The 2nd-shortest Path

Date: 2023-12-20

Declaration

I hereby declare that all the work done in this project titled "***The 2nd-shortest Path***" is of my independent effort.

Chapter 1: Introduction

Problem description

The Shortest Path Problem is a classic problem in graph theory, aiming to find the shortest path between two vertices in a weighted graph. The Secondary Shortest Path Problem is an extension of this problem, typically referring to finding the second shortest path in a graph, excluding the shortest path. This problem is more complex compared to the single-source shortest path problem because it requires identifying the secondary shortest path rather than just the shortest one.

Background of the algorithms

There are multiple algorithms to solve The Shortest Path Problem, and **Dijkstra's algorithm** is one of them. Dijkstra's algorithm employs a greedy strategy, starting from the source node and choosing the vertex with the current shortest distance at each step. It continually updates the shortest distances from the source to other vertices. In this project, I have enhanced Dijkstra's algorithm to solve the second shortest path problem.

Task

This project will address the specific 'second shortest path problem' outlined below using an enhanced version of the Dijkstra's algorithm:

Holiday is coming. Lisa wants to go home by train. But this time, Lisa does not want to go home directly -- she has decided to take the second-shortest rather than the shortest path. Your job is to help her find the path.

Chapter 2: Algorithm Specification

Here is my concrete algorithm to the problem:

1. Use Dijkstra's algorithm to find the shortest path.
2. Remove one path from the graph and find a new shortest path.
3. If the length of the new shortest path is greater than the original shortest path, record it in set S. If the length is equal, it indicates that there is a need to repeat visiting certain nodes. Therefore, find the minimum cycle containing the nodes of the new shortest path and add the cycle to the new shortest path.
4. Restore the deleted path in the original graph, then repeat steps 2 and 3 until all paths in the graph have been removed.

5. Find the shortest path in set S, which is the second shortest path.

Pseudocode for the second shortest path algorithm

```
def find_second_shortest_path(graph):
    # Step 1: Find the original shortest path
    original_shortest_path = dijkstra(graph)

    # Step 2: Iterate through each edge in the graph
    for edge in graph.edges:
        # Step 2.1: Remove the current edge from the graph
        graph.remove_edge(edge)

        # Step 2.2: Find the new shortest path after removing the edge
        new_shortest_path = dijkstra(graph)

        # Step 3: Compare the lengths of the new and original shortest paths
        if new_shortest_path.length > original_shortest_path.length:
            S.add(new_shortest_path)
        elif new_shortest_path.length == original_shortest_path.length:
            # Step 3.1: Find the minimum cycle containing the new shortest path nodes
            cycle = find_minimum_cycle(graph, new_shortest_path.nodes)
            # Step 3.2: Add the cycle to the new shortest path
            new_shortest_path.add_cycle(cycle)

        # Step 4: Restore the removed edge in the graph
        graph.restore_edge(edge)

    # Step 5: Find the shortest path in set S
    second_shortest_path = find_shortest_path(S)

    # Return the second shortest path
    return second_shortest_path
```

Therefore, solving this problem involves two key points: first, the specific implementation of Dijkstra's algorithm, and second, how to find the shortest cycle on the shortest path. I will now analyze them one by one.

the specific implementation of Dijkstra's algorithm

The algorithm maintains two sets: one for the vertices with determined shortest paths and the other for those with undetermined paths. At each step, it selects the vertex with the shortest distance from the set of undetermined paths and updates the shortest distances of its neighboring vertices. Here is pseudocode for the algorithm(The actual code is familiar to everyone, so I won't include it here).

```
void Dijkstra( Table T )
{
    Vertex  V, W;
    for ( ; ; ) {
        V = smallest unknown distance vertex;
        if ( V == NotAVertex )
    break;
        T[ V ].Known = true;
        for ( each W adjacent to V )
    if ( !T[ W ].Known )
```

```
        if ( T[ V ].Dist + Cvw < T[ W ].Dist ) {
            Decrease( T[ W ].Dist  to
             T[ V ].Dist + Cvw );
        T[ W ].Path = V;
        }
    }
}
```

how to find the shortest cycle

Here is the the algorithm for finding the shortest cycle in a path:

1. Find two distinct points in the path. Take the later point in the path (u) as the starting point and the other point (w) as the destination. Use Dijkstra's algorithm to find the minimum value, denoted as s1, between these two points.
2. Add s1 to the length of the path from w to u, denoted as s2. The result, s3, represents the length of the cycle.
3. Traverse all pairs of points in the path, find the shortest cycle length, which is the desired result.

Pseudocode

```
function find_shortest_cycle_length(path):
    shortest_cycle_length = INFINITY

    for each pair of distinct points u, w in path:
        s1 = dijkstra(u, w)  # Step 1
        s2 = distance_between(u, w)  # Distance from w to u in the original path
        s3 = s1 + s2  # Step 2

        if s3 < shortest_cycle_length:
            shortest_cycle_length = s3  # Update the shortest cycle length

    return shortest_cycle_length
```

Exmaple

identify the shortest path: 1->2->4

sequentially remove each edge to obtain a new shortest path:

delete 1->2 : 1->3->4 delete 1->3 : 1->2->4 delete 2->4 : 1->3->4

delete 3->4 : 1->2->4 delete 4->1 : 1->2->4

Identify paths with a length equal to the original shortest path, and add the shortest cycle.

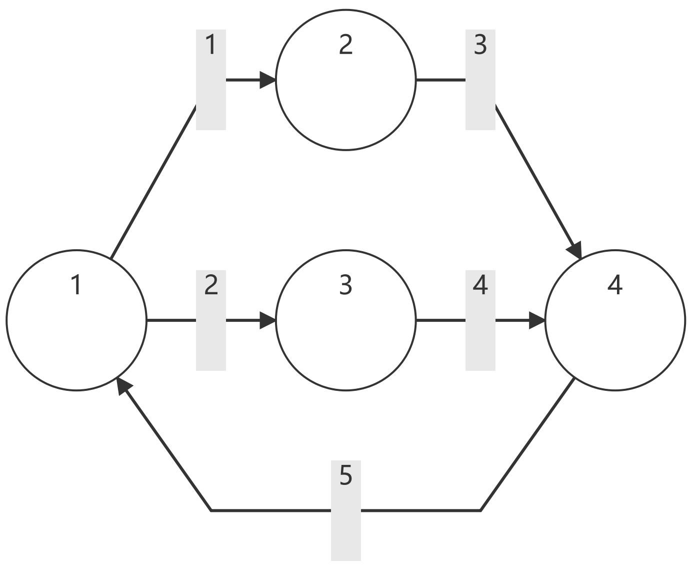
1->2->4 : cycle of (4,2) : lenth(4->1->2) + lenth(2->4) = 9

cycle of (4,1) : lenth(4->1) + lenth(1->2->4) = 9

cycle of (2,1) : lenth(2->4->1) + lenth(1->2) = 9

Compare all shortest path lengths to obtain the second shortest path.

length(1->2->4) = 9 < length(1->3->4), so the second shortest path is (1->3->4).



how to store the data

Structure to represent a graph

```
typedef struct {
    int Nv;                         // Number of vertices
    int Ne;                         // Number of edges
    int Edge[MaxSNum + 1][MaxSNum + 1];  array to store edge weights
} StationGraph;
```

Structure to store information for Dijkstra's algorithm

```
typedef struct record {
    int Dist[MaxSNum + 1];  // Distance array
    int Path[MaxSNum + 1];  // Path array
    int state[MaxSNum + 1]; // State array (visited or not)
} Record;
```

Structure to represent a circle (used for backtracking)

```
typedef struct circle {
    int Path[MaxSNum + 1];  // Stores the nodes of the cycle
} Circle;

Circle circles[MaxSNum + 1];

// Global variables to store the min-circle information
int path[MaxSNum + 1];  // Stores the information about the shortest cycle corresponding
to a particular path
int a;
int aa[MaxSNum + 1];  // Stores the endpoint of the shortest cycle corresponding to a
particular path
int b;
int bb[MaxSNum + 1];  // Stores the starting point of the shortest cycle corresponding
to a particular path
```

Chapter 3: Testing Results

I ensure the authenticity of all test cases.

1. Sample Input:

```
5 6
1 2 50
2 3 100
2 4 150
3 4 130
3 5 70
4 5 40
```

Sample output:

```
240 1 2 4 5
```

2. Sample Input:

```
1 0
```

Sample output:

```
No second shortest road
```

3. Sample Input:

```
6 7
1 2 1
2 3 2
3 6 2
3 5 5
5 2 1
3 4 3
4 2 4
```

Sample output:

```
13 1 2 3 5 2 3 6
```

4. Sample Input:

```
3 2
1 2 3
2 4 2
```

Sample output:

```
No second shortest road
```

5. Sample Input:

```
6 9
1 2 10
1 3 6
1 5 1
5 3 1
3 4 1
4 2 1
4 6 2
3 2 5
2 6 3
```

Sample output:

```
7 1 5 3 4 2 6
```

6. Sample Input:

```
5 5
1 2 1
2 3 2
3 4 1
4 2 3
3 5 1
```

Sample output:

```
10 1 2 3 4 2 3 5
```

7. Sample Input:

```
3 6
1 3 4
1 2 1
1 2 1
1 2 2
2 3 1
2 3 1
```

Sample output:

```
4 1 3
```

Chapter 4: Analysis and Comments

Analysis of the time complexities

1. **Dijkstra's Algorithm in `Dijkstra` Function:**
   - Time Complexity: O(V^2), where V is the number of vertices. This is because the code uses an adjacency matrix to represent the graph, and each iteration of the main loop involves scanning through all vertices.

2. **Edge Deletion Loop in `DeleteEdge` Function:**
   - Time Complexity: O(V^2 * E), where E is the number of edges. This is because, in the worst case, for each edge, the Dijkstra's algorithm is run, resulting in a total of V^2 * E operations.
3. **Backtracking Loop in `MinDistance` Function:**
   - Time Complexity: O(V^3), where V is the number of vertices. This is because, in the worst case, the backtracking loop runs V^2 times, and in each iteration, Dijkstra's algorithm is run, resulting in a total of V^3 operations.
4. **Printing the Second Shortest Path in `Second_shortest` Function:**
   - Time Complexity: O(V), where V is the number of vertices. This is because, in the worst case, the loop runs through all vertices to print the path.

Analysis of the space complexities

The space complexity is primarily influenced by the record array the backtrack array. The record array requires storage for distance, path, and state arrays, occupying O(V*E) space. The backtrack array occupies O(V^2) space. The overall space complexity is O(V^E).

Areas for Improvement:

1. **Time Complexity:**
   - The use of Dijkstra's algorithm for edge deletion and backtracking results in high time complexity, especially in scenarios with large graphs. Consider optimizing these sections for better efficiency.
2. **Space Complexity:**
   - The space complexity, particularly the use of an adjacency matrix, may lead to inefficiencies for sparse graphs. Consider using an adjacency list for better space utilization.

Appendix: Source Code (in C)

Refer to the code file.