complexity for array-based priority queue:

- **Time: O(v^2), where v is the number of vertices**
- **Space: O(v^2)**

This might be better than the min-heap-based variant if the graph has a lot of edges.

O(v^2) is better than O((v + v^2) log v).

Complexity for min-heap-based priority queue:

- **Time: O((v + e) log v), where v is the number of vertices, e is the number of edges**
- **Space: O((v + e) log v)**

This might be better than the array-based variant if the graph is sparse.

-