SpanningTreesPrims

August 7, 2020

1 Minimal Cost Spanning Trees (MST)

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

1.0.1 Table of Contents

1.1 Spanning Tree

- spanning tree of a graph is a sub-graph that is a tree and connects all the vertices together
- a graph can have many different spanning trees

1.2 MST Problems

- given a connected, undirected weighted graph G, MST is the graph containing the vertices of G along with the subset of G's edges that:
  1. has minimum total cost measured by summing the values for all of the edges in the subset
  2. keeps all the vertices connected
- some properties of MST
  1. contains NO cycles
  2. free tree with $|V| - 1$ edges
  3. the required set of edges forms a tree, it spans the vertices (i.e., connects them together)
  4. has minimal cost (hence MST)
- red edges indicated the subset making up the MST in the following tree
- note that edge (C, F) could be replaced with edge (D, F) to form a different MST with equal cost

1.3 MST Applications

http://www.utdallas.edu/~besp/teaching/mst-applications.pdf - building a connected network (e.g., electrical grid, computer network, transportation networks, water supply networks) fully connected at the lowest cost - Artificial Intelligence (AI) application - clustering: grouping a bunch of points into k clusters - handwriting recognition - curvilinear feature extraction in computer vision - computer circuit design - traveling salesman problem (TSP) - given a list of cities and the

distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

1.4 Prim's Algorithm

- Prim's algorithms for MST is very similar to Dijkstra's SSSP algorithm
- algorithm steps:
  1. start with any vertex, N in the graph, MST is initially N
  2. pick the least-cost edge connected to N connecting to say M
  3. MST now has vertices N and M and edge (N, M)
  4. pick the least-cost edge coming from current nodes in MST to any other vertex
  5. continue step 4 until all the nodes are in the MST
- priority queue-based implementation is extremely similar to Dijkstra's algorithm except for updating the weight/distance of each vertex
- the primary difference with Dijkstra's algorithm is that Prim's seeks not the next closest vertex to the start vertex, but rather the next closest vertex to any vertex currently in the MST
- e.g. while exploring the unvisited neighbor node v of u:
  - **Dijkstra's**:
  - minimimizes the tentative distance vector for each node from the source `// update distance[v] if the distance[u] + weight of the edge between u->v is smaller than current distance[v] if (distance[v] > distance[u] + weightBetween(u, v)) distance[v] = distance[u] + weightBetween(u, v)`
  - **Prim's**:
  - minimizes the tentative weight vector for each node from the parent node `// update weight[v] if the weight of edge between u->v is smaller than current weight[v] if (weight[v] > weightBetween(u, v)) weight[v] = weightBetween(u, v)`

1.4.1 visualize Prim's algorithm: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/MCS

1.4.2 Implementation of Prim's algorithm using priority queue

[1]:
```
#include <iostream>
#include <vector>
#include <queue> // priority_queue
#include <utility> // make_pair
#include <list>
#include <climits> // sizes of integral types
#include <algorithm>

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

[2]:
```
// Prim's algorithm using priority_queue
// finds and updates the MST - vector of parent node indices
template<class T>
```

```
void PrimsMST(T & G, vector<int> & MST, int source) {
    // min priority_queue of vertices that need to be processed
    // stores pair of <weight, vertex>
    priority_queue<iPair, vector<iPair>, greater<iPair> > pq;
    vector<bool> visited(G.nodeCount(), false);
    MST.resize(G.nodeCount());
    fill(MST.begin(), MST.end(), -1); //remembers parent or where each node is␣
 ↪visited from
    vector<int> weight(G.nodeCount(), INT_MAX); //initialize weigth vectors␣
 ↪with INT_MAX
    weight[source] = 0; // weight of source
    pq.push({0, source}); // {weight, vertex}
    MST[source] = -1; // source node doesn't have a parent
    while (! pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        visited[u] = true;
        for(int v: G.neighbors(u)) {
            if (visited[v]) continue;
            int w = G.getWeight(u, v);
            // if the weight from u to v is smaller than the previously known␣
 ↪weight
            // update the weight
            if (w < weight[v]) {
                weight[v] = w;
                pq.push({w, v});
                MST[v] = u;
            }
        }
    }
}
```

1.5 Test Prim's Algorithm

- use adjacency matrix representation of graph
- matrix representation is easier to quickly find the weight of the MST using indices

[3]:
```
class Graph {
    private:
        vector<vector<int> > graph;

    public:
        Graph(size_t n) {
            this->graph.assign(n, vector<int>(n, 0));
            /*
            for (int i=0; i<n; i++) {
                vector<int> v(n, 0);
```

```
                //initialize to 0; means not connected
                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
        void addEdge(int u, int v, int w) {
            graph[u][v] = w;
        }

        // returns vector of pairs containing neighbors weight
        vector<int> neighbors(int u) {
            vector<int> neighs;
            for(int v = 0; v < graph[u].size(); v++)
                if (graph[u][v] != 0) neighs.push_back(v);

            return neighs;
        }

        int getWeight(int u, int v) {
            return graph[u][v];
        }
};
```

Let's work with the following graph:

[4]:
```
// represent undirected graph shown in above diagram
// A->0, B->1, C->2, D->3, E->4, F->5
Graph graph(6); // graph with 6 nodes 6x6 matrix
vector<int> MST; // store parent index
```

[5]:
```
// 16 bidirectional edges
graph.addEdge(0, 2, 7);
graph.addEdge(0, 4, 9);
graph.addEdge(1, 2, 5);
graph.addEdge(1, 5, 6);
graph.addEdge(2, 0, 7);
graph.addEdge(2, 3, 1);
graph.addEdge(2, 5, 2);
graph.addEdge(2, 1, 5);
graph.addEdge(3, 2, 1);
```

```
graph.addEdge(3, 5, 2);
graph.addEdge(4, 0, 9);
graph.addEdge(4, 6, 1);
graph.addEdge(5, 1, 6);
graph.addEdge(5, 2, 2);
graph.addEdge(5, 3, 2);
graph.addEdge(5, 4, 1);
```

[6]:
```
// function to print MST with corresponding weight
void printMST(vector<int>& MST, Graph& G) {
    int total = 0;
    cout << "Edge\t Weight\n";
    for (int i=0; i < G.nodeCount(); i++) {
        if (MST[i] == -1) // source node
            continue;
        total += G.getWeight(MST[i], i);
        cout << char(MST[i]+65) << "->" << char(i+65) << "\t = " << G.
 ↪getWeight(MST[i], i) << "\n";
    }
    cout << "Total Weight = " << total << endl;
}
```

[7]:
```
int source = 5;
```

[8]:
```
PrimsMST<Graph>(graph, MST, source);
```

[9]:
```
printMST(MST, graph);
```

```
Edge     Weight
C->A     = 7
C->B     = 5
F->C     = 2
C->D     = 1
F->E     = 1
Total Weight = 16
```

[10]:
```
// start from different source vertex
source = 2;
PrimsMST<Graph>(graph, MST, source);
```

[11]:
```
printMST(MST, graph);
```

```
Edge     Weight
C->A     = 7
C->B     = 5
C->D     = 1
F->E     = 1
```

```
C->F     = 2
Total Weight = 16
```

1.6 Time Complexity of Prim's Algorithm

- priority queue-based implementation cost is exactly same as that of Dijkstra's
- bulk of the cost comes from the loop which depends on the running time of priority queue
- because nodes are repeatedly added into the priority queue with different weight while exploring |E| edges, it'll raise the number of elements in the 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

1.7 Exercises

1. Jurassic Jigsaw - https://open.kattis.com/problems/jurassicjigsaw
   - Hints: Complete Graph - use adjacency matrix where weight is # of DNA differences
   - Print total weight of a MST and it's edges

- Minimum Spanning Tree problem: https://open.kattis.com/problems/minspantree
- Island Hopping - https://open.kattis.com/problems/islandhopping
- A Feast For Cats - https://open.kattis.com/problems/cats
- Lost Map - https://open.kattis.com/problems/lostmap
  - Hints: complete graph; don't read redundant data; don't use ADT but directly use matrix
- Nature Reserve - https://open.kattis.com/problems/naturereserve
- Communication Satellite - https://open.kattis.com/problems/communicationssatellite
  - Hints: complete graph where weight is the gap (dist - (r1+r2) between each pair of antennas

[ ]: