Documentation for implementation of Prim and Kruskal algorithms

José Pedro Baltazar Mendes

xmende04

FIT 2019

Kruskal

The Kruskal algorithm is implemented as a function of the class KruskalGraph. This class is composed of an integer that holds the count of the vertices, a vector of struct Edge (containing the origin, destination and weight). The implementation starts by filling the edge vector with edges according to number of vertices and if the graph is dense (discussed later). It then starts counting it’s runtime. Consequently, it allocates all necessary structures such the “A” set that holds the selected edges and an array of class DisjointElement. It then sorts the edges using the std::sort function of E\*LogE of complexity.

During the execution of the loop, to find if the representative vertex of the origin and destination of the evaluated edge are the same, we call getSetHead() method which loops through the parent of the current disjoint element until it hits an element that is parent to itself. As it uses the path compression heuristic, the disjoint element will set its parent pointer to the set representative. In case the edge differs in its origin and destination representative vertex, it is added to the A set (complexity amortized 1), and the set is merged.

The merging of the two sets is done by comparing the disjoint element’s parent’s rank (order by rank heuristic) in order to achieve a reduced number of possible changes in parent for the annexed nodes. At maximum, the merge will have complexity V/2 as due to the order by rank heuristic, we can’t have a set that has more than half the elements be annexed, as it would constitute the majority.

After the selected edges vector (“A”) gets to the size of V – 1, the maximum number of vertices in a minimum spanning tree, it doesn’t need to go through all the other edges as they will be redundant and of higher weight. This really speeds up the processing as it is very probable that we will have all the edges we need early.

An earlier implementation used a vector inside each disjoint element in order to keep up with its children, this was found to give each parent swap a lot more operation time. This implementation gives no improvement regarding the order by rank heuristic since asking the vector for its size is a static operation which is constant for any number of elements.

Prim

The Prim algorithm is implemented as a function of the class PrimGraph. This class is composed of an integer that holds the number of edges (important in the creation of the edges), another for the number of vertices (vertexCount), a Boolean to hold if it is dense in edge count, a binary heap to serve as a priority queue and an adjacency list in which we store the edges.

In the implementation of the priority queue, we have an array of class Vertex that contain the order of the vertex (identified by index in the array) and it’s priority, as well as an array of integers that actually represent the vertices in the queue. Since they are pre-allocated, the value of non-used positions in the queue is set to -1. When a vertex is popped out of the queue, it’s position in the queue is set to -1 and its priority to INT\_MAX (maximum value for an integer) in order to further avoid a priority error.

A prior implementation used an array of vertices that contained their vertex number and their priority. This resulted in having to search for the vertices multiple times by following the array until hitting the required vertex. This implementation relies on indexes and removes the need to go through an array, increasing performance.

The Prim algorithm function starts by changing the capacity of the binary heap to V as to remove the need for algorithm runtime reallocations. It then generates the edges by filling the adjacency list (this is done in a single thread, therefore takes much more time then the actual algorithm). After it the method createVertices of the priority queue is called in order to initialize all both the vertex array and the integer array (referred to earlier). This binary heap implementation was inspired in the one referred in the “References” section and was altered to better fit the algorithm.

In the main loop of the algorithm, we remove the minimum element of the priority queue in a LogV complexity. Next, we receive an adjacency (in the form of a structure that contains both the destination of the edge and it’s weight), in a static fashion (we move a pointer forward in each call, an available option since we will go through all adjacencies). In a similar static fashion, we find the priority of a vertex since it only needs to follow an index to get the priority of the target vertex. In the case the priority of the vertex is bigger than the weight of the edge, we will decrease the priority of the vertex. The performance of this operation will be of LogV complexity since we must restructure the heap.

Generating the graphs

For both graphs we distinguish two types of graphs, dense and sparse. The number of edges is calculated using curves discussed with professor Zbynek Krivka in the class. As the maximum number of edges (undirected graph and no self-loops) is V\*(V-1)/2, we have to have curves that are smaller from at least 30 vertex count (it was tested for around ~1000 vertices max). The actual assignment of the values is not very strict, a probability function is used order to know if the edge is supposed to be there (as a side effect this also adds some variety to the graphs of the same number of vertices).

For Kruskal, the generation of the edges is very straight forward as the edges are represented as a vector. We use a nested loop and the method push back of the vertex to add it.

For Prim the generation of the edges isn’t as efficient (13 seconds for making around half a million edges). This is due to the storage of the edges that really benefits their access and the space complexity (in comparison to the matrix approach) and its need to call the function twice (edges on both directions). This happens in a nested loop of the same structure as in Kruskal, which calls the function of insertion to the adjacency list. This function calls itself with the origin and destination swapped in the parameters as to create the connection the other way.

The density curves were calculated using two points, 30 vertices and 1000 vertices. Their equations are:

These serve their purposes satisfactorily.

Time complexity

Through the analysis of the code and the functions of the code previously referred to, we can theorize the time complexity of the algorithms (in a dense graph, the edge number is around V^2 and in a sparse it’s around V).

)

for (itr = edges.begin(); itr != edges.end() && selectedEdges.size() < vertexCount - 1; itr++)//O(E\*(V+logV)) time complexity //we stop the collection of edges as we have enough to connect all vertices

{

if (disjointSets[itr->origin].getSetHead() != disjointSets[itr->destiny].getSetHead())//logV

{

selectedEdges.push\_back(\*itr);//1

disjointSets[itr->origin].mergeSet(&disjointSets[itr->destiny]);//O(V) time complexity

}

}

)

while (!bh.isEmpty()) {//V

int u = bh.extractMin();//LOGV

AdjacencyResult v = adj.giveAdjacency(u);//1

while (v.destination != -1)//O(V\*LOGV) //DENSE -> LOOP RUNS AROUND V TIMES //SPARSE -> LOOP RUNS AROUND V/SCALAR

{

if (v.weight < bh.findPriority(v.destination))//O(1) {

parentVertex[v.destination] = u;

bh.decreaseKey(v.destination, v.weight);//O(LOGV)

}

v = adj.giveAdjacency(u);//1

}

}

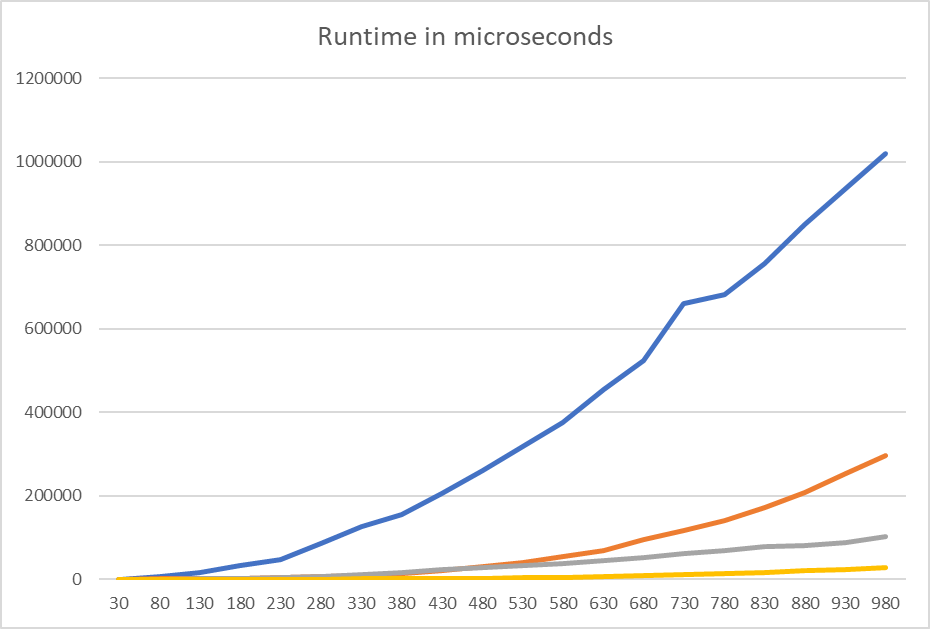
The main.cpp script by default runs the Kruskal and prim algorithm in 20 batches (varying the number of vertices) and doing 15 runs per batch. Since the algorithms return their runtime in microseconds, the script can take that time and average out a batch (removing the maximum and the minimum) and create a spreadsheet (csv file) which can be used to create a curve of the performance. One example ran on a laptop:

Figure 1-Runtime of algorithms in number of vertices by microseconds

The blue line is the Kruskal behaviour on a dense graph, orange is on a sparse graph. The grey line is the performance of the Prim algorithm on a dense graph and the yellow line is on a sparse graph. Through visual analysis we see that the Kruskal algorithm is more time complex and therefore if possible, the Prim algorithm should be run. This however comes with the draw back of the slow Prim initialization which can take multiple times more time than the actual algorithm (linked list adjacency list implementation for example).

By visual analysis it can be seen with confidence that the curves are in accordance with the previous theoretical analysis.

References

Zbynek Krivka, Graph Algorithms (slides), Brno University of Technology

Cormen, T. (2009). Introduction to algorithms. Cambridge, Mass.: MIT Press.

Binary Heap. (2018, September 4). Retrieved from https://www.geeksforgeeks.org/binary-heap/.