Kristin Schaefer

HW6 : Graph Algorithms

Sources:

Problem 2
1. https://bradfieldcs.com/algos/graphs/dijkstras-algorithm/
Problem 3
2. https://stackoverflow.com/questions/13159337/why-doesnt-dijkstras-algorithm-work-for-negative-weight-edges

1. Graph Traversal

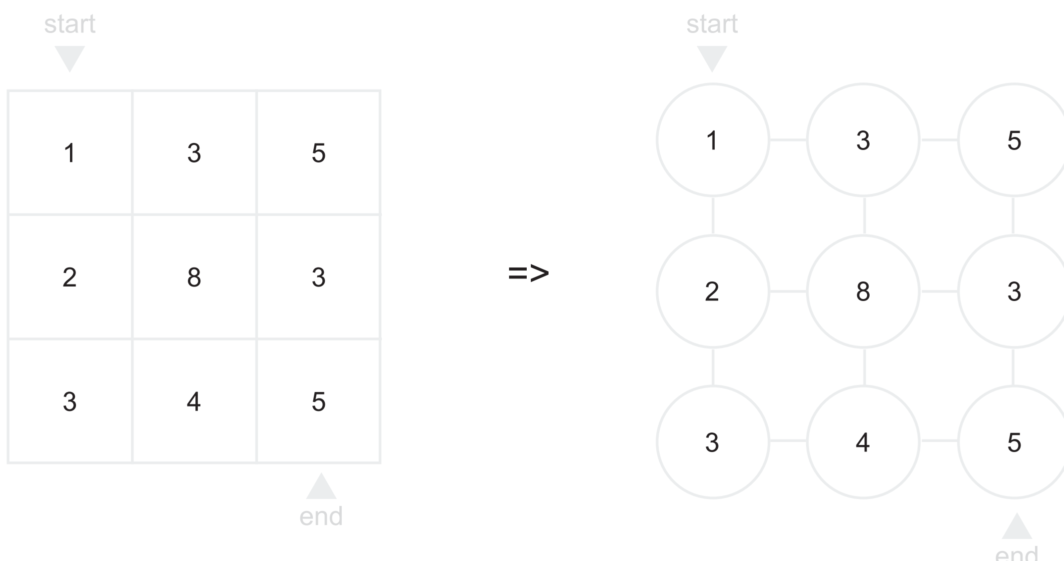
BFS: *A, B, D, C, F, G, E*

DFS: *A, B, F, C, D, E, G*

2. Apply BFS/DFS to solve a problem

a. description

Using the example provided in the assignment, we can translate the 2d puzzle[m][n] into an undirected graph:



Evaluating at the puzzle as an undirected graph, it looks a lot like the Dijkstra algorithm. The difference is that instead of looking at the cumulative weights of a path, we want to only consider the minimum effort of a path.

The relation of the Dijkstra of the algorithm is (where we are considering x->y) :
dist[y] = min{ dist[y] , weight[x,y]+dist[x] }

Our modified version to consider the minimum effort of the path:
effort[y] = min{ effort[y], effort[x] }

Since the *minEffort(puzzle)* algorithm is very similar to the Dijkstra algorithm, I adapted the *calculate_distances(graph, starting_vertex)* provided in the lecture. This algorithm uses a BFS approach, so a priority queue implemented as a minimum heap is used.

Since puzzle[][] is a 2d array, a nested for-loop is used to convert the puzzle to a graph. For each vertex in the graph, it's neighbours are computed and stored as cell indices with their corresponding efforts.

Kristin Schaefer

HW6 : Graph Algorithms

b. pseudocode

```
minEffort(puzzle[m][n]):

    # convert 2d array puzzle[m][n] into graph for calculate_min_effort()
    starting_vertex = “r” + str(0) + “-” + “c” + str(0)
    end_vertex = “r” + str(m-1) + “-” + “c” + str(n-1)
    puzzle_graph = {}

    for m in range(len(puzzle)):
        for n in range(len(puzzle[0])):
            current_cell = “r” + str(m) + “-” + “c” + str(n)
            puzzle_graph[current_cell] = {}
            # if not top row
            if m != 0:
                neighbor_cell = “r” + str(m-1) + “-” + “c” + str(n)
                puzzle_graph[current_cell][neighbor_cell] = abs(puzzle[m][n] - puzzle[m-1][n])
            # if not left column
            if n != 0:
                neighbor_cell = “r” + str(m) + “-” + “c” + str(n-1)
                puzzle_graph[current_cell][neighbor_cell] = abs(puzzle[m][n] - puzzle[m][n-1])
            # if not bottom row
            if m != (len(puzzle)-1):
                neighbor_cell = “r” + str(m+1) + “-” + “c” + str(n)
                puzzle_graph[current_cell][neighbor_cell] = abs(puzzle[m][n] - puzzle[m+1][n])
            # if not rightmost column
            if n != (len(puzzle[0])-1):
                neighbor_cell = “r” + str(m) + “-” + “c” + str(n+1)
                puzzle_graph[current_cell][neighbor_cell] = abs(puzzle[m][n] - puzzle[m][n+1])

    return calculate_min_effort(puzzle_graph, starting_vertex, end_vertex)
```

```
calculate_min_effort (graph{}, starting_vertex, end_vertex) :

    efforts = {vertex: float(‘infinity’) for vertex in graph}
    efforts[starting_vertex] = 0

    # set up priority queue with starting vertex
    pq = [(0, starting_vertex)]
    while len(pq) > 0:
        current_effort, current_vertex = pop (effort, vertex) from heap pq

        if current_effort > efforts[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            effort = weight

            if effort < efforts[neighbor]:
                efforts[neighbor] = effort
                push (effort, neighbor) onto heap pq

    return efforts[end_vertex]
```

Kristin Schaefer

HW6 : Graph Algorithms

c. time complexity

Let V equal the number of vertices in the puzzle[m][n] which is m*n
Let E equal the number of edges in the puzzle[m][n] which is 2*(m*n), since the puzzle is an undirected graph

For *calculate_min_effort()* the while loop is executed E times for each edge and the inner for loop will be executed V times for each vertex. Push and pop from pq take log V time complexity. So *calculate_min_effort()* will have O((V+E)logV) time complexity.

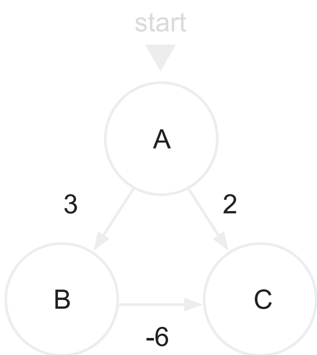
For *minEffort()*, we need to convert the 2d array puzzle[m][n] into a graph for *calculate_min_effort()*. This takes O(V+E), since for each vertex we need to calculate the weight of each neighbour and add it to the graph, which there are E in total.

Since O((V+E)logV) > O(V+E), the time complexity will be O((V+E)logV)

Kristin Schaefer

HW6 : Graph Algorithms

3. Analyze Dijkstra with Negative Edges



Vertex	Cost
A	0
B	∞
C	∞

Step 1.

We will first compare the length of the edges (A, B, 3) and (A, C, 2) and update our path length in the table.

Vertex	Path length
A	0
B	3
C	2

Step 2.

We have reached the end vertex C, with a cost of 2. However, if we had explored B, we would have found that the total path length would be 3 + -6 = -3. -3 < 2, so the solution provided with the Dijkstra algorithm is incorrect.

Vertex	Path length
A	0
B	-3
C	2

We have an incorrect solution because the algorithm only 'visits' a vertex once, and at a vertex we follow the relation:
(where we are considering x->y) : dist[y] = min{ dist[y] , weight[x,y]+dist[x] }
We assume that this relation provides us with an optimal solution for a vertex and then move on to the next vertex. However, the inductive step does not hold if there is a negative weight, because the Dijkstra algorithm is greedy by design. At each step we assume that by choosing a locally optimum value we will reach a globally optimum solution. If the next step holds a better locally optimum value than the previous step because there is a negative weight, then the problem does not have an optimal substructure, and thus won't provide a globally optimum solution.

Kristin Schaefer

HW6 : Graph Algorithms

Extra credit

BFS: *A, B, C, D, E, G, F, I, H, J*

DFS: *A, B, C, D, E, G, F, H, I, J*