EVALUATING A SUBLINEAR-TIME ALGORITHM FOR THE MINIMUM SPANNING TREE PROBLEM

GABRIELE SANTI* AND LEONARDO DE LAURENTIIS†

Abstract. We present an implementation and an experimental evaluation of an algorithm that, given a connected graph G (represented by adjacency lists), estimates in sublinear time, with a relative error e, the Minimum Spanning Tree Weight of G (B. Chazelle, R. Rubinfeld, and L. Trevisan. Approximating the minimum spanning tree weight in sublinear time. SIAM J *Computing*, 34, 2005). Since the theoretical performances have already been shown and demonstrated in the paper of Chazelle et al. our goal is, exclusively, to experimental evaluate the mentioned algorithm and at last to present the results. Some technical insights are given on the implementation of the algorithm and on how the technologies used for this purpose could participate in the overall performances; those are then evaluated empirically and widely discussed, comparing these with the performances of the Prim algorithm and the Kruskal algorithm, launching several runs on a heterogeneous set of graphs.

We assume hereafter that the reader has knowledge about the cited paper as we will just recup the theoretical results.

Key words. minimum spanning tree, sublinear time algorithms, randomized algorithm, approximation algorithm, minimum spanning tree weight, experimental evaluation

AMS subject classifications. 68W20, 68W25, 68R10

1. Introduction.

We would discuss here some preliminary observations and assumptions. First of all, we observe that we need a set of graphs that satisfies the following points:

- they should be finite and should not be multigraphs;
- they should be undirected;
- they should have weighted edges;
- the weithgts on the edges should be integers; since the graph is finite, it is enough to show that there exist W such that it is the maximum weight on the edges of the graph;[1]
- they might contain self-loops;
- they have to be connected (the graph has only one connected component);
- they should be represented with adjacency lists;
- they should be represented in the same manner (we need an unvarying file format).

Unfortunately, the graphs and their representations which can easily be found on the Internet doesn't accomplish all of this requirements at the same time, although many standards for the file format are available.

Given this observation, our choice was to use randomly generated graphs, hence to implement our own graphs generator. This gives us the opportunity to generate a wide set of connected graphs, with tunable parameters, carefully chosen looking forward to the tests; these parameters include the number of nodes, the number of edges and the edges weight, nonetheless the distribution law for the edges. The edges between the nodes are step-by-step randomly constructed, respecting the connection

*Master's Degree in Computer Science, University of Rome "Tor Vergata" (gabriele.santi@acm.org).

†Master's Degree in Computer Science, University of Rome "Tor Vergata" (ldelaurentiis@acm.org).

[1]more generally, it is enough to have a numerable set of values for the weights which is always true when the graph is finite

requirement. The different types of graphs that we use in our experimental evaluation are presented afterwards.

After studying the paper we made some assumptions. One of the problem we encountered is that the theoretical algorithm assume to have in input the graph G and to have direct access to the family of graphs ${G_i}^2$; with "direct" we intend that no computation is required to extract G_i, which is not true. In fact, we can show easily that a lower bound for the extraction of all the family costs at least $O(m)$ (i.e. is linear on the number of edges). A naïve approach would cost $O(m)$ for the extraction of G_i for a given i, hence $O(wm)$ for the whole family; a better approach could order the edges in $O(m \log m)$ and build the family in a single pass on the edges, achieving $O(m + m \log m)$. Having in input only G it would seem that the algorithm is responsible for the extraction of the family, but it couldn't be due to this lower bound that would sabotage the overall performance. We decided finally to consider this cost a part of the construction of the data structure of the graph, to be done prior to the call of the algorithm.

2. Design studies and choices.

2.1. Random Graphs Generator.

2.1. Random Graphs Generator. As mentioned before, our choice is to implement our own graphs generator. The aim is to generate *connected random graphs* with a specific set of parameters, like the number of nodes, the number of edges, the edges weight and the average degree. More, we want to test how our algorithm behaves in different environments, so we want to control the distribution law of the edges. Keep in mind that the graph has to satisfy all the points of the previous section, among which the graph connection. Given a desired number of nodes n and $e >= n-1$ edges, the key-concepts under the implementation of our connected graphs generator are the following:

- we begin by generating a random permutation of the vertices $v_0, v_1, \dots v_n$
- we generate a random spanning tree by iteratively adding edges in this vector in a uniform random manner; suppose we have added τ vertices to the tree T, we randomly select a vertex s in the set $\{T_0, \dots, T_\tau\}$ and add a new edge $< T_s, v_{\tau+1} >$. At the end we will have an acyclic graph with $n-1$ edges.
- following a certain probability law, we add the remaining $e - (n-1)$ edges

note that every time we add an edge, a weight is associated to it, with a value uniformly choosen in $[1, w]$.

We even decided to use a custom file format to save the data, wich is a `.ssv` file, which stands for *space separated values*: every line of the file corresponds to two edges, reporting the values $< v_s, v_t, w >$ that means source and target nodes, edge weight. Being the graph undirected, it is implicit that the edge $< v_t, v_s, w >$ exists also.

2.2. Graph Data Structures. We implemented two version of the algorithm, the first using the well-known *Boost*'s BGL[3] for the graphs data structures and efficient implementations of Kruskal's and Prim's algorithms. The latter uses our own implementation of both the structures and the algorithms. We decided to do so because we wanted more control over the code, for testing purposes, and because at the moment (version `1.61.0`) the BGL subgraphs framework presents some still unfixed bug.

[2]we recall that G_i is a subgraph of G such that the maximum weight on his edges is i (e.g. G_w is exactly G)

[3]`Boost Graph Libray`

Unfortunately, our Kruskal algorithm, althought using *union-find* structures with path compression[4], is not fine-tuned as that proposed by the Boost's libraries; we didn't take further time on this becase on the other side, our version of Prim's algorithm shows the *same exact performances* of the Boost version and it counts as lower bound being optimal and way better than the Kruskal solution. Our data structures have been called FastGraph for their optimization over the operation required for the test. In any way our data structures *can* nor *do* "help" the CRT[5] algorithm in improving his time complexity.

2.3. Tuning the algorithm. In the implementation of the algorithm, the graph stored in the text file is read in a FastGraph, which is of course a representation by adjacency lists. We want to compare the performances of CRT algorithm with a standard MST algorithm that in addition computes the total weight.
We want to emphasize now that the CRT algorithm is based on probabilistic bases; some parameters, that depend asymptotically on ε should be selected carefully in order to provide a good approximation very fast. These includes:

- r, the number of vertices uniformly choosen in the "approx-number-connected-components". This number is critical for the performances, as it directly determine the running time of the entire algorithm.
- C, the large costant that we use to pick the number of vertices determinants for the application of Lemma 4.

Both these values largely affect the overall performances because they undirectly decide how many BFS will be carried out by the CRT algorithm; the BFSes represent the fundamental cost driver of the entire run.

Initially in our opinion, the choice of these parameters must depend on the number of vertices in a way that they are dynamic, keeping their dependency on ε. We tried to untie this bond to 'n but having static values for this parameters show poor performances and a relative error exploding too fast as we grow the input instance.

As the paper says, the only requirement for r is that it is $O(\frac{1}{\varepsilon^2})$ and $C \in O(\frac{1}{\varepsilon})$.

The demonstration reported on the original paper bound these values in a way that helped us on choosing the right function to compute them; in fact, we have that

$$r <= \sqrt{\frac{n}{w}}$$

hence we choose

In a similar way, we have for C that

$$C\sqrt{\frac{w}{n}} < \varepsilon < 1/2$$

Therefore we choose the following function to compute the r parameter: $\lfloor \frac{(\sqrt{\frac{n}{w}} \cdot \varepsilon) - 1}{\varepsilon^2} \rfloor$.

Regarding the large constant C, we choose the following function to compute the vertices for the Lemma 4: $\lfloor (\sqrt{n} \cdot \varepsilon) - 1 \rfloor$; therefore we compute the vertices with the following: $\frac{C}{\varepsilon}$.

[4]it is known that the Kruskal algorithm time complexity is $O(m \log m)$, but can be improved to $O(m \log n)$ using union find structures with some euristic conveniently applied

[5]Short for B. Chazelle, R. Rubinfeld, and L. Trevisan.[1]

Another design choice include the way we use to choose the 'r' vertices from the graph G. We already discussed how to choose the number of vertices for the various BFS in the approx-number-connected-components. Now we want to discuss how to pick the 'r' vertices.
In our implementation, we use the Fisher-Yates shuffle algorithm. This allows us to compute the 'r' vertices in constant time (i.e. $O(1)$), simply calculating in advance the same vertices. This choice is done to not make the vertices drawing a bottleneck for the algorithm performances, as this is not specified in the paper.

3. Implementation choices. As already discussed, the algorithm is implemented in C++ language; although absolute performances are not important, whereas relative performances of the CRT algorithm and other MST algorithms are the focus, the choice of the language is something not of the greatest importance. The choice of C++ is led by the intention to have code modularity and to decouple the performances from any potential noise inserted by other factors (e.g. a JVM). The IDE tool used for the implementation is JetBrains Clion. We also used GitHub, as our code versioning control system.
The main function of the implementation is in the AlgoWEB.cpp file. Launching the program from this file allows us to run either the CRT algorithm or the Prim algorithm,or the Kruskal algorithm, and to view either the running time or the computed weight of the Minimum Spanning Tree.
Since we want to test the performances of the CRT algorithm on a large and various set of graphs we decide to use our random graphs generator to create random graphs of the following types:

- Erdos-Renyi, that models random graphs constructed with a uniform distribution.
- Gaussian, that models random graphs with some clusters.
- Albert-Barabasi, that models random scale-free graphs.
- Watts-Strogatz, that models random graphs with small-world properties.

For each type of graph we decide to produce a dataset of graphs with the following parameters:

- n: 5000,30000,55000,80000,105000,130000,155000,180000,205000,230000.
- m: 10N, 50N, 100N, where N is the number of vertices.
- w: 20, 40, 60, 80.

Regarding the ε parameter, we choose: 0.2, 0.3, 0.4, 0.49999.
As mentioned before, the parameters should be selected carefully in order to provide a good approximation very fast. Since the CRT algorithm is a probabilistic algorithm, in which the ε parameter is crucial either for performances or the accuracy of the estimate MST weight, it does not make sense to choose values too small for this parameter, as the performances could dramatically degrades (as, indeed, is expected).
Given the intrinsically probabilistic nature of the CRT algorithm, we have to launch several runs on the same graph, to have a correct estimate of the CRT running time. For this purpose we decide to launch 10 runs for each graph, and to take the average running time as the estimate of the running time.

4. Main results. Following there are some considerations and some line charts regarding the main results.
We know that the CRT running time is $O(dw\varepsilon^{-2}\log\dfrac{dw}{\varepsilon})$, where d is the average degree of a graph . Therefore what we expect is that:

- Keeping d and ε unchanged, if we consider an increase in the value of w there must be a worsening of the running time
- Keeping d and w unchanged, if we consider an increase in the value of ε there must be an improvement of the running time
- Keeping ε and w unchanged, if we consider an increase in the value of d there must be a worsening of the running time

We also know that the aim of the CRT algorithm is to be sublinear in the number of edges (i.e. below $O(m)$, where m is the number of edges of the graph); we want now to show that it really happens, possibly for all the different types of graphs in our dataset.

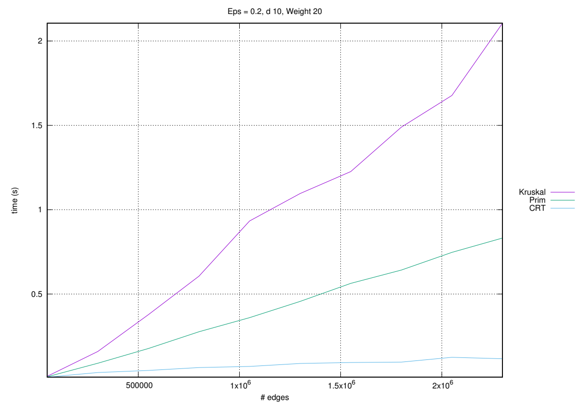

Fig. 4.1: Gaussian

At first, we can observe that given fixed values of d, w and ε the CRT Algorithm respects the sub-linearity property, for all the different types of graphs in our dataset, as the reader can see in the figures from 4.1 to 4.4.

Now we want to see what happens if there is an increase in the value of d, keeping other values unchanged.

While in the corresponding Small-World graph and the corresponding Uniform graph the trend is similar to the Gaussian graph (Fig. 4.5), in the Scale-Free case the reader can easily see that CRT algorithm takes a little 'more time' to down permanently under Prim execution time (Fig. 4.6). This can be related to the presence, in this kind of graph, of different nodes that have an high degree, which can make slower the execution.
The reader can also may notice that for small instances (few edges) the CRT algorithm is worse than deterministic algorithms, but when increasing the graph size the trend returns to be sub-linear again.
Also comparing, for example, (Fig. 4.5) with the corresponding (Fig. 4.1), one can

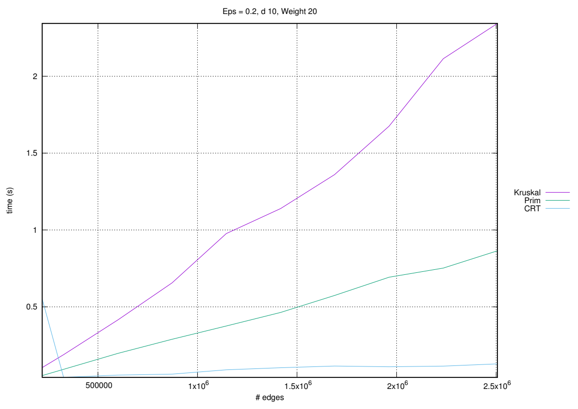

Fig. 4.2: Scale-Free

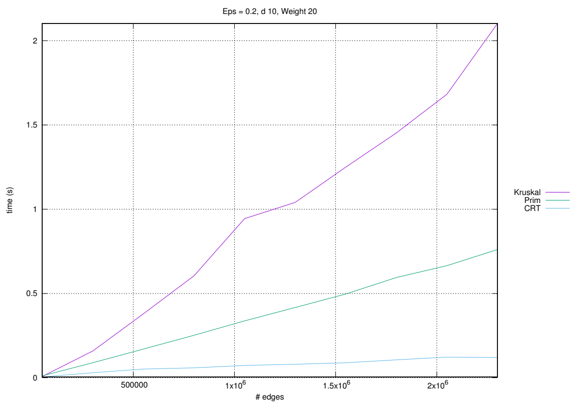

Fig. 4.3: Small-World

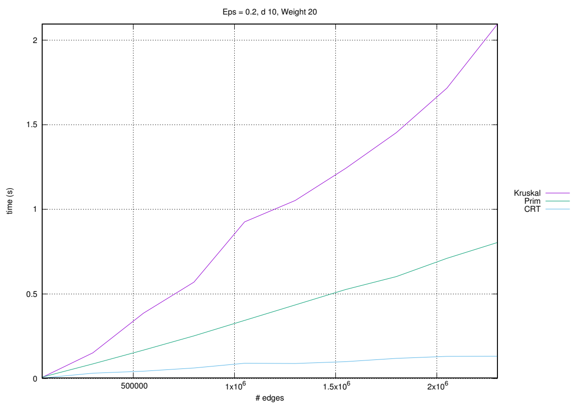

Fig. 4.4: Uniform

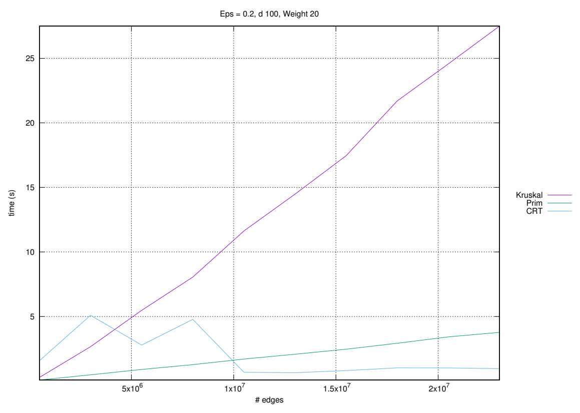

Fig. 4.5: Gaussian

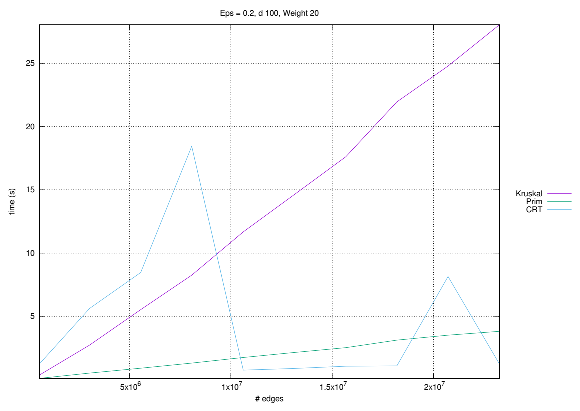

Fig. 4.6: Scale-Free

easily see that the performances become worse, as expected having increased the d value.

Let us now see what happens if we increase w, keeping d and ε unchanged.

As expected, the CRT running time is higher than before. To be precise, what happens is that for the Gaussian, for the Scale-Free and for the Uniform the trend is about the same (Fig. 4.7), while for the Small-World case the reader can observe (Fig. 4.8) that for large graph instances the trend becomes sublinear. We can imagine that the same happens for even larger instances also in the previous three kinds of graph.

To see if this is true we create an ad-hoc graphs dataset, that is an extension of our previously discussed dataset.
We generate an extension in the Uniform graphs dataset, adding larger instances (i.e. in terms of nodes and edges), and we want to see if the CRT algorithm becomes sub-linear, as expected, onwards from a certain point.

As the reader can see in Fig. 4.9, for larger instances the CRT algorithm really

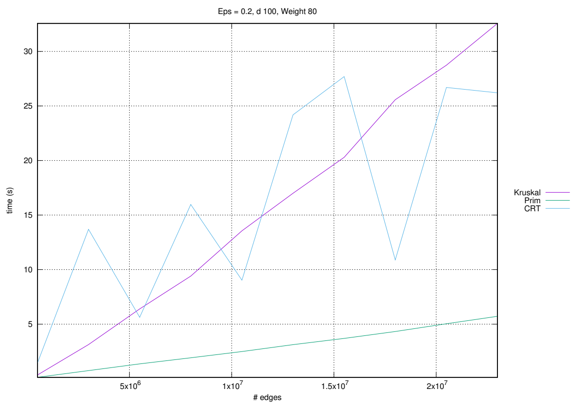

Fig. 4.7: Gaussian

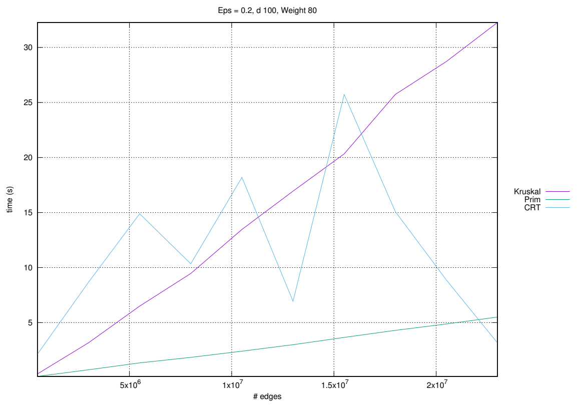

Fig. 4.8: Small-World

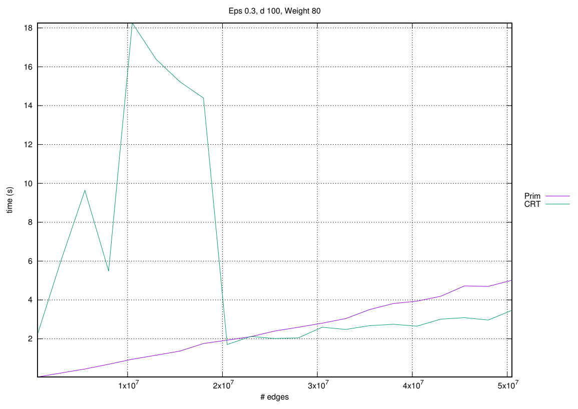

Fig. 4.9: Uniform

becomes sub-linear and its performances are better than, for example, Prim Algorithm.

This is in agreement with the definition of the O notation, that, for large arguments, indicates that the asymptotic behavior is of a certain type.

To conclude the discussion on the consistency of the execution time, we want now to see what happens in the case we increase ε, keeping d and w unchanged.
Since the ε parameter is a denominator for the estimated running time, what is expected is that an increase of this value corresponds to improved performances.

We can compare, for example, the performances of the CRT algorithm on two Small-World graphs with the following parameters:

- $d = 10$
- $w = 20$
- and compare what happens for $\varepsilon = 0.2$ and $\varepsilon = 0.4$

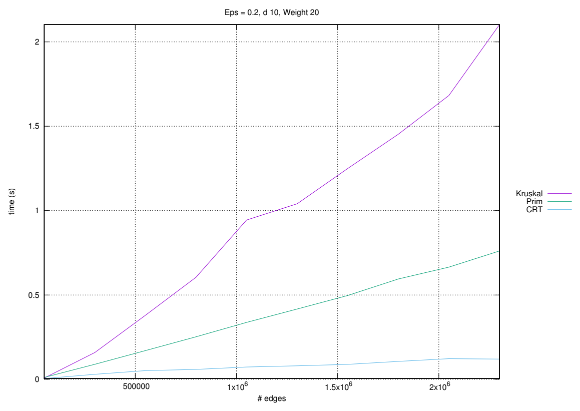

Fig. 4.10: Small-World, $\varepsilon = 0.2$

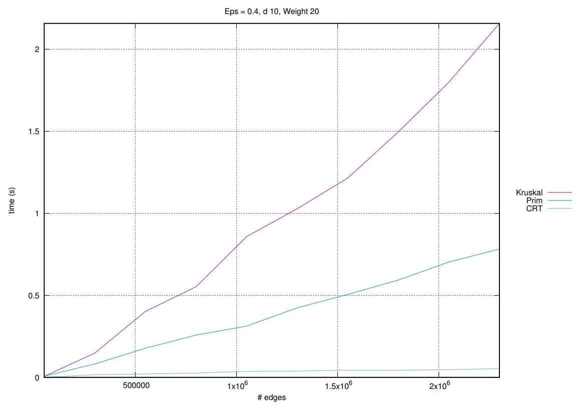

Fig. 4.11: Small-World, $\varepsilon = 0.4$

The reader can easily see that the CRT trend has decreased.

5. Final thoughts. As expected, a probabilistic algorithm like the CRT allows us to compute an approximation of the Minimum Spanning Tree in sublinear time on the number of edges, under certain conditions.
Tunable parameters, that depends on ε, allows us to perform either a better or a worse approximation, implying respectively a very slow and a very fast computation. The choice of a small value of ε can lead to terrible running times, and for these values it does not make sense to compare the CRT algorithm with any other deterministic algorithm.
For other ε values, instead, we prove the good performances of the CRT. The reader can easily view the better performances of CRT algorithm versus Prim algorithm or Kruskal algorithm watching the line charts in the previous section of this paper.

REFERENCES

[1] B. Chazelle, R. Rubinfeld, and L. Trevisan,Approximating the minimum spanning tree weight in sublinear time. SIAM J *Computing*, 34, 2005.