Homework Assignment Week 4: Graph

Hyuntaek Oh

ohhyun@oregonstate.edu

CS 514_400 Algorithm

October 28, 2024

1. **Implement the BFS to solve a sliding tile puzzle called 8-puzzle.**

```python
def ShortestPath(goal_state, init_state):

    ans = []

    # Reshape goal_state into 2-D array (3x3)
    goal_matrix = [goal_state[i:i+3] for i in range(0, len(goal_state), 3)]

    # By using bfs, search the possible paths
    visited = bfs(goal_matrix)

    # Find the length of paths for each init_state
    for init_i in init_state:
        init_matrix = [init_i[i:i+3] for i in range(0, len(init_i), 3)]
        init_tuple = tuple(map(tuple, init_matrix))

        # Append the shortest path, otherwise -1
        ans.append(visited.get(init_tuple, -1))

    return ans
```

```python
def bfs(state):
    q = [(state, 0)]
    visited = {tuple(map(tuple, state)): 0}

    while q:
        curr_state, path_length = q.pop(0)

        # Try all possible paths
        for dir in direction:
            new_state = move_blank(curr_state, dir)
            if new_state is not None:
                new_state_tuple = tuple(map(tuple, new_state))
                if new_state_tuple not in visited:
                    visited[new_state_tuple] = path_length + 1
                    q.append((new_state, path_length + 1))

    return visited
```

Fig 1. ShortestPath and bfs functions

```python
def move_blank(state, dir):
    i, j = find_blank_position(state)
    di, dj = direction[dir]
    ni, nj = i + di, j + dj

    # Swap the values after moving if the move is within boundary
    if 0 <= ni < 3 and 0 <= nj < 3:
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[ni][nj] = new_state[ni][nj], new_state[i][j]
        return new_state

    return None
```

```python
def find_blank_position(state):
    # Find the position of zero
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

    return None
```

```python
direction = {
    'up': (-1, 0),
    'down': (1, 0),
    'left': (0, -1),
    'right': (0, 1)
}
```

Fig 2. Move_blank, find_blank_position, and direction functions

2. **Suppose that the only negative edges are those that leave the starting node S. Does Dijkstra's algorithm find the shortest path from S to every other node in this case? Justify your answer carefully.**

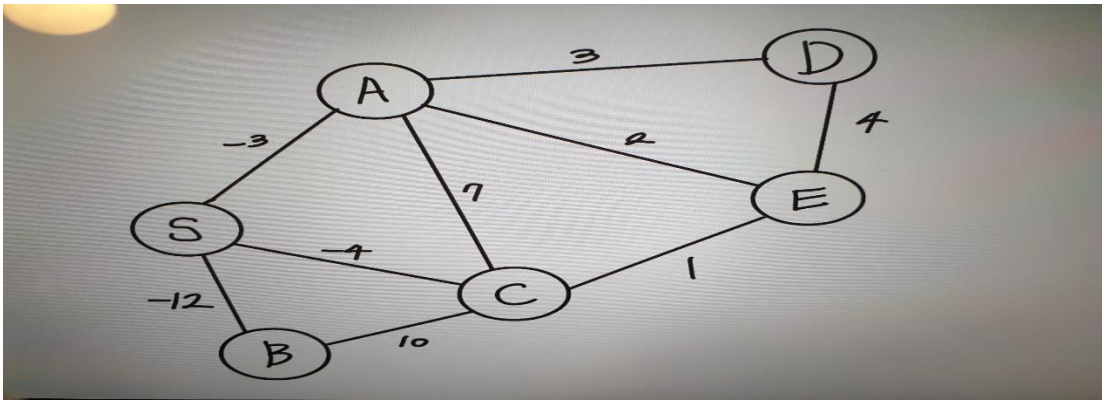

Fig 3. The graph with negative weight edges

Dijkstra's algorithm basically assumes that all edge weights are non-negative. If node S has negative weight edges, the algorithm's priority queue or greedy approach can fail to find possible shortest paths.

For example, as can be seen in Fig 3., every node near node S, which are A, B, C, is the shortest path from node S if the target node is one of the adjacent nodes. On the other hand, if we use negative values of weight, there can be an infinite negative cycle during the searching the possible paths. Let's assume the travel from node S to node D. When leaving from node S, Dijkstra's algorithm tries to minimize the weight on edges, choosing a value of '-12', which is a route of node B. Then, the algorithm chooses the next path containing a minimum value, which is 10. In the figure, from the node C, the next minimum value is connected to node S, which is a value of -4, and thus the process to travel from node S to node D is failed since every next selection is being a negative infinite cycle ($S \rightarrow B \rightarrow C \rightarrow S \rightarrow B \rightarrow C \rightarrow S$ ....).

Thus, Dijkstra's algorithm cannot find the shortest path from node S to every other node in this case.

3. Give an algorithm that takes a directed graph as input and returns the length of the shortest cycle in the graph where n is the number of nodes.

A cycle is defined as a path in a given graph that starts and ends at the same vertex, meaning that there is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $v_1 = v_k$. The shortest cycle means the cycle with the minimum total edge length among all possible cycles in the graph.

According to the contents in Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, we can use the modified Floyd-Warshall algorithm to solve this problem. The algorithm is typically used to find the shortest paths between all pairs of nodes in $O(n^3)$ time. The pseudo code of Floyed-Warshall algorithm is below.

FLOYD-WARSHALL

$D = W$

for $k = 1$ to n

    for $i = 1$ to n

        for $j = 1$ to n

            $d_{ij} = \min\{d_{ij}, d_{ik} + d_{kj}\}$

return D

where D is vertices and W is the weights of edges. k, i, j are the nodes that can be possible pairs. The basic idea of the algorithm above is to find the shortest path in all pairs of nodes. Each path in the shortest path can combine with other shortest path to minimize the weight of the target path. For example, a graph is like:

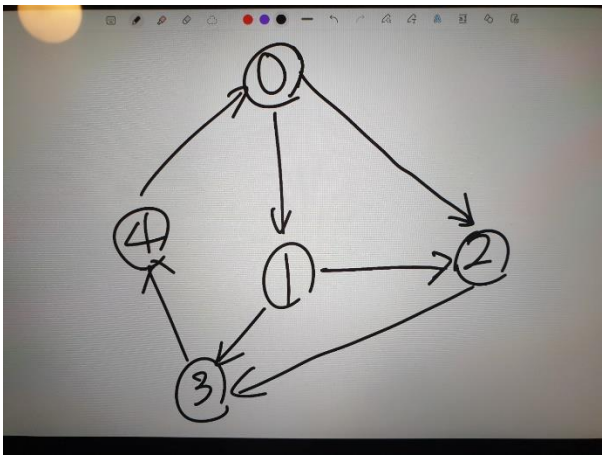

Fig 4. A graph containing cycles

If we are looking for a cycle of $node\ 1$, there is one of the shortest cycles ($1 \rightarrow 3 \rightarrow 4 \rightarrow 0 \rightarrow 1$). The length of the cycle can be calculated by the combination of the shortest paths such as $(1 \rightarrow 3)$, $(3 \rightarrow 4)$, $(4 \rightarrow 0)$, and $(0 \rightarrow 1)$. For this, the Floyd-Warshall algorithm calculates all the possible pairs and compare them with other possible path pairs to minimize the cost. In particular, it uses three nodes at once to figure out which path is the shortest one. The value of k is the intermediate node to traverse from $node\ i$ to $node\ j$.

The time complexity of the algorithm is $O(V^3)$ since three layers of nested loops required to examine all paths via each intermediate node for each pair of starting and ending nodes.

In the Initialization step, it takes $O(V^2)$ time, as each of the $V \times V$ entries in the matrix needs to be initialized based on the input graph.

In the main iterative process, the three nested loops effectively go through every combination of nodes i, j, and k, making this part of the algorithm run in $O(V^3)$ time.

Lastly, we need to loop for finding the shortest cycle since there is no specific articulation about finding the shortest cycle in the Floyd-warshall algorithm (That's the reason we use modified one). The index i, i pair can be used to achieve the goal since the same index means a specific cycle that starts and ends at the same node, so the algorithm has a loop for it, which takes $O(V)$.

So, the total time complexity of the algorithm is:

$$O(V^2) + O(V^3) + O(V) = O(V^3)$$

V is equal to the number of nodes, n.

In conclusion, we can say that the time complexity of the algorithm is $O(n^3)$

4. Give an efficient algorithm for finding the shortest paths between all pairs of nodes with the restriction that they all must pass through node A given a strongly connected directed graph $G = (V, E)$ with positive edge weights.

A strongly connected directed graph is a type of directed graph in which there is a path from any node to every other node. This means that for any two vertices u and v in the graph, there exists a directed path from u to v and another directed path from v to u.

An efficient algorithm for finding the shortest paths that must be passed through node A can be implemented by using Dijkstra's algorithm with min-heap priority queue. The basic idea of solution to this problem is that the shortest path from starting node to target node passing through node A can be found if we know the shortest path to start and end at node A, meaning that

$$(i \to A) + (A \to j) = (i \to A \to j)$$

The main function is below.

```
def all_pairs_shortest_path_through_A(graph, n, A, i, j):
    distance_from_A = Dijkstra(graph, A, n)
    reversed_graph = reverse_graph(graph, n)
    distance_to_A = Dijkstra(reversed_graph, n)

    return distance_from_A[j] + distance_to_A[i]
```

We utilize Dijkstra algorithm to find the shortest path reaching target node j, $(A \to j)$. The algorithm is below.

Dijkstra(graph, start, n):

Initialize all elements in distance list to infinite

Initialize starting node to zero

A set for visited node

Priority queue (pq) contains starting node and zero distance

While pq is not empty:

Pop (current_distance, current_node) from pq

If current_node is visited, continue

Else update visited set to True

For every next adjacent node and distance of current node:

If the sum of the distance of current node and next weight is smaller than the distance of next adjacent node,

Update the node to smaller value and push this node and distance to pq.

Return distance list

The main point of Dijkstra algorithm is that it chooses minimum weights by updating the minimum cost of every traversal, yielding the shortest path from starting node to target node.

The time complexity of Dijkstra algorithm is $O((V+E)logV)$ since maintaining min-heap priority queue and extraction from the heap take $O(VlogV)$. For each of the E edges, the heap updates the minimum distance to neighboring nodes, taking $O(ElogV)$. So, the total time complexity of Dijkstra algorithm is $O((V+E)logV)$.

In addition, we use reverse graph to apply Dijkstra algorithm searching all paths when node A is at the end, $(i \rightarrow A)$ since the current Dijkstra algorithm uses starting node, not ending node. The reverse graph function is below.

Reverse_graph(graph, n):

Initialize empty list for reversed graph

For each node u:

For each neighboring node, weight of node u:

Add nodes and weights inverse order to list

Return reversed graph list

The Reverse_graph function adds reverse relationship between node u and v including weights into the reverse graph list. By using this, Dijkstra algorithm provides the shortest paths from node A to node j. Since initializing empty list takes $O(V)$ time and adding edges takes $O(E)$ time. Thus, the time complexity of Reverse_graph function is $O(V+E)$.

In the last step, we can get the sum of the distance from node i to j through node A. Thus, the total time complexity is:

$$O\big((V+E)logV\big) + O(V+E) + O\big((V+E)logV\big) = O\big((V+E)logV\big)$$