Project INF442: Minimum Spanning Trees

Youssef Allouah, Jeremy Sroussi

May 2020

Sequential implementation of classic algorithms[1]

Task 1

Boruvska algorithm

The principle of Boruvska algorithm is to construct a forest (initially composed of one-vertex trees) whose trees gradually fuse to form a minimum spanning tree. A Union-Find structure is used to enable these mergers.

Pseudo-code Given an input graph $G = (V, E)$ with edge weights $w = E \to \mathbb{R}$, we output the edge set F of a minimum spanning tree $T = (V, F)$.

Initialize a forest F to be a set of one-vertex trees, one for each vertex of the graph
While F has more than one component **do**
  Find the connected components of F and use Union method to merge their respective trees
  Traverse through all edges and update cheapest edge of every vertex
  for every cheapest edge above **do**
    if it links two vertices belonging to different trees **then** merge the trees
return F

Complexity In the pseudo-code above, the outer while-loop takes $\mathcal{O}(log(n))$ iterations until it terminates. Inside this loop, we examine every single edge to find the minimum for each vertex. The final complexity is then of order $\mathcal{O}(mlog(n))$.

Performances We used aSHIIP[2] to generate random connected networks for the evaluation of implemented algorithms. We chose to base our topologies on Erdos-Rényi ($p = 0.1$).
Below is a graph representing the run time of Boruvska algorithm with networks of different sizes.The graphs used are complete and edges' weight are randomly generated between 0 and 1.

[1] The pseudo-codes and complexity were inspired from "Design and Analysis of Algorithms" by Benjamin Doerr
[2] aSHIIP, A random topology generator of interdomain: available at https://github.com/scalen/wais-model- internet

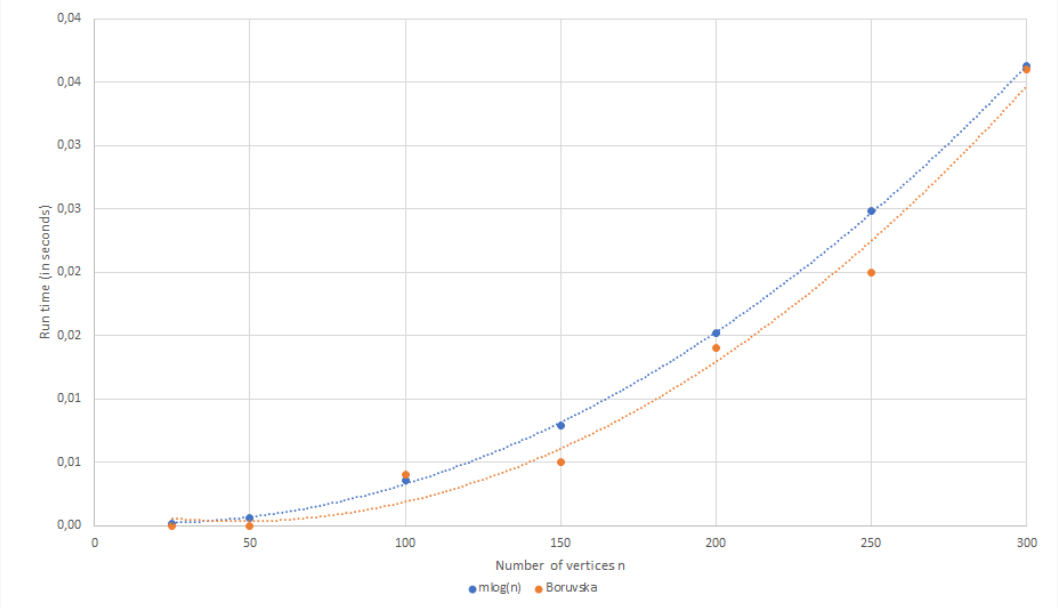

Figure 1: Run time of Boruvska algorithm as a function of number of vertices

Prim algorithm

The basic idea of Prim's algorithm is that we start with a tree consisting of a single node and then iteratively add this node to our tree that can most cheaply be connected with it (we also add its connecting edge).
Let G be our graph with V the set of vertices and E the set of edges (we will refer to these notations in the rest of the report, with n being the number of vertices and m being the number of edges).

Pseudo-code Given an input graph $G = (V, E)$ with edge weights $w = E \to \mathbb{R}$, we output the edge set F of a minimum spanning tree $T = (V, F)$

Let $v \in V$ be any vertex
$V_T = \{v\}$
$F = \emptyset$
while $V_T \neq V$ **do**
  Let $e = \{x, y\}$ be a cheapest edge such that $x \in V_T$ and $y \in V \setminus V_T$
  $V_T := V_T \cup y$
  $F := F \cup \{\{x, y\}\}$
  return F

Complexity We use a priority queue Q to store the edges going from V_T to the rest of the vertices. As key of the items in the queue we use simply their weight, hence with an extractmin operation we can read off the cheapest edge from T to the rest of G. Whenever we add a vertex y to V_T , we add all edges incident with y to the queue Q. In the while-loop, we extract from the queue an edge with smallest weight. If this edge contains two vertices already in V_T , we discard it. If the edge contains one vertex from V_T and one from $V \setminus V_T$,then we add the edge to F and the new vertex to V_T (and thus also add many new edges to the queue). With this approach, we add each edge twice to the queue and possibly extract it also twice. Since extracting the smallest element from a priority queue takes time $\mathcal{O}(log(m))$, we have a total effort of order $\mathcal{O}(m(log(n))$ (since $\mathcal{O}(log(m)) \sim \mathcal{O}(log(n))$.

Performances Below is a graph representing the run time of Prim algorithm with networks of different sizes.The graphs used are complete and edges' weight are randomly generated between 0 and 1.

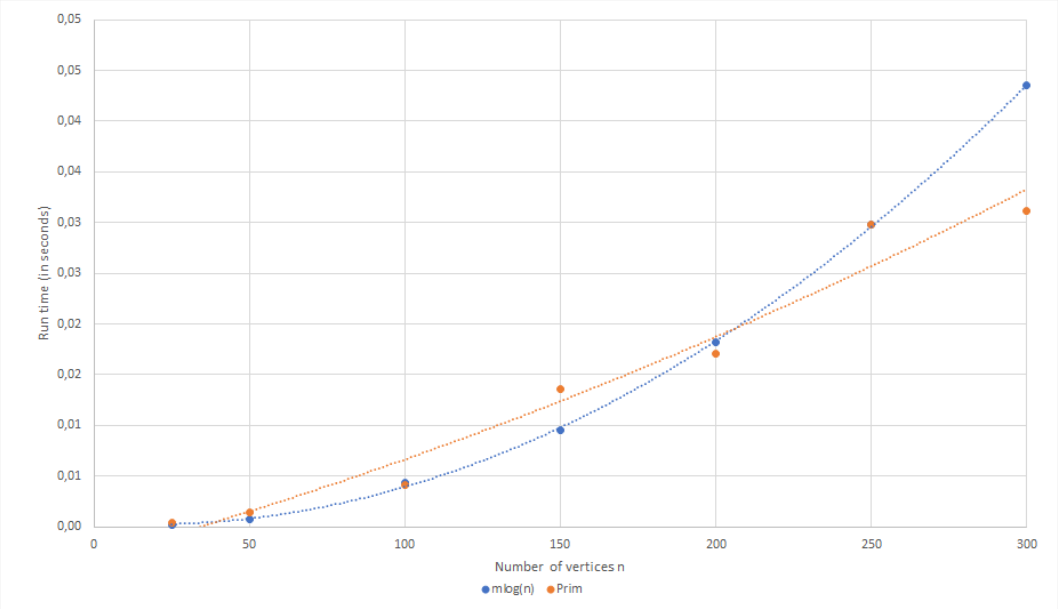

Figure 2: Run time of Prim algorithm as a function of number of vertices

Task 2

In Kruskal algorithm, we generate a minimum spanning tree by taking successively the cheapest edges in the whole graph. Using a Union-Find data structure we check if each selected edge would create a cycle (if vertices have the same parent), if it is the case, we discard it. The pseudo-code is given below.

Pseudo-code Given an undirected connected graph $G = (V, E)$ with edge weights $w = E \to \mathbb{R}$, we output the edge set F of a minimum spanning tree $T = (V, F)$.

Sort E by increasing weight, let $e_1, e_2, ..., e_m$ be the result
$F = \emptyset$
for $i = 1$ **to** m **do**
  if $(V, F \cup \{e_i\})$ contains no cycle **then** $F := F \cup \{e_i\}$
return F

Complexity We use $2m$ Find operations and $(k-1)$ Union operations. Both operations have complexity $\mathcal{O}(log(n))$. The cost of sorting initially all the edges is $\mathcal{O}(mlog(m))$. This costs therefore dominates the global cost of the algorithm. The final complexity if thus $\mathcal{O}(mlog(m))$.

Performances Below is a graph representing the run time of Kruskal algorithm with networks of different sizes.The graphs used are complete and edges' weight are randomly generated between 0 and 1.

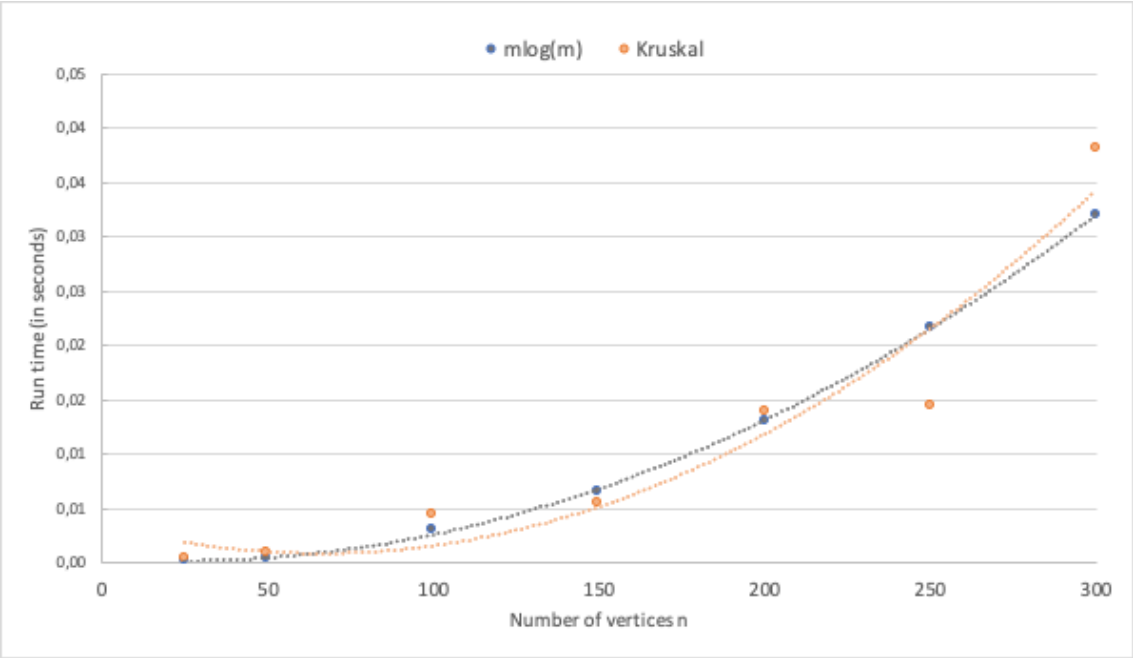

Figure 3: Run time of Kruskal algorithm as a function of number of vertices

Task 3

In this task, we implement a parallel version of the Prim algorithm. The classic sequential implementation, that we also used, consisting of keeping all vertices in a min-heap and updating their weights as we keep adding vertices to the MST can be misleading as it does not lend itself easily to parallelization.
However, we turned around this issue by identifying that the main step that has to be parallelized is that of finding the cheapest edge going from a vertex already in the MST to one that is not. Each process deals solely with a partition of the adjacency matrix. Its task is to find the local minimum of the weights of the eligible edges in its partition and to send it to the root. These local minima are then min-reduced to get the global minimum. Therefore, the pseudo-code is as follows:

Pseudo-code Given an input graph $G = (V, E)$ with edge weights $w = E \to \mathbb{R}$, we output the edge set F of a minimum spanning tree $T = (V, F)$

Let $v \in V$ be any vertex
$isInMST$ an array such as $isInMST[vertex] = True$ iff vertex is in MST
$isInMST[v] = True$
$F = \emptyset$
while $\exists$ vertex not in MST **do**
  Let $e_{local} = \{x, y\}$ be a local minimum for weights for each process such that $x \in$ MST and $y \notin$ MST
  Let $e_{global} = \min_{processes} e_{local}$ the global cheapest edges for all processes
  $F := F \cup \{\{x, y\}\}$
return F

Complexity Per iteration, the computation of the local minimum costs $\mathcal{O}(n/p)$ and the communication -the reduce operation- costs $\mathcal{O}(log(p))$.

Thus, for all iterations:

- the computation cost: $\mathcal{O}(n^2/p)$
- the communication cost: $\mathcal{O}(nlog(p))$

In total, the complexity is:

$$\mathcal{O}(n^2/p + nlog(p))$$

Performances We used the mpi4py[1] library available for Python and the Microsoft MPI environment on a laptop with 4 cores. The best results are obtained for a number of 4 parallel processes. This is not a coincidence as it is exactly the number of cores!

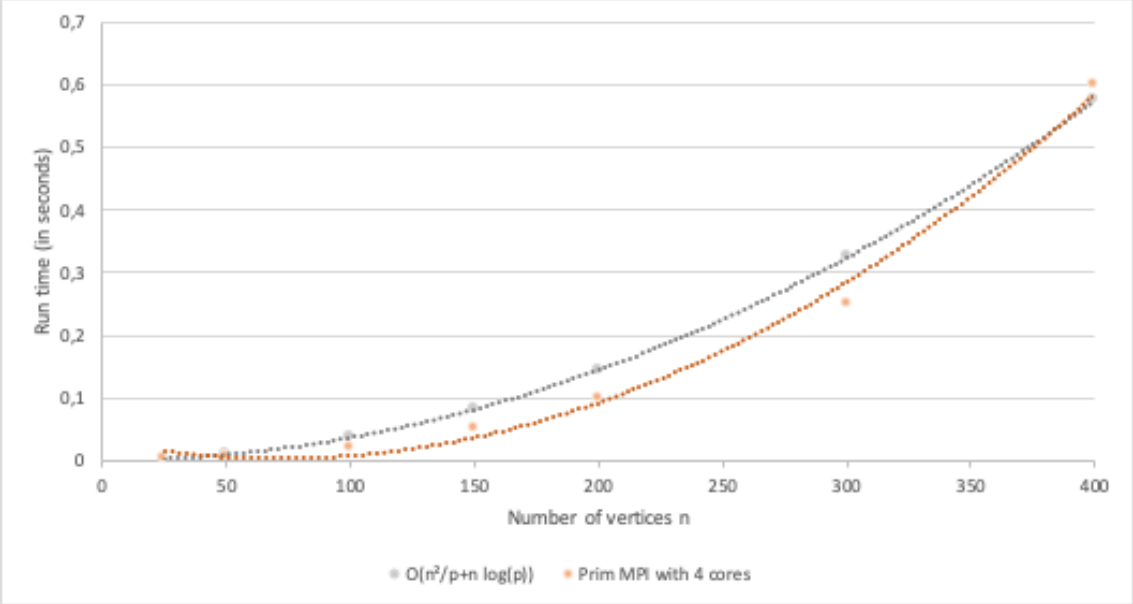

Figure 4: Run time of parallel Prim algorithm using 4 parallel processes as function of number of vertices

Task 5

We want to use Minimum Spanning Tree for data clustering. Given a data set S, we construct a graph with data items $s \in S$ (which are vectors in $\mathbb{R}^d$) represented by vertices $v \in V$. This is done with the *read_file.py* program. Every two vertices u and v are connected through an edge that represents the Euclidean distance between the two items (even if other distances can be chosen).

Given a value k, the MST T over a graph G corresponding to a dataset S allows to determine k clusters by removing the $(k-1)$ most expensive (with highest distance) edges in the MST. The reduced graph obtained consists of k connected components. The set of vertices of each connected component is then a cluster of S.

After removing the $(k-1)$ most expensive edges in the MST, we chose to use a Union-Find structure to gather vertices in the same cluster. The MST-based k clustering has the same complexity than

[1] See `https://mpi4py.readthedocs.io/en/stable/` for documentation

Kruskal algorithm, e.g. $\mathcal{O}(mlog(m))$. However, it appeared that some clusters were composed of only one or two items. They can be considered as noise, isolated from the other vertices. To solve this issue we have to create more than k clusters to end up with k true clusters.

Performances We worked with the iris dataset including already labelled items, so that we can compare their label with our results. For each produced cluster, we label it by the most represented label among the items it contains. For $k = 3$ we have an intra-cluster variance of 0.82 against 0.53 for k-means.

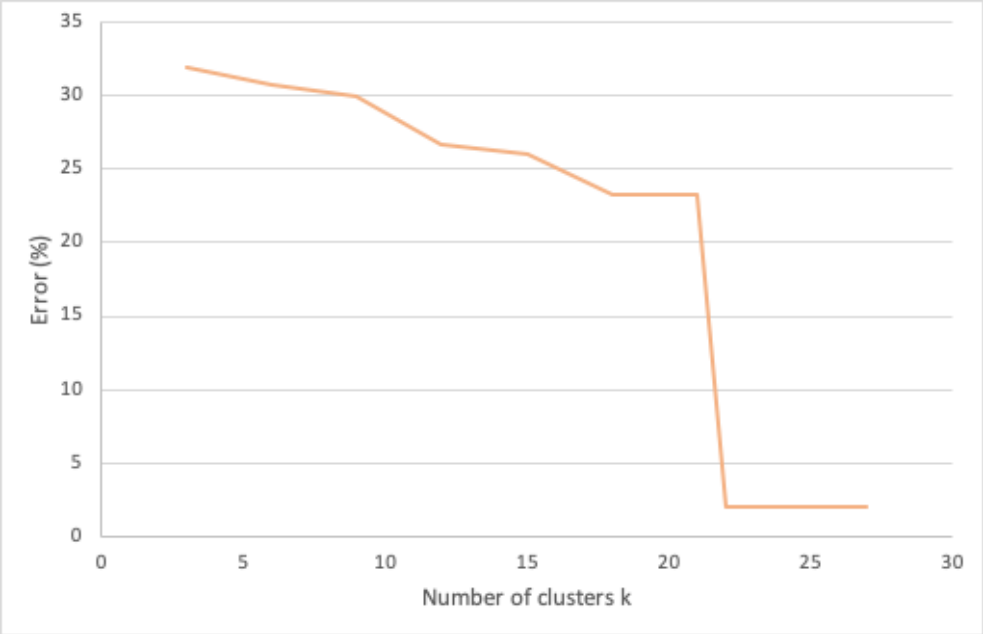

Figure 5: Error percentage as a function of number of clusters produced

We observe that the error percentage decreases as the number of clusters increases. This can be explained as clusters contain less vertices when k increases. Another consequence is that some clusters can have the same label.

We see that the method is particularly efficient with $k \approx 21$ which is quite big: this suggests we can improve our implementation, even though we need to take into account the "noise" clusters.

On the other hand we have implemented k-means algorithm with Forgy initialization. We chose to implement it in C++ as we already did it during TD3. In our case, this algorithm is more performant as it has a 88.7% accuracy for $k = 3$ on the iris dataset.