Maximum Leaf Spanning Tree & Parallelization

GALe Project

Léa NGUYEN

Table of contents

Introduction

The definition of the Maximum Leaf Spanning Tree problem is to find a tree spanning a graph G, defined G={V,E}, whose number of leaves is maximized. This problem is known to be MAX SNP-Complete.

Here are the characteristics of the problem, which explains its class:

- The spanning tree is cycle-free.

- The solution T maximizes {v ∈ V and $deg(v)_T = 1$}, its number of vertices.
- It is not possible to have a determinist algorithm to find the optimal solution in polynomial time.

Indeed, finding the optimal solution with a determinist algorithm would mean testing all possible spanning tree combinations. This would take exponentially longer. This is why the MLST problem is a non-determinist polynomial problem.

Therefore, we rely on approximation algorithms, which will approximate the maximum leaves.

The parallelization will be able to lighten the complexity of some instructions block, dividing approximately by the number of created threads.

For the following of the documentation, we will define the maximum leaf spanning tree as G' for a graph G={V,E}.

The implementation is in Java21.

2-Approximation Algorithm, Robert Solis-Oba

Source : https://www.sciencedirect.com/science/article/pii/S0304397511006219

Definition

The 2-Approximation algorithm, by Robert Solis-Oba, provides an algorithm in linear time that ensures to find at least half of leaf from the optimal solution.

In other words, the number of leaves of the algorithm computation $L_{2\text{-app}}$ is $L_{2\text{-app}}=L_{opt}/2$, with L_{opt} the number of leaves of the optimal solution calculated by $L_{opt}=|V|-1$.

The algorithm

The algorithm can be explained in a few steps:

1. Create a forest of trees, for which each root v, v ∈ V, is a black vertex with $deg(v)_G>=3$.
2. For each tree, check the expandability of the new leaves and expand them by priority, until none are expandable
3. Connect each tree of the forest into G'
4. Return G'

The implementation

Algorithm $tree(G)$

① $F \leftarrow \emptyset$

while there is a vertex v of degree at least 3 **do**

  ② Build a tree T_i with root v and leaves the neighbors of v.

  ③ **while** at least one leaf of T_i can be expanded ④ **do**

    Find a leaf of T_i that can be expanded with a rule of largest priority ⑤ and expand it.

  end while

  $F \leftarrow F \cup T_i$

  ⑥ Remove from G all vertices in T_i and all edges incident to them.

end while

⑦ Connect the trees in F and all vertices not in F to form a spanning tree T.

Pseudo code from the source

Explanation of the implementation:

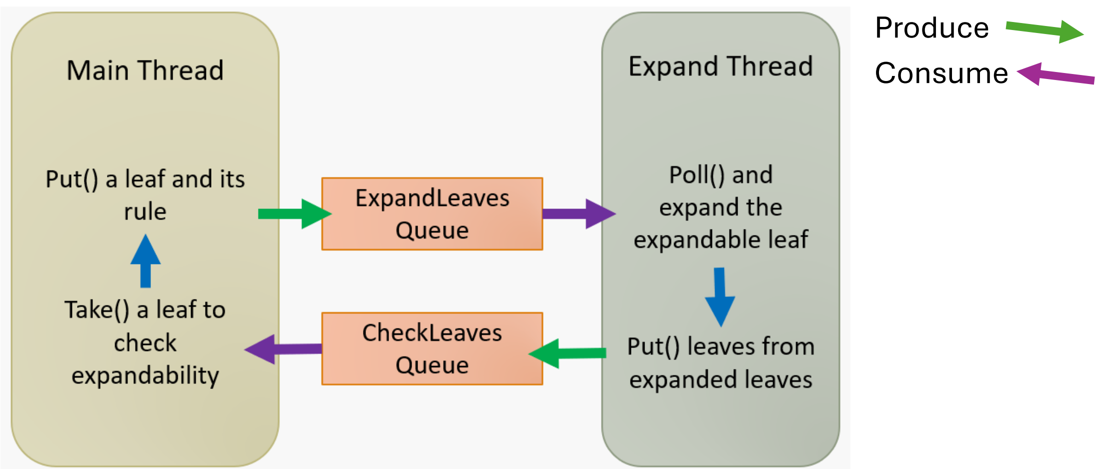
① F, the forest, is a list that contains the type of `Graph`.

② A new tree is the type of `Graph`.

③ This while-loop is done by a thread. In a Producer-Consumer-like architecture, this thread acts like a consumer of expandable leaves, it expands a leaf according to it expand-rule, and produces new leaves to check for their expandability.

④ The main thread consumes the leaves to check for their expandability and produces the expandable leaves.

The main thread and the expand-thread exchange their leaves by 2 `BlockingQueue`, that are thread-safe. One contains the leaves to expand, and the other contains the leaves to check for their expandability.



Léa NGUYEN

⑤ Each leaf is associated with it expand-rule, it is encapsulated in an internal record in the `Graph` class, defined `LeafRule(int leaf, Rule rule)`, with `Rule`, an enum.

The expand-thread takes the most prioritized leaf by taking from the `PriorityBlockingQueue`. It orders the leaves by their rule priority according to a custom compare function.

⑥ We remove the edges from G and, one by one, add incident edges to another Graph, stored in a list of incident edges tree. We can associate incident_edges_tree[i]=F[i]. This will help for ⑦ .

⑦ To connect the trees of the forest, represent each tree of F by a vertice of a graph P={V", E"}. The graph P, with |v"|=|F|, represents each tree by a vertice and how they are connected to each other, by applying DFS() on it.

To populate E", we run through the list of incident_edges_tree.

Therefore, for the edge u,v with $u \in Tree_i$ and $v \in Tree_j$, there will be an edge i->j∈E", and we store u->v associated to i->j aside. After applying DFS to P, we can add to G' the edge u_{Tree_i}->v_{Tree_j} associated to $i_{E''}$ ->$j_{E''}$.

Here are the rules in highest priority order:

1. x->y->[ab] : **x**, in the tree, has one child **y** out of the tree, **y** has exactly 2 children.
2. x->y->[ab...z] : **x**, in the tree, has one child **y** out of the tree, **y** has over 2 children.
3. x->[ab...z] : **x**, in the tree, has over 2 children out of the tree.
4. x->y : (custom rule) : **x**, has one child **y** out of the tree, and **y** is a leaf.

Time Complexity

```java
    static public Graph approximationSolisOba(Graph g) throws InterruptedException {
        ArrayList<Graph> forest = new ArrayList<>();
        ArrayList<Graph> incidentEdgesForest = new ArrayList<>();
        PriorityBlockingQueue<Graph.LeafRule> expandLeaves = new PriorityBlockingQueue<>(g.n / 2, LeafRule::compare);
        ArrayBlockingQueue<Integer> checkLeaves = new ArrayBlockingQueue<Integer>(g.n / 2);

        for (int root = 0; root < g.n; root++) { // N
            if (g.getDegree(root) >= 3) {

                Graph tree = new Graph(g.n);
                Graph incident_edges = new Graph(g.n);
                checkLeaves.put(root);

                // Thread--
                Thread expandthread = Thread.ofPlatform().start(() -> {
                    while (!Thread.interrupted()) {
                        // "Consume" expanding leaves
                        try {
                            var x = expandLeaves.poll(); // N
                            if(x==null) {
                                continue;
                            }
                            // Apply rule
                            var newNeighbors = g.getNeighborsByRule(x.rule, tree, x.leaf); // M

                            switch (x.rule) {
                            case Rule1, Rule2:
                                tree.addEdge(x.leaf, newNeighbors[0]);
                                for (int i = 1; i < newNeighbors.length; i++) {
                                    tree.addEdge(newNeighbors[0], newNeighbors[i]);
                                    // "Produce" check leaves
                                    checkLeaves.put(newNeighbors[i]);
                                }
                                break;
                            case Rule3, Rule4:
                                for (var vertice : newNeighbors) {
                                    tree.addEdge(x.leaf, vertice);
                                    // "Produce" check leaves
                                    checkLeaves.put(vertice);
                                }
                                break;
                            default:
                                break;
                            }

                        } catch (InterruptedException e) {
                            // Nothing
                        }

                        // Update existing leaves
                        LeafRule[] leavesCopy = expandLeaves.toArray(new LeafRule[0]); // Deep copy of the LeafRules
                         expandLeaves.clear();
                         for (var leaf : leavesCopy) { // N
                             var rule = g.whichRule(tree, leaf.leaf);
                             if (rule != Rule.None) {
                                 leaf.rule = rule;
                                 expandLeaves.add(leaf);
                             }
                         }

                    }
                    System.out.println("Close expand thread");
                 });
                 // --Thread


                 while (!expandthread.isInterrupted()) {

                     if (expandLeaves.isEmpty() && checkLeaves.isEmpty()) {
                         expandthread.interrupt();
                         continue;
                     }
                     var x = checkLeaves.take(); // N
                     var rule = g.whichRule(tree, x);
                     if (rule != Rule.None) {
                         // "Produce" expanding leaves
                         expandLeaves.add(new LeafRule(x, rule));
                     }
                 }

                 expandthread.join();
                 forest.add(tree);
```

n

```java
        for (int i = 0; i < tree.n; i++) { // N
            if (tree.getDegree(i) > 0) {
                g.simpleClearEdges(i);
            }
        }
```

n + m

```java
        for (int u = 0; u < g.n; u++) { // N
            if (g.getDegree(u) > 0) {
                var vertices = g.adj[u].stream().toList();
                for (var v : vertices) { // M
                    if (tree.getDegree(v) > 0) {
                        incident_edges.addEdge(u, v);
                        g.removeSingleEdge(u, v);
                    }
                }
            }
        }
        incidentEdgesForest.add(incident_edges);
    }
}

// CREATE MLST
var mergeThreads = new Thread[forest.size()];
var mlst = new Graph(g.n);

IntStream.range(0, forest.size()).forEachOrdered((nthTree) -> {
    mergeThreads[nthTree] = Thread.ofPlatform().start(() -> {
```

n+m

```java
        var tree = forest.get(nthTree);
        for (int u = 0; u < tree.n; u++) { // N
            var uIterator = tree.edgeIterator(u);
            while (uIterator.hasNext()) { // M
                mlst.addEdge(u, uIterator.next());
            }
        }
    });
});
```

n+m+c[1]

```java
var computed = computeLinkingEdges(forest, incidentEdgesForest);

for (Thread thread : mergeThreads) {
    thread.join();
}
```

n+m

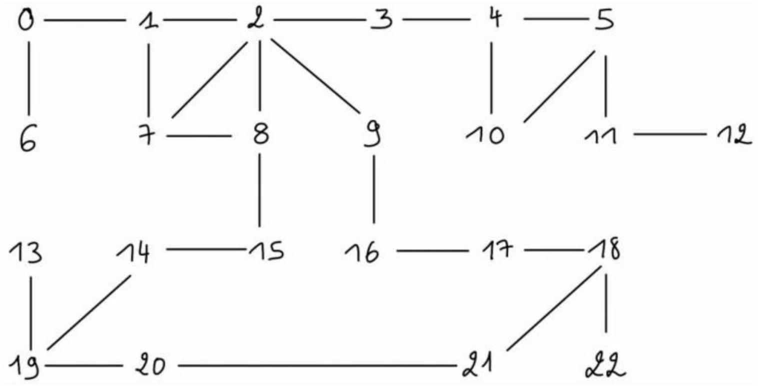
```java
mlst.addLinkingEdges(computed);

return mlst;
}
```

[1]C = |F|

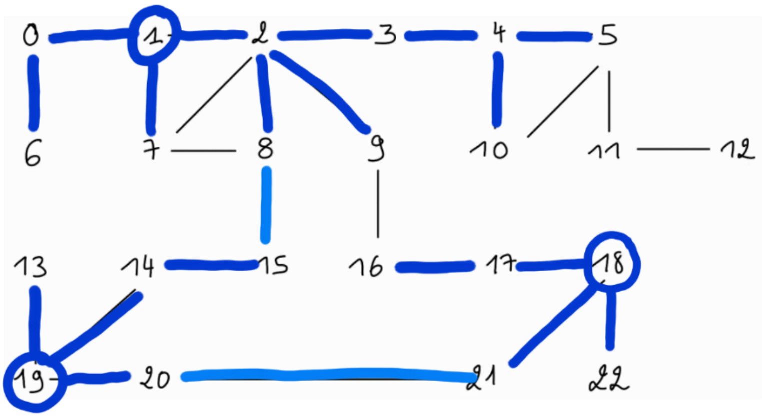
Here is briefly detailed complexity of block of code. We can see that the complexity of the implementation of the 2-approximation algorithm is O(c+m+n) which is linear.

Application

The original graph



The MLST in blue :



For |V|=23, Leafs = {6,7,9,10,5,13,16,22}

L_{opt} = 22 and $L_{2\text{-}app}$ = 8

Ratio = 22/8 = 2

3-Approximation Algorithm, Lu & Ravi

Source : [Approximating Maximum Leaf Spanning Trees in Almost Linear Time]

Definition

The 3-Approximation algorithm, by Hsueh-I Lu and R. Ravi, provides an algorithm in linear time that ensures to find at least a third of leaf from the optimal solution.

In other words, the number of leaves of the algorithm computation $L_{3\text{-app}}$ is $L_{3\text{-app}}=L_{opt}/3$, with L_{opt} the number of leaves of the optimal solution calculated by $L_{opt}=|V|-1$.

The algorithm

The algorithm can be explained in a few steps:

1. F is the G'
2. Create a set for each vertices, and initialize their degree to 0
3. For each vertices v, prepare a possible set of the vertice u, such as *vu*∈E to add to in its own set.
    a. Add u to the temporary set if:
        u is not in the set of v
        and the set that contains u is not in the temporary set.
    b. If v is a black vertex, such that deg(v)>=3
        For each u in the temporary set
            Add uv to F
            Add the set that contains u into the set of v
            Increment both degree by 1
4. Make sure that F is connected into G'
5. Return G'

The implementation

MaximallyLeafyForest(G)

① 1 Let F be an empty set.
2 For every node v in G do
3 ② $S(v) := \{v\}$.
4 $d(v) := 0$.
5 ③For every node v in G do
6 $S' := \emptyset$.
7 $d' := 0$.
8 For every node u that is adjacent to v in G do
9 ⑤④ If $u \notin S(v)$ and $S(u) \notin S'$ then
10 $d' := d' + 1$.
11 Insert $S(u)$ into S'.
12 ④ If $d(v) + d' \geq 3$ then
13 For every $S(u)$ in S' do
14 Add edge uv to F. ⑦
15 ⑤ Union $S(v)$ and $S(u)$.
16 Update $d(u) := d(u) + 1$ and $d(v) := d(v) + 1$.
⑥ 17 Connect F into G'
18 Output G'

Pseudo code above line 17 is from [source]

① Since the algorithm doesn't clearly separate the trees, F, the forest, is a Graph.

② d, is an AtomicIntegerArray

S is an array of a custom ConcurrentHashSet

In addition:

- There is a custom array of locks for each set of S, setLocks[], it will be useful for multi-threading. This choice is explained in ④.
- A parent AtomicIntegerArray and a height AtomicIntegerArray for union-find operations. This choice is explained in ⑤.

③ This for-loop is multithreaded.

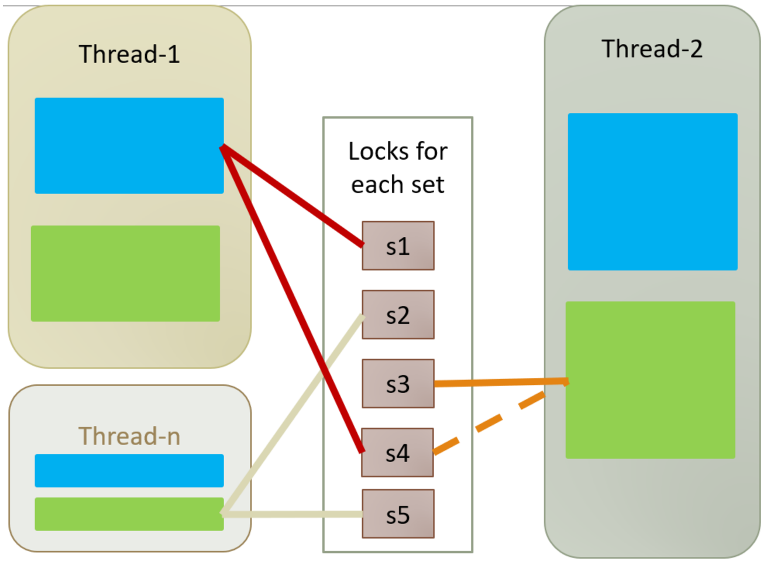
④ These blocks are synchronized with setLock[v] and setLock[u]. This prevents unwanted instructions of the line 13 to be executed.

Explanation :

Let i ∈ V

We don't want to interrupt the if-block of line 9, to ensure that for u=i, u∉ S(v) is still verified when d' is incremented, in case v=I in another thread and S(v) is updated in that other thread. We will also re-verify that u∈ S(v) when doing the for-each-loop on line 13.

In the thread of v=i, we will ensure d(v) is the same value on lines 12 and 16.



⑤ According to [source], S(u) means the set that contains u.

Therefore, by working with union-find and a parents[] array, we can track which set contains u by getting set[parents[u]].

⑥ We separate by trees with DFS. And connect them with the same method as the 2-approximation implementation, [here].

⑦ For each u of S', we need to verify that still u∉S(v), as explained in ④

Time complexity

```java
static public Graph approximationLuAndRavi(Graph g) throws InterruptedException {
    var forest = new Graph(g.n);
    var parents = new int[g.n];
    var heights = new int[g.n];
    var d = new AtomicIntegerArray(g.n);
    var subsets = new ConcurrentHashSet[g.n];
    var subsetLocks = new Object[g.n];

    var verticeThreads = new Thread[g.n / 2];
    var verticeCounter = new AtomicInteger();

    // initialize
    for (int i = 0; i < g.n; i++) {
        subsets[i] = new ConcurrentHashSet();
        subsets[i].add(i);
        parents[i] = i;
        subsetLocks[i] = new Object();
    }

    IntStream.range(0, verticeThreads.length).forEachOrdered(i -> {
        verticeThreads[i] = Thread.ofPlatform().start(() -> {
```

n + 2m

```java
            while (true) {
                int v = verticeCounter.getAndIncrement();
                if (v >= g.n) return;

                var set_prime = new HashSet<Integer>();
                var d_prime = 0;
                var neighbors = g.edgeIterator(v);
```

m

```java
                while (neighbors.hasNext()) {
                    var u = neighbors.next();
                    int min = Math.min(v, u);
                    int max = Math.max(v, u);
                    synchronized (subsetLocks[min]) {
                        synchronized (subsetLocks[max]) {
                            var pu = find(u, parents);
                            boolean uInSetPrime = set_prime.stream().map(x -> find(x, parents) == pu).anyMatch(r -> r == true);
                            if (find(u, parents) != find(v, parents) && !uInSetPrime) {
                                d_prime++;
                                set_prime.addAll(subsets[find(u, parents)].get());
                            }
                        }
                    }
                }

                if (d.get(v) + d_prime >= 3) {
```

m

```java
                    var set_primeIterator = set_prime.iterator();
                    while (set_primeIterator.hasNext()) {
                        var u = set_primeIterator.next();
                        int min = Math.min(v, u);
                        int max = Math.max(v, u);

                        synchronized (subsetLocks[min]) {
                            synchronized (subsetLocks[max]) {
                                if (find(u, parents) != find(v, parents)) {
                                    forest.addEdge(u, v);
                                    union(u, v, parents, heights, subsets);
                                    d.incrementAndGet(v);
                                    d.incrementAndGet(v);
                                    d.incrementAndGet(u);
                                }
                            }
                        }
                    }
                }
            }
        });
    });

    for (int i = 0; i < verticeThreads.length; i++) {
        verticeThreads[i].join();
    }


    // From forest graph to list of trees
    var trees = DFS_trees(forest);

    // Compute forest of incident edges for each tree
    var incidentEdgesForest = new ArrayList<Graph>(trees.size());
    for (int i = 0; i < trees.size(); i++) {
        incidentEdgesForest.add(new Graph(g.n));
    }

    for (int i = 0; i < trees.size(); i++) {
        var currentTree = trees.get(i);
        var incidentEdgesCurrentTree = incidentEdgesForest.get(i);
```

c*n

n

```java
        var leafs = currentTree.getLeafs();
        for (var leaf : leafs) {
            if (g.getDegree(leaf) > 1) { // has outward edges out of tree
                var edges = g.getNeighborsOutOfTree(currentTree, leaf);
                for (var v : edges) {
                    incidentEdgesCurrentTree.addEdge(leaf, v);
                }
            }
```

```java
        // CREATE MLST
        var mergeThreads = new Thread[trees.size()];
        var mlst = new Graph(g.n);

        IntStream.range(0, trees.size()).forEachOrdered((nthTree) -> {
            mergeThreads[nthTree] = Thread.ofPlatform().start(() -> {
                var tree = trees.get(nthTree);
                for (int u = 0; u < tree.n; u++) { // N
                    var uIterator = tree.edgeIterator(u);
                    while (uIterator.hasNext()) { // M
                        mlst.addEdge(u, uIterator.next());
                    }
                }
            });
        });
```

c*m

n+m+c

```java
        var computed = computeLinkingEdges(trees, incidentEdgesForest);

        for (Thread thread : mergeThreads) {
            thread.join();
        }
```

n+m

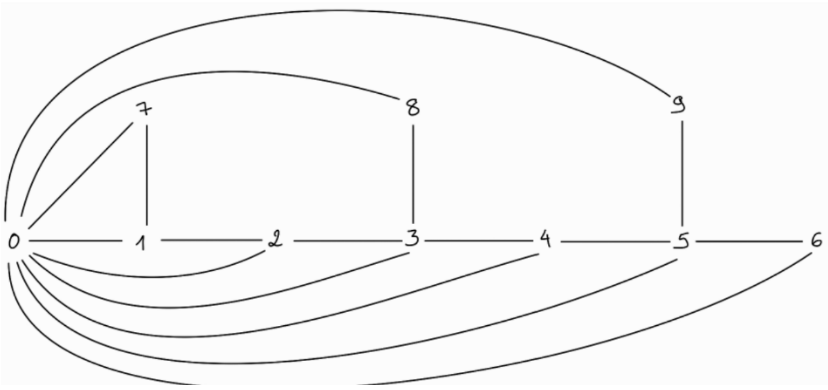
```java
        mlst.addLinkingEdges(computed);
        return mlst;
    }
```

We can conclude that the time complexity is linear by O(n+m).

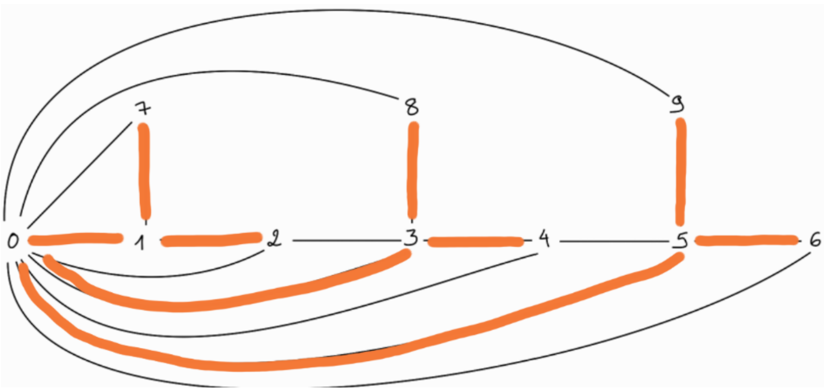
Application

Application 1

The original graph



The MLST



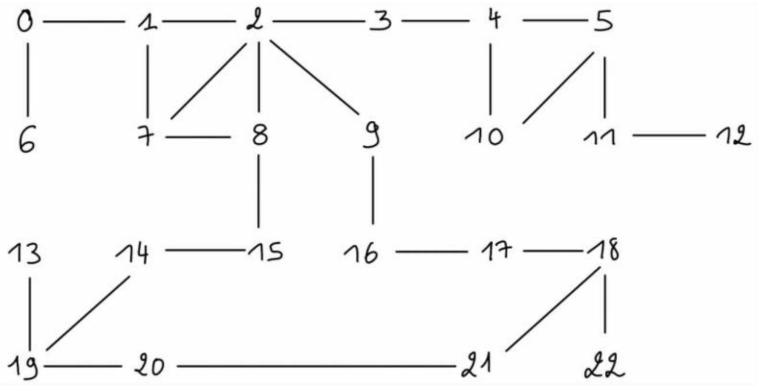
For |V|=10, Leafs = {2,4,6,7,8,9}

L_{opt} = 9 and $L_{3\text{-}app}$ = 6

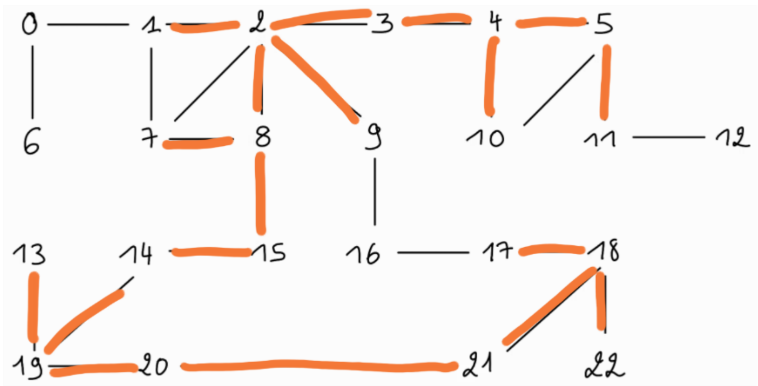
Ratio = 9/6 = 1

Application 2

The graph



The MLST



For |V|=23, Leafs = {1,7,9,10,11,13,17,22}

L_{opt} = 22 and $L_{3\text{-app}}$ = 8

Ratio = 22/8 = 2

Even if we haven't got any Ratio = 3, the algorithm ensure that we find at least a third of L_{opt}.

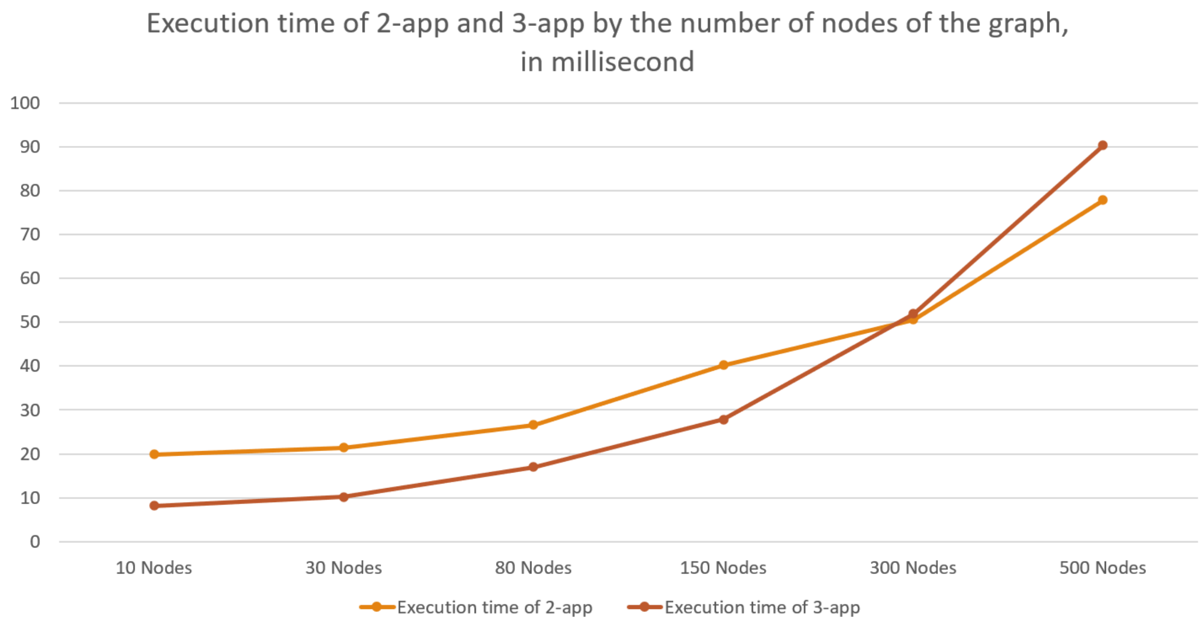
Experiments

Graphs are created with the probability of adding an edge of 0.14.

As we know, our 2-approximation algorithm uses 2 Threads (Producer-Consumer), and 3-approximation algorithm uses 1+|V|/2 Threads.

We can assume that by using more threads, the execution time will be lower than using only 2 threads.

Here is the dot-linear graph of the execution time, in milliseconds, of our 2-approximation and 3-approximation algorithms by the number of vertices in a graph:



<u>Observation:</u>

It is true that below a certain number of vertices, the execution of the 3-approximation algorithm is faster than the 2-approximation algorithm, similarly to our assumption.

But after a certain value of nodes, here 300 nodes, the execution of the 2-approximation algorithm is faster.

<u>Conclusion:</u>

From the experiment, using threads does accelerate the execution of an algorithm, but it also depends on deep components of a computer, so we must be careful while analyzing the information. However, at some point, using too many threads can lower the efficiency of the computer components making it useless and counterproductive.

Conclusion

The Maximum Leaf Spanning Tree is a MAX SNP-complete problem, apart from its constraints, there are no determinist algorithms to find the optimal solution in polynomial time. Indeed, the determinist algorithm would need to compare all the possible combinations of tree making it an exponential time algorithm.

But to go over that, there are a few algorithms which provide an approximation of the number of possibles leaves by a percentage, depending on the algorithm. In this document, a 2-approximation and 3-approximation algorithms were defined and provide a ratio of 5/2 and 1/3 ratio of number of leaves compared to the optimal number of leaves. They are a good alternative to a determinist algorithm because they provide an approximation of the solution in linear time.

From our experiments, it is clear that while using threads can significantly accelerate the execution of an algorithm, their effectiveness heavily depends on the underlying hardware and system architecture. Overusing threads may lead to inefficiencies, as excessive threading can overwhelm the computer's resources, resulting in counterproductive outcomes. Therefore, careful consideration and analysis are required to balance performance and resource utilization effectively.