GraphsShortestPaths

August 7, 2020

1 Shortest Paths in Graphs

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GraphShortest.html

1.0.1 Table of Contents

- **Section ??**
- **Section ??**
- **Section ??**

1.1 Shortest-Path Problems

- modeling road networks to find shortest path from point A to point B
- road networks can be modeled as a directed graph whose edges are labeled with real numbers
    - labels may be called weights, costs, or distances
- a typical problem is to find the total length of the shortest path between two specified vertices
- see figure below, e.g.:
    - $w = $ weight
    - $d = $ shortest path
    - $w(A, D) = 20$
    - $d(A, D) = 10$
    - $w(E, B) = \infty$
- assume that all weights are positive

1.2 Single-Source Shortest Paths (SSSP)

- given vertex S in Graph G, find the shortest paths from S to every other vertex in G
- finding the shortest path from S to T requires us to find the shortest paths from S to every other vertex as well (in the worst case)
- algorithm presented here computes only the distance to every vertex rather than recording the actual path
- path can be recorded and printed by remembering parent vertex for each vertex using a vector (left as an exercise)

1.2.1 Applications

- find the cheapest way for one computer to broadcast a message to all other computers on a computer network
- find the fastest route from point A to point B

- find the cheapest flight from point A to point B

1.2.2 SSSP for Unweighted Graphs

- SSSP for unweighted graphs (or all edges with same cost) can be found using a simple breadth-first search

1.2.3 SSSP for Weighted Graphs

- use Dijkstra's algorithm
  - assumes weights are positive values

1.3 Dijkstra's SSSP Algorithm

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm - given a graph $G = (V, E)$: - shortest path from A to B: $d(A, B) = min(d(A, U)) + w(U, B)$ - $d(A, B)$ is the minimum over all paths that go from A to U, then have an edge from U to B, where U is some vertex in V. - Diajkstra's algorithm will assign some initial distance values and will try to improve them step by step - the technique commonly called "greedy"

- algorithms steps:
  1. create a set of all the **unvisited nodes**
  2. assign every node a tentative distance value using array: 0 for start vertex, ∞ for all other nodes
  3. for each node, consider all of its unvisited neighbors and calculate their tentative distances through the current node, update the distance with the smaller value e.g., if the current node u is marked with a distance of 6 and the edge connecting it with a neighbor v has length 2, then the distance to v through u is $6 + 2 = 8$. If v's current distance is greater than 8, then update it to 8
  4. when done considering all the unvisited neighbors of the current node, mark the current node as visited and remove it from the **unvisited set**
  5. select the next unvisited node that has the smallest tentative distance, and repeat from step 3
  - at the end, array created in step 2 will contain the shortest distance values

1.3.1 visualize Dijkstra's SSSP algorithm here: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GraphShortest.html

[ ]:
```cpp
#include <iostream>
#include <vector>
#include <queue> // priority_queue
#include <climits> // sizes of integral types
#include <utility> // make_pair
#include <list>
#include <algorithm>

using namespace std;
using iPair = pair<int, int>;
```

[ ]:
```
// Given a graph G, Dijkstra() finds SSSP to all the nodes from given source
// dist vector will have the shortest distances from the given source
// when the function terminates
// function can be modified to find shortest path to a single destination
// see Single Destination comment below
// function can also be modified to trace the shortest path using parent vector
template<class T>
void Dijkstra(T & G, int source, vector<int>& dist) {
    // min priority_queue of vertices that need to be processed
    // stores pair of <weight, vertex>
    priority_queue<iPair, vector<iPair>, greater<iPair> > pq;
    dist.resize(G.nodeCount());
    fill(dist.begin(), dist.end(), INT_MAX); // initialize distance vecrot to␣
 ↪some large int
    vector<bool> visited(G.nodeCount(), false);
    dist[source] = 0; // distance of source from source is 0
    pq.push({0, source}); // source node's {weight, vertex}
    while (! pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        // Single Destination:
        // if interested to find the path to one destination
        // check here if u == dest node; break if so
        if (visited[u]) continue; // if u already visited get next smaller␣
 ↪weight u
        visited[u] = true; // mark u as visited
        for(auto p: G.neighbors(u)) { // explore all the neighbors of u
            int v = p.first; // let's check a neighbor v of u
            if (visited[v]) continue; // if v is already visited; move to next␣
 ↪neighbor
            int w = p.second; //otherwise: w = w(u, v)
            // is this the shorter path to v via u?
            int d = dist[u] + w; // newd = dist(source, u) + w(u, v)
            if (d < dist[v]) { // newd < dist(source, v)
                dist[v] = d; // update the dist(source, v)
                pq.push({d, v}); // add {d, v} pair to the priority queue
                // update parent vector if path needs to be recorded
                //parent[v] = u; // use this edge: u->v
            }
        }
    }
}
```

1.4 apply and test Dijkstra's SSSP

- let's create Graph ADT using adjacency list
- use it to represent some graph

- test Diajkstra's SSSP algorithm using the graph

[ ]:
```
// Directed Graph using Adjacency List
// updae addEdge() for Undirected Graph
class Graph {
    private:
        vector<list<iPair> > graph; // list stores pair of neighbor id and␣
 ↪weight

    public:
        Graph(size_t n) {
            for (int i=0; i<n; i++) {
                list<iPair> v; // create an empty list of int, int pair type
                graph.push_back(v);
            }
        }

        // return the number of vertices/nodes
        size_t nodeCount() {
            return graph.size();
        }

        // add a new edge from node u to node v, with weight w
        // assumes nodes are numbered from 0 to n-1
        void addEdge(int u, int v, int w) {
            graph[u].push_back({v, w});
            // if undirected graph must add edge from v to u
            // graph[v].push_bck({u, w});
        }

        // returns list of pairs containing neighbors of u, and weight
        list<iPair> neighbors(int u) {
            return graph[u];
        }
};
```

[ ]:
```
// let's represent above directed graph
// A->0, B->1, C->2, D->3, E->4
Graph graph(5);
vector<int> dist;
```

[ ]:
```
graph.addEdge(0, 1, 10);
graph.addEdge(0, 3, 20);
graph.addEdge(0, 2, 3);
graph.addEdge(1, 3, 5);
graph.addEdge(2, 1, 2);
graph.addEdge(2, 4, 15);
```

```
graph.addEdge(3, 4, 11);
```

[ ]:
```
int source = 0;
```

[ ]:
```
Dijkstra<Graph>(graph, source, dist);
```

[8]:
```
cout << "shortest distances from source " << char(source+65) << " to all the␣
↪nodes are:\n";
for (int i=0; i< dist.size(); i++)
    cout << char(source+65) << " ~~> " << char(i+65) << " = " << dist[i] <<␣
↪"\n";
```

```
shortest distances from source A to all the nodes are:
A ~~> A = 0
A ~~> B = 5
A ~~> C = 3
A ~~> D = 10
A ~~> E = 18
```

[9]:
```
dist.clear();
source = 2; // C
```

[9]: 2

[10]:
```
Dijkstra<Graph>(graph, source, dist);
```

[11]:
```
cout << "shortest distances from source " << char(source+65) << " to all the␣
↪nodes are:\n";
for (int i=0; i< dist.size(); i++)
    if (dist[i] == INT_MAX)
        cout << char(source+65) << " ~~> " << char(i+65) << " = " <<␣
↪"Impossible" << "\n";
    else
        cout << char(source+65) << " ~~> " << char(i+65) << " = " << dist[i] <<␣
↪"\n";
```

```
shortest distances from source C to all the nodes are:
C ~~> A = Impossible
C ~~> B = 2
C ~~> C = 0
C ~~> D = 7
C ~~> E = 15
```

1.4.1 How can you tell if there's a path from source to a destination?

- if the distance(source, destination) is NOT the max sentinel value after running Dijkstra's SSSP

1.5 Time Complexity of Dijkstra's algorithm

- bulk of the cost comes from the loop which depends on the running time of priority queue
- because nodes are added into the priority queue repeatedly with different weight while exploring |E| edges, it'll raise the number of elements in the min-heap from $O(|V|)$ to $O(|E|)$
- when the graph is sparse, its cost is $O(|V| + |E|)log(|E|)$ in the worst case
- when the graph is dense, $|E|$ approaces $|V|^2$, so the cost can be as much as $O(|V|^2 log|E|)$ in the worst case

1.6 Exercises

1. Practice how Dijkstra's algorithm works using simulation at the end of: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GraphShortest.html

- George - https://open.kattis.com/problems/george
  - SSSP with extra weight time for some edge
  - SSSP to a single destination
- Single source shortest path, non-negative weights: https://open.kattis.com/problems/shortestpath1
- Flowery Trail - https://open.kattis.com/problems/flowerytrail
  - multiple paths with same minimum weight
  - traceback paths
- Honey Heist - https://open.kattis.com/problems/honeyheist
- Geezer Scripts - https://open.kattis.com/problems/geezerscripts
  - user max priority queue on remaining health from 1 to N
  - in order to continue to fight to determine winnner, use division to detect winner
    - whoever lasts longer rounds will win; remember player attacks first!
- Horror List - https://open.kattis.com/problems/horror
  - use HI as dist; run SSSP from horror list
  - report the index of max HI/dist value after running SSSP from all the horror list
- Tweak Dijkstra's algorithm to record path so that you can print shortest path from source to all other nodes
- Apply Dijkstra's algorithm to adjacency matrix-based graph
- Tweak Dijkstra's algoirithm to find shortest path to a single destination and test it

[ ]:

```
```