Minimum Spanning Tree Algorithms Comparison and Analysis

Lan Chen, Jessica Chen, Xiaomeng Chen and Zhanglong Peng

Abstract
Graph Theory is a common study of graphs in computer sciences and mathematics fields.
One of its well-known problems is finding the Minimum Spanning Tree(MST). A minimum spanning tree is a spanning tree of a connected, undirected graph which connects all the vertices together with a minimal total weight for its edges. In 1926, Czech mathematician Otakar Borůvka found the first algorithm of finding MST. As a matter of fact, there has been many algorithms invented since then. In this project, we will evaluate three existing algorithms by the following steps: first, modify the original code of three algorithms to make their input data types the same; second, run all of them on the same computer architecture -O3 -march=native; lastly, compare their performance based on access pattern, cache misses and execution time etc. The three algorithms we will use are Prim's Algorithm, Kruskal's Algorithm and Borůvka's algorithm.

Keywords MST, Kruskal, Prim, Boruvka, Architecture, Performance

I. Introduction

Minimum Spanning Tree (MST) is a subset of E of a undirected graph G = (V, E) that has minimum weight. MST is commonly used algorithm in many fields. In this project, we are going to use three algorithms: Kruskal‘s algorithm, Boruvka’s algorithm and Prim’s algorithm to compare their performances, in detail, cache misses and execution time etc in the same computer architecture. We will input the data in the same format for these three algorithms. Then, evaluate these algorithms in the same graph so that we can see the differences between these three.

II. Related Work

We haven’t found out any similar work yet.

III. The Algorithm

A. Boruvka’s Algorithm

Boruvka’s algorithm is used to find a minimum spanning tree in a graph that all edge weights are distinct. It was first introduced in 1926 by Otakar Boruvka. Similar to Kruskal’s algorithm, Boruvka first examines each vertex, then adds the cheapest edge from that vertex to another, continues grouping all edges into MST until all vertices are completed.

B. Prim’s Algorithm

Prim’s algorithm aims to find a minimum spanning tree for a weighted undirected graph. It was developed in 1930 by Czech mathematician and republished by Robert C.Prim in 1957. It runs in O((V+E)logV) time. This algorithm builds the tree by adding data at the subset of edges including every vertex,where the total weight is minimized. It operates from one starting vertex, continually adds the cheapest possible connection from the tree to another vertex.

C. Kruskal’s Algorithm

Kruskal’s algorithm is similar to Prim's algorithm. They both add an edge to the MST. However, Kruskal is different. It builds the MST in forest. Each vertex is in its own tree, then each edge is ordered by increasing weight. If an edge is connected between two different trees, (u,v) is added to the set of edges, and these two trees will be merged into a single tree. If (u,v)connects in the same tree, (u,v)will be discarded.

IV. Experimental Results

Undecided

V. Alternative Parallelization Approach

Reference

#include "Graph.h"

```
Graph::Graph() {
   this->directed = true;
}

bool Graph::addVertex(string vertexKey) {
   if (!this->exists(vertexKey)) {
         this->verticesMap.insert(std::make_pair(vertexKey, Vertex(vertexKey)));
         return true;
   }

   return false;
}

bool Graph::addEdge(string va, string vb) {
   if (!this->exists(va)) {
         throw std::invalid_argument("The vertex " + va + " doesn't exist.");
   }

   if (!this->exists(vb)) {
         throw std::invalid_argument("The vertex " + vb + " doesn't exist.");
   }

   if (exists(va, vb)) {
         return false;
   } else {
         this->verticesMap[va].adjList.push_back(
                        Edge(&verticesMap[va], &verticesMap[vb]));
   }

   if (!this->directed) {
         if (!exists(vb, va)) {
               this->verticesMap[vb].adjList.push_back(
                              Edge(&verticesMap[vb], &verticesMap[va]));
         }
   }

   return true;
}

Vertex* Graph::getVertexPtrByKey(string key) {
```

```
    return &verticesMap[key];
}

void Graph::printAdjList() {
    for (const auto &pair : this->verticesMap) {
        cout << pair.first;
        for (const auto &adjEdge : pair.second.adjList) {
            cout << " -> " << adjEdge.to->key;
        }
        cout << endl;
    }
}

void Graph::printGraphJson() {
    cout << "{ \"Nodes\": [";
    for (const auto &pair : this->verticesMap) {
        cout << "{ \"Name\": \"" << pair.first << "\"},";
    }
    cout << "],";
    cout << "\"Links\" : [";
    for (const auto &pair : this->verticesMap) {
        for (const auto &edge : pair.second.adjList) {
            cout << "{\"Source\": \"" << edge.from->key << "\"" << ",";
            cout << "\"Target\": \"" << edge.to->key << "\"" << ",";
            cout << "\"Value\": \"0\"}," << endl;
        }
    }
    cout << "]}";
}

string Graph::searchSubtypes(int numSubTypes, string sp, int order) {
    if (!this->exists(sp)) {
        throw std::invalid_argument("The vertex " + sp + " doesn't exist.");
    }
    string result = "";
    queue<Vertex*> Q;

    initializeVerticesForSearch();

    Q.push(getVertexPtrByKey(sp));
```

```
    while (!Q.empty()) {
        Vertex* u = dequee(Q);

        //Iterate through all the edged connected with the vertex.
        for (const auto& edge : u->adjList) {
            Vertex* v = edge.to; //Adjacent vertex (i.e. connected to the other end of the
edge).

            if (v->color == WHITE) {
                v->color = GRAY;
                v->distFromRoot = u->distFromRoot + 1;
                v->parent = u;

                //If the vertex is in the depth specified as parameter (param order) then
                //add it to the result string.
                //And considering that all the vertices connected to it will be in a deeper
level
                //we won't add in the queue for further exploration.
                if (v->distFromRoot == order) {
                    result += "<" + to_string(v->distFromRoot) + ">-> " + v->key
                            + "\n";
                    v->color = BLACK;
                    if (numSubTypes != -1 && --numSubTypes == 0) {
                        return result;
                    }
                } else {
                    Q.push(v);
                }
            }
        }

        u->color = BLACK;
    };

    if (result.length() == 0) {
        result = "No subtype was found!";
    }

    return result;
}
```

```
int Graph::countSubtypes(string sp) {
    if (!this->exists(sp)) {
        throw std::invalid_argument("The vertex " + sp + " doesn't exist.");
    }
    int count = 0;
    queue<Vertex*> Q;

    initializeVerticesForSearch();

    Q.push(getVertexPtrByKey(sp));

    while (!Q.empty()) {
        Vertex* u = dequee(Q);

        //Iterate through all the edged connected with the vertex.
        for (const auto& edge : u->adjList) {
            Vertex* v = edge.to; //Adjacent vertex.
            if (v->color == WHITE) {
                count++; //Counts each one.
                v->color = GRAY;
                Q.push(v);
            }
        }

        u->color = BLACK;
    };

    return count;
}

void Graph::createBFSTreeBetween(Vertex* root, Vertex* va, Vertex* vb) {
    initializeVerticesForSearch();

    queue<Vertex*> Q;
    Q.push(root);

    while (!Q.empty() && (va->color == WHITE || vb->color == WHITE)) {
        Vertex* u = dequee(Q);

        for (const auto& edge : u->adjList) {
            Vertex* v = edge.to;
```

```
                if (v->color == WHITE) {
                        v->color = GRAY;
                        v->distFromRoot = u->distFromRoot + 1;
                        v->parent = u;
                        Q.push(v);
                }
        }

        u->color = BLACK;
    };
}

Vertex* Graph::extractLowestCommonAncestorFromTree(Vertex* va, Vertex* vb) {
    //Pointers used to search through the tree for the common ancestor
    Vertex* sa = va;
    Vertex* sb = vb;

    //Brings the deepest search pointer to the level of the shallowest one,
    //considering that if there is difference between the depth of
    //the search pointers then the ancestor was still not found.
    while (sa->distFromRoot != sb->distFromRoot) {
        if (sa->distFromRoot > sb->distFromRoot) {
                        sa = sa->parent;
        } else {
                        sb = sb->parent;
        }
    }

    //Once both search pointers at a same level we can star searching
    //for the common ancestor.
    while ((sa != NULL && sb != NULL) && (*sa != *sb)) {
        sa = sa->parent;
        sb = sb->parent;
    }

    //If one vertex X turns to be descendant of
    //the other vertex Y, then the common parent of them both is
    //the parent of Y.
    if (*sa == *va || *sb == *vb) {
        return sa->parent;
    } else {
```

```
            return sa;
    }
}

string Graph::searchSubtypes(string sp, int order) {
    return searchSubtypes(-1, sp, order);
}

string Graph::searchCommonLowestAncestor(string sp1, string sp2, string sp3) {
    //Checks whether all the vertices exist.
    if (!this->exists(sp1)) {
            throw std::invalid_argument("The vertex " + sp1 + " doesn't exist.");
    }
    if (!this->exists(sp2)) {
            throw std::invalid_argument("The vertex " + sp2 + " doesn't exist.");
    }
    if (!this->exists(sp3)) {
            throw std::invalid_argument("The vertex " + sp3 + " doesn't exist.");
    }

    //Gets a pointer to all the vertices
    Vertex* sp1Vertex = getVertexPtrByKey(sp1);
    Vertex* sp2Vertex = getVertexPtrByKey(sp2);
    Vertex* sp3Vertex = getVertexPtrByKey(sp3);

    //Creates a BFS tree over the graph, starting on sp1Vertex and stopping once
    //sp2Vertex and sp3Vertex are found.
    createBFSTreeBetween(sp1Vertex, sp2Vertex, sp3Vertex);

    if(sp2Vertex->color == WHITE || sp3Vertex->color == WHITE){
            return "No common ancestor found!";
    }

    //Once the tree is complete and we have all the pointers between the
    //vertices and their parents set we can extract the common ancestor
    //between then;
    Vertex* commonAncestor = extractLowestCommonAncestorFromTree(sp2Vertex,
                    sp3Vertex);

    //Returns the name of the common ancestor if it's found.
    return (commonAncestor == NULL ?
```

```
                "No common ancestor found!" : commonAncestor->key);
}

Vertex* Graph::dequee(queue<Vertex*>& q) {
    Vertex* front = q.front();
    q.pop();
    return front;
}

void Graph::initializeVerticesForSearch() {
    for (auto &pair : verticesMap) {
        Vertex& vertex = pair.second;
        vertex.parent = NULL;
        vertex.color = WHITE;
    }
}

bool Graph::exists(string vertex) {
    return this->verticesMap.count(vertex) > 0;
}

bool Graph::exists(string va, string vb) {
    for (Edge &edge : verticesMap[va].adjList) {
        if (edge.to->key == vb) {
            return true;
        }
    }

    return false;
}

void Graph::addDataFromFile(string filePath) {
    ifstream file;
    string line;
    file.open(filePath);

    if (!file) {
        throw std::runtime_error("Could not open file " + filePath);
    }

    while (getline(file, line)) {
```

```
            for (int j = 0; j < line.size(); j++) {
                line[j] = tolower(line[j]);
            }

            replace(line.begin(), line.end(), ':', ',');

            list<string> types = StringUtils::split(line, ',');

            for (string& type : types) {
                StringUtils::trim(type);
                replace(type.begin(), type.end(), ' ', '_');
            }

            for (const string& type : types) {
                addVertex(type);
            }

            string originVertex = types.front();
            types.pop_front();

            for (const string& type : types) {
                addEdge(originVertex, type);
            }
        }
    }
```