Project 2: Graph Algorithms and Related Data Structures

ITCS-6114, Summer 2024
July 29, 2024

Introduction

This report covers the implementation of two graph algorithms: the Single-source Shortest Path Algorithm (Dijkstra's) and the Minimum Spanning Tree Algorithm (Kruskal's). The project involves running these algorithms on directed and undirected graphs, analyzing their runtime, and writing a report on implementation and usage

Single-source Shortest Path Algorithm

Pseudocode for Dijkstra's Algorithm:

```
Algorithm Dijkstra(graph, source):
    Input: graph, a weighted graph represented as an adjacency list
           source, the starting vertex
    Output: distances, a dictionary of the shortest distances from the source to each vertex
            paths, a dictionary of the shortest paths from the source to each vertex

    Initialize:
        distances <- a dictionary with all vertices set to infinity
        distances[source] <- 0
        priority_queue <- a min-heap initialized with (0, source)
        paths <- a dictionary with all vertices set to an empty list
        paths[source] <- [source]

    While priority_queue is not empty:
        current_distance, current_vertex <- extract the vertex with the smallest distance fr

        If current_distance > distances[current_vertex]:
            continue

        For each neighbor, weight in graph[current_vertex]:
```

```
            distance <- current_distance + weight

            If distance < distances[neighbor]:
                distances[neighbor] <- distance
                Insert (distance, neighbor) into the priority_queue
                paths[neighbor] <- paths[current_vertex] + [neighbor]

    Return distances, paths

End Algorithm
```

Detailed Analysis:

Dijkstra's algorithm finds the shortest path from one vertex to all other vertices in a weighted graph by selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighboring vertices. Dijkstra's algorithm does not work for graphs with negative edges. Although not covered in this project If the graph has negative edges then the Bellman-Ford algorithm is the more efficient algorithm to use.

Runtime Analysis:

- **Initialization:** $O(V)$, where V is the number of vertices.
- **Priority Queue Operations:**
  - Using a simple array or linked list: $O(V^2)$
  - Using a binary heap: $O((V + E) \log V)$
  - Using a Fibonacci heap: $O(V \log V + E)$
- **Total Time Complexity:**
  - Using a simple array or linked list: $O(V^2)$
  - Using a binary heap: $O((V + E) \log V)$
  - Using a Fibonacci heap: $O(V \log V + E)$

In this project, the Dijkstra's algorithm implementation uses a binary heap for the priority queue operations. Therefore, the specific runtime analysis is:

- **Initialization:** $O(V)$
- **Priority Queue Operations:** $O((V + E) \log V)$
- **Total Time Complexity:** $O((V + E) \log V)$

Data Structures Used

- **Graph Representation:** The graph is represented using a `defaultdict` of dictionaries. This allows efficient storage and retrieval of edge weights between vertices.

  ```
  graph = defaultdict(dict)
  ```

- **Priority Queue:** Min-heap implemented using Python's `heapq` module. This allows the algorithm to always extract the minimum element (vertex with the smallest distance).

  ```
  priority_queue = [(0, source)]
  ```

- **Distance Dictionary:** Keeps track of the shortest known distance to each vertex. Initially, all distances are set to infinity, except the source which is set to 0.

  ```
  distances = {vertex: float('infinity') for vertex in graph}
  distances[source] = 0
  ```

- **Path Dictionary:** Keeps track of the shortest paths to each vertex for reconstruction.

  ```
  paths = {vertex: [] for vertex in graph}
  paths[source] = [source]
  ```

Effect on Runtime

Graph Representation: O(V + E) space, O(1) access time.

Priority Queue: O((V + E) log V) time for Dijkstra's algorithm.

Distance Dictionary: O(V) space.

Implementation

The implementation of Dijkstra's algorithm is provided in `problem1.py`. The graph is read from an input file, and the algorithm processes the graph to find the shortest paths.

Code Snippet:

```
import heapq
from collections import defaultdict

def dijkstra(graph, source):
    distances = {vertex: float('infinity') for vertex in graph}
```

```
    distances[source] = 0
    priority_queue = [(0, source)]
    paths = {vertex: [] for vertex in graph}
    paths[source] = [source]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
                paths[neighbor] = paths[current_vertex] + [neighbor]

    return distances, paths
```

Example Input:

```
6 10 U
A B 1
A C 2
B C 1
B D 3
B E 2
C D 1
C E 2
D E 4
D F 3
E F 3
```

Example Output:

```
Enter the source node: A
Single-source Shortest Paths from a
Path to a: a, Cost: 0
Path to b: a -> b, Cost: 1
Path to c: a -> c, Cost: 2
Path to d: a -> c -> d, Cost: 3
Path to e: a -> b -> e, Cost: 3
Path to f: a -> c -> d -> f, Cost: 6
Runtime: 0.0000524000 seconds
```

Minimum Spanning Tree Algorithm

Pseudocode for Kruskal's Algorithm:

```
Algorithm Kruskal(graph):
Input: graph, a weighted graph represented as an adjacency list
Output: mst, a list of edges in the Minimum Spanning Tree
        total_cost, the total weight of the Minimum Spanning Tree

Initialize:
    edges <- an empty list
    For each vertex u in graph:
        For each vertex v in graph[u]:
            If (u, v, weight) is not in edges and (v, u, weight) is not in edges:
                Add (weight, u, v) to edges
    Sort edges by weight in non-decreasing order

    vertices <- list of all vertices in the graph
    ds <- DisjointSet(vertices)

    mst <- an empty list
    total_cost <- 0

For each edge in edges:
    weight, u, v <- edge
    If ds.find(u) != ds.find(v):
        ds.union(u, v)
        Add (u, v, weight) to mst
        total_cost <- total_cost + weight

Return mst, total_cost
```

Detailed Analysis:

Kruskal's algorithm finds the Minimum Spanning Tree (MST) of a weighted, undirected graph. The algorithm works by repeatedly selecting the smallest edge that doesnt form a cycle with the already selected edges and adding it to the MST. This process continues until the MST spans all vertices of the graph.

Runtime Analysis:

Data Structures Used

- **Graph Representation:** Edge list.
- **Union-Find Structure:** Supports union and find operations.

Table 1: Kruskal's Algorithm Runtime Analysis

Component	**Time Complexity**
Initialization	$O(V)$
Disjoint Set Operations: Find	$O(\alpha(V))$
Disjoint Set Operations: Union	$O(\alpha(V))$
Sorting Edges	$O(E \log E)$
Processing Edges	$O(E\alpha(V))$
Total Time Complexity	$O(E \log E)$

Table 2: Specific Runtimes for Provided Inputs

Input File	**Edges in MST**	**Total Cost of MST**	**Runtime (seconds)**
input1.txt	5	8	0.0000642000
input2.txt	9	100	0.0000747000
input3.txt	8	68	0.0000760000
input4.txt	10	87	0.0000784000

Implementation

The implementation of Kruskal's algorithm is provided in `problem2.py`.

Code Snippet:

```
class UnionFind:
    def __init__(self, vertices):
        self.parent = {vertex: vertex for vertex in vertices}
        self.rank = {vertex: 0 for vertex in vertices}

    def find(self, vertex):
        if self.parent[vertex] != vertex:
            self.parent[vertex] = self.find(self.parent[vertex])
        return self.parent[vertex]

    def union(self, vertex1, vertex2):
        root1 = self.find(vertex1)
        root2 = self.find(vertex2)

        if root1 != root2:
            if self.rank[root1] > self.rank[root2]:
                self.parent[root2] = root1
            elif self.rank[root1] < self.rank[root2]:
                self.parent[root1] = root2
            else:
                self.parent[root2] = root1
                self.rank[root1] += 1
```

```
def kruskal(graph):
    edges = sorted(graph['edges'], key=lambda edge: edge[2])
    union_find = UnionFind(graph['vertices'])
    mst = []

    for edge in edges:
        u, v, weight = edge
        if union_find.find(u) != union_find.find(v):
            union_find.union(u, v)
            mst.append(edge)

    return mst
```

Testing

Example Input:

```
9 12 U
A B 23
A C 8
B C 11
B D 4
B E 6
C D 18
C E 11
D E 6
D F 9
E F 8
F G 5
G H 17
H I 9
```

Example Output:

```
Edges in the Minimum Spanning Tree:
B -- D == 4
F -- G == 5
B -- E == 6
A -- C == 8
E -- F == 8
H -- I == 9
B -- C == 11
G -- H == 17
Total cost of MST: 68
Runtime algorithm: 0.0000704000 seconds
```

Conclusion for Kruskal

When it comes to Kruskal's algorithm the number of edges directly impacts the algorithm's efficiency during the sorting phase. The algorithm is better with sparse graphs because of the fewer edges to process and sort and it has a time complexity of O(ElogE) or O(ElogV)

Conclusion

This project is the implementation of Dijkstra's algorithm for finding the shortest paths in a graph and Kruskal's algorithm for finding the minimum spanning tree.

The `README` file includes instructions for setting up and running the project.

References

References

[1] GeeksforGeeks, *Kruskal's Minimum Spanning Tree Algorithm - Greedy Algo-2*, Available: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/.

[2] Wikipedia, *Kruskal's Algorithm*, Available: https://en.wikipedia.org/wiki/Kruskal%27s_algorithm.

[3] GeeksforGeeks, *Difference Between Prim's and Kruskal's Algorithm for MST*, Available: https://www.geeksforgeeks.org/difference-between-prims-and-kruskals-algorithm-for-mst/?ref=lbp.

[4] GeeksforGeeks, *Dijkstra's Shortest Path Algorithm - Greedy Algo-7*, Available: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/.

[5] W3Schools, *Dijkstra's Algorithm*, Available: https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php.

[6] Wikipedia, *Dijkstra's Algorithm*, Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[7] CS Academy, *Graph Editor*, Available: https://csacademy.com/app/graph_editor/.