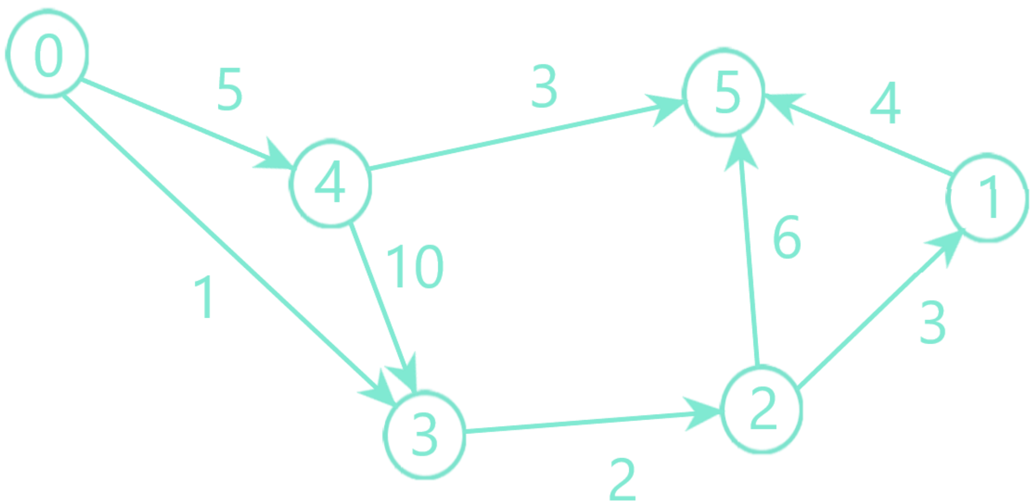
The implementation of backwards_dijkstra(graph, start, end) can be found in Walks.py.

The output of the function is composed of two dictionaries: successors(every node is mapped to it's successor in the minimum cost walk) and distances(every node is mapped to the minimum cost walk between start and itself). Then in the controller the path is computed using the successors dictionary and the distance using the distances dictionary.

For the following graph, the processed output is: 1← 2← 3←0 Length: 6, when start is 0 and end is 1.



And the unprocessed output is:

```
successors={
1: -1,
2: 1,
3: 2,
0: 3,
4: 3
}
```

```
Distances={
1: 0,
2: 3,
3: 5,
0: 6,
4: 15
}
```

Initial state:

Queue: (0, 1)

Successors: {1: -1}

Distances: {1: 0}

Step 1

Before changes

Queue after change: (0, 1)

Successors: {1: -1}

Distances: {1: 0}

After 2 is processed

Queue after change: (3, 2)

Successors: {1: -1, 2: 1}

Distances: {1: 0, 2: 3}

Step 2

Before changes

Queue after change: (3, 2)

Successors: {1: -1, 2: 1}

Distances: {1: 0, 2: 3}

After 3 is processed

Queue after change: (5, 3)

Successors: {1: -1, 2: 1, 3: 2}

Distances: {1: 0, 2: 3, 3: 5}

Step 3

Before changes

Queue after change: (5, 3)

Successors: {1: -1, 2: 1, 3: 2}

Distances: {1: 0, 2: 3, 3: 5}

After 0 is processed

Queue after change: (6, 0)

Successors: {1: -1, 2: 1, 3: 2, 0: 3}

Distances: {1: 0, 2: 3, 3: 5, 0: 6}

After 4 is processed

Queue after change: (6, 0) (15, 4)

Successors: {1: -1, 2: 1, 3: 2, 0: 3, 4: 3}

Distances: {1: 0, 2: 3, 3: 5, 0: 6, 4: 15}

Step 4

Before changes

Queue after change: (6, 0) (15, 4)

Successors: {1: -1, 2: 1, 3: 2, 0: 3, 4: 3}

Distances: {1: 0, 2: 3, 3: 5, 0: 6, 4: 15}

Now if start is in the keys of either dictionaries, then it means there is a walk between the two nodes and the path is reconstructed from the successors dictionary (implementation in controller.py, backwards_dijkstra).

Initial state:

Path: []

Distance: -1

Step 1

Current: 0

Step 2

Current: 3

Path: [0]

Step 3

Current: 2

Path: [3, 0]

Step 4

Current: 1

Path: [2, 3, 0]

Step 5

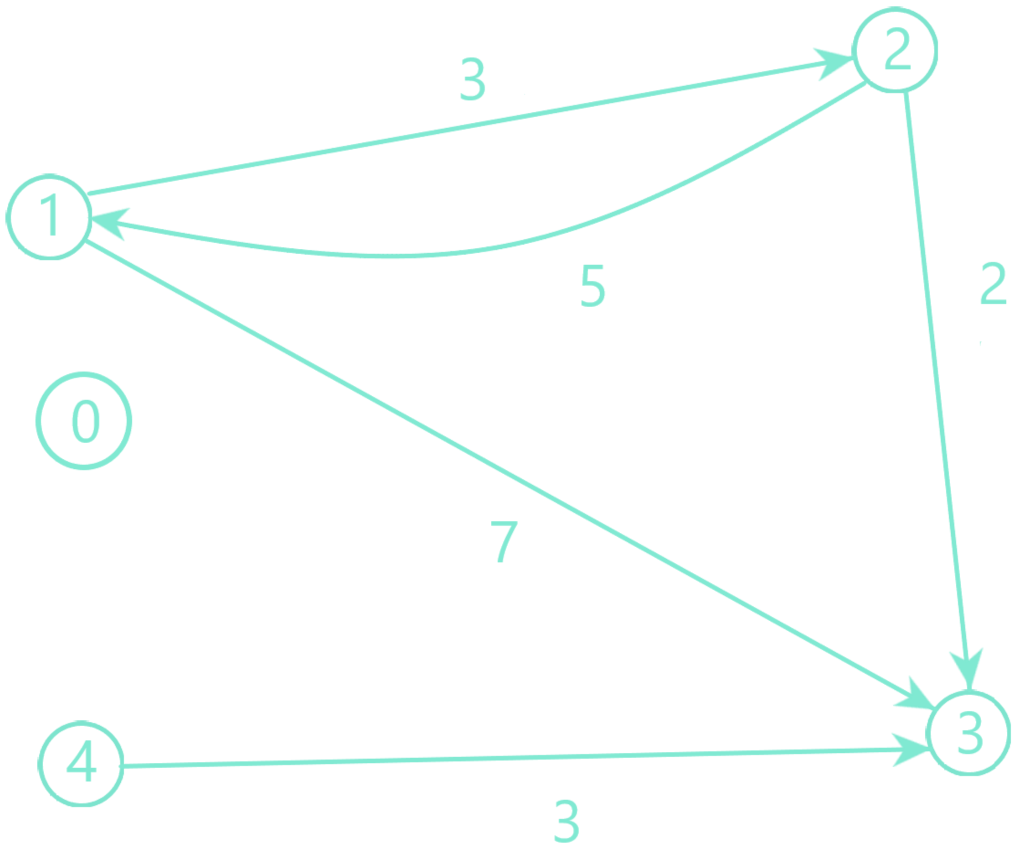
Current: -1

Path: [1, 2, 3, 0]

6

1← 2← 3← 0 Length: 6

Second example:



Start is 0 and end is 3

Initial state:

Queue: (0, 3)

Successors: {3: -1}

Distances: {3: 0}

Step 1

Before changes

Queue after change: (0, 3)

Successors: {3: -1}

Distances: {3: 0}

After 1 is processed

Queue after change: (7, 1)

Successors: {3: -1, 1: 3}

Distances: {3: 0, 1: 7}

After 2 is processed

Queue after change: (2, 2) (7, 1)

Successors: {3: -1, 1: 3, 2: 3}

Distances: {3: 0, 1: 7, 2: 2}

After 4 is processed

Queue after change: (2, 2) (3, 4) (7, 1)

Successors: {3: -1, 1: 3, 2: 3, 4: 3}

Distances: {3: 0, 1: 7, 2: 2, 4: 3}

Step 2

Before changes

Queue after change: (2, 2) (3, 4) (7, 1)

Successors: {3: -1, 1: 3, 2: 3, 4: 3}

Distances: {3: 0, 1: 7, 2: 2, 4: 3}

After 1 is processed

Queue after change: (3, 4) (5, 1) (7, 1)

Successors: {3: -1, 1: 2, 2: 3, 4: 3}

Distances: {3: 0, 1: 5, 2: 2, 4: 3}

Step 3

Before changes

Queue after change: (3, 4) (5, 1) (7, 1)

Successors: {3: -1, 1: 2, 2: 3, 4: 3}

Distances: {3: 0, 1: 5, 2: 2, 4: 3}

Step 4

Before changes

Queue after change: (5, 1) (7, 1)

Successors: {3: -1, 1: 2, 2: 3, 4: 3}

Distances: {3: 0, 1: 5, 2: 2, 4: 3}

Step 5

Before changes

Queue after change: (7, 1)

Successors: {3: -1, 1: 2, 2: 3, 4: 3}

Distances: {3: 0, 1: 5, 2: 2, 4: 3}

Initial state:

Path: []

Distance: -1

The start vertex is not accessible