EVALUATING A SUBLINEAR-TIME ALGORITHM FOR THE MINIMUM SPANNING TREE WEIGHT 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 (see [1] for a theoretical exposure of the algorithm). Since the theoretical performances have already been shown and demonstrated in the above-mentioned paper of Chazelle et al. our goal is, exclusively, to experimental evaluate the algorithm and at last to present the results. Some technical insights are given on the implementation of the algorithm and on the dataset used in the test phase, hence to show how the experiment has been led out even for reproducibility purposes; the results 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 and different theoretical models for them.

We assume hereafter that the reader has knowledge about the cited paper as we will just recap 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 weights 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. 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 maximum 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 connectivity requirement. Given a desired number of nodes n and $e \geq n - 1$ 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 $v_0, v_1, \ldots v_n$
- 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 $\{T_0, \ldots, T_\tau\}$ 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 an induced subgraph of G such that the maximum weight on his edges is i (e.g. G_w is exactly G)

[3] `Boost Graph Libray`, [2]

Unfortunately, our Kruskal algorithm, although 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 because 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 assumptions; 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 determines 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 the original paper's 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[6]

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

hence we choose

$$r = \left\lfloor \frac{\lfloor \sqrt{\frac{n}{w}}\varepsilon - 1 \rfloor}{\varepsilon^2} \right\rfloor \in O\left(\frac{1}{\varepsilon^2}\right)$$

$$C = \left\lfloor \frac{\lfloor \sqrt{n}\varepsilon - 1 \rfloor}{\varepsilon} \right\rfloor \in O\left(\frac{1}{\varepsilon}\right)$$

[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]

[6] see Theorem 7 and Theorem 8 of [1]

2.4. Random Sequences. Another problem we encountered was the random selection of k distinct vertices out of n with $k < n$; after some research, we found that this kind of problem cannot be solved in sublinear time on the number of vertices, which can't be done for the same reasons exposed in the introduction about the subgraph family. The problem here is that we can't extract *with no reinsertion* k values without using an external data structure; this structure has a linear cost for the maintainance that depends on n. A special case is represented if n is a prime that fulfills certain properties, in that case we can extract a value in constant time without repetition; this solution is not affordable here because we need a dynamic range for each call.

The solution we found is to use Fisher-Yates sequences[7] which permits us to prepare in advance the sequences and then get different, distinct values at each call in constant time and with dynamic bounds. The cost of the preparation is linear and is not considered in the total cost.

3. Implementation choices. We choose to implement the algorithm in C++ because we wanted a language that offers the advantages of an OOP language, not least a faster development, that had a standard library and sufficiently "near" the machine to have a tighter control over the performances and the memory occupation (e.g. performances high variance due to overhead of a VM or an interpreter). We considered that absolute performances are not important here, whereas relative performances of the CRT algorithm and other MST algorithms are the focus, so finally the choice of the language is something not of the greatest importance. The IDE tool used for the implementation is JetBrains Clion. We also used GitHub, as our code versioning control system.

Also, as already mentioned, we did use the Boost library to extend the STL that C++ already offers; we did implement FastGraph in place of BGL also to address memory issues in relation to the method Boost uses to store subgraphs of a graph; we in fact used an advanced method to store the family of subgraphs $G_i, i = 0, 1, \ldots, w$ that simply store the difference of vertices between them, $\Delta_{G_i} := V(G_i) - V(G_{i-1})$. It is always possibile to reconstruct every G_i because this family only has *induced* subgraphs, and this cost is not taken into account in the computation of the total time.

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. It takes some argument in input, namely the file containing the graph, a suitable value for ε and the path where to save the results of the single run.

Besides, it is possible to run the random graph generator by the `grandom` utility we developed apart. It takes many arguments in input, that you can find just by calling

```
$> grandom -h|--help
```

3.1. Random Graphs Models. In order to give exhaustive results, we designed `grandom` to produce 4 classes of random graphs:

- Erdős-Rényi model, that builds random graphs constructed with a uniform distribution fo the edges and an average degree $d \approx \frac{2m}{n}$;

[7] originally decribed in [3] and then redesigned for computer use in [4]

- Gaussian model, that builds random graphs with a "cluster zone" where the edges are more frequent, hence the gaussian shape of the degree distribution; we have still $d \approx \frac{2m}{n}$;
- Barabási-Albert model, that builds random *scale-free* graphs. The average degree is $d \approx \frac{m}{n}$;
- Watts-Strogatz model, that builds random graphs with *small-world* properties and $d \approx \frac{2m}{n}$.

4. Tests. The following section reports the results we had launching the algorithm over a variegate dataset of graphs, as described in the next paragraph.

4.1. Dataset. For each random graph model listed in the previous section we decided to produce a dataset; each dataset has a "family" of graphs that differs one from each other following a pattern for the parameters. Precisely, this is how the dataset is composed following the rules:

- build one graph for each model (Uniform, Gaussian, Small-World, Scale-Free):
- for each selected value of n, i.e. the number of nodes (5000, 30000, 55000, 80000, 105000, 130000, 155000, 180000, 205000, 230000):
- for each selected value of d, i.e. the average degree $(20, 100, 200)$:
- since we have, for our models and for the fact that $m \propto n$, a proportion between the previous quantity and the number of edges m, i.e. $d \xrightarrow{n \to +\infty} \frac{2m}{n}$, we have the selected values for m $(10, 50, 100$ times $n)$[8]:
- then for each selected value of w, i.e. the weight $(20, 40, 60, 80)$.

Finally we have the number of models $\times$ the number of selected values for $n \times$ the number of selected values for d (hence m) $\times$ the number of selected values for w for a total of $4 \cdot 10 \cdot 3 \cdot 4 = 480$ graphs. This dataset, made of plain text files as already described, is quite heavy: 42.1 GiB of data.

4.2. Runs. Using the dataset just described, we used a pattern for the runs, in order to have a complete view of the behavior of the algorithm in the domain of the various parameters:

- build three plots for each model:
- for each selected value of ε $(0.2, 0.3, 0.4, 0.49999)$:
- for each selected value of $d/2 \simeq \frac{m}{n}$ $(10, 50, 100)$:
- for each selected value of w.

The first two plots report the absolute and relative error of the CRT result compared to the correct one, calculated with Prim's algorithm; the third report the trend of the used time. As we mentioned, we indeed used Kruskal too but it's not reported here for it was not faster than Prim for the times and unuseful for the results.

We had this way a total of $3 \cdot 4 \cdot 3 \cdot 4 = 144$ plots, or better 48 different cases of study. As mentioned before, the parameters should be selected carefully in order to provide a good approximation very fast. In this sense ε plays a crucial role here: since the CRT algorithm is a probabilistic algorithm and ε is indeed the driver parameter either for performances and for 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). So, although it could be of interest to study the right limit of $1/2$, we empirically noted that values of ε below 0.2 shows undesired behaviour of the CRT algorithm, for the results and for the running times either.

[8] operatively we fixed the number of edges, so d is a random variable

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 decided to launch 10 runs for each graph of the dataset, and to take the average running time as the estimate of the running time. For the approximated MST weight we instead toke one of them randomly, to preserve the information on the tolerance of the estimate.

5. Results. Following there are some considerations and some line charts regarding the main results.

We know that the CRT time complexity is $O(dw\varepsilon^{-2}\log\frac{dw}{\varepsilon})$, where d is the average degree of a graph; indeed we know that the accuracy depends on ε also, so we expect an inversely proportional relation with the running time. Therefore what we expect is that:

- increase one or more of the parameters d, w, there should be a worsening of the average running time;
- keeping all the other parameters unchanged, if we consider an increase in the value of ε there must be an improvement of the running time (as well as a worsening of the result, though);
- viceversa we expect the opposite behaviour if we decrease d and/or w or decrease ε with the other parameters unchanged.

Let us call the above *crucial parameters*.

First thing to show is that the CRT algorithm is sublinear in the number of edges, hence is better than any other *exact* algorithm. This can be easily seen in figures from 5.1a to 5.1c. For the rest of the plots listed from now on, it is possibile to see the values of the parameters for the presented run above the graph.

Given that the CRT Algorithm respects the sub-linearity constraint, let's see the variations that occur when selectively changing other parameters.

5.1. Variations of crucial parameters.

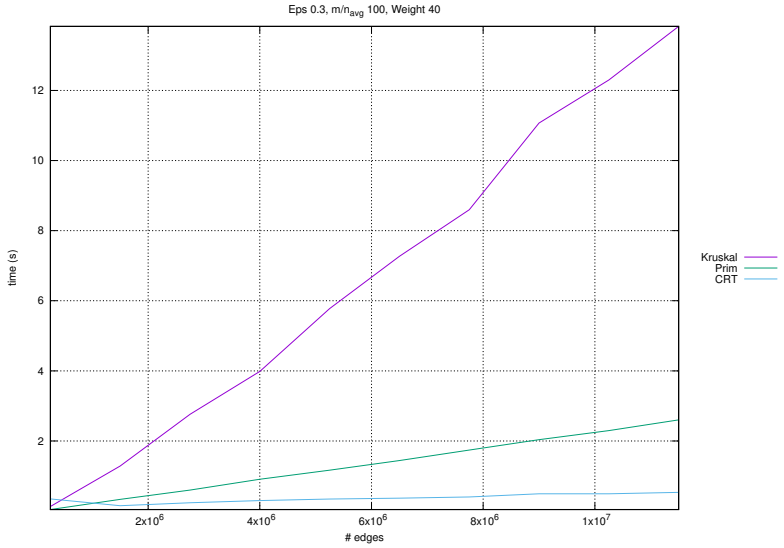
5.1.1. Average degree. Let us now see the behaviour for the variation of d; we will initially concentrate on the running time and solely for the uniform model. The selected values of ε and w are respectivey 0.3 and 40.

As we can see in figures 5.2a to 5.2c, there is a worsening in the performance *for small instances*: an increase of the average degree d is somewhat correlated to a loss of performances to the point that our property testing algorithm needs more time that the deterministic one; still, that seems to be true *under* a certain dimension of the instance of the graph, so that we don't lose the truthfulness of the theoretical time complexity because for a fixed d^* it will always be possibile to find empirically a certain number of edges $m^* \propto d^*$ beyond which the running time function is always below $C \cdot dw\varepsilon^{-2}\log\frac{dw}{\varepsilon}$ for a certain C.[9]

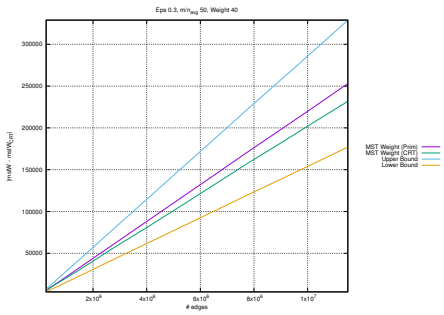
We would speculate that the error could be related to this. Indeed in figure 5.3 we can see

5.2. Variations of graph model.

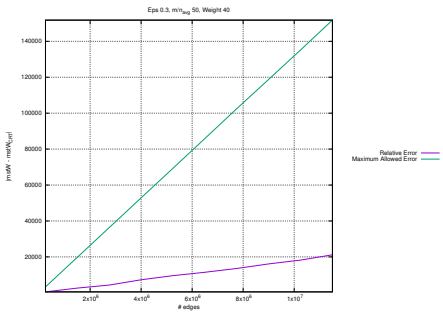
[9]that comes directly from the definition of *asymptotic complexity*



(a) *Running time.*

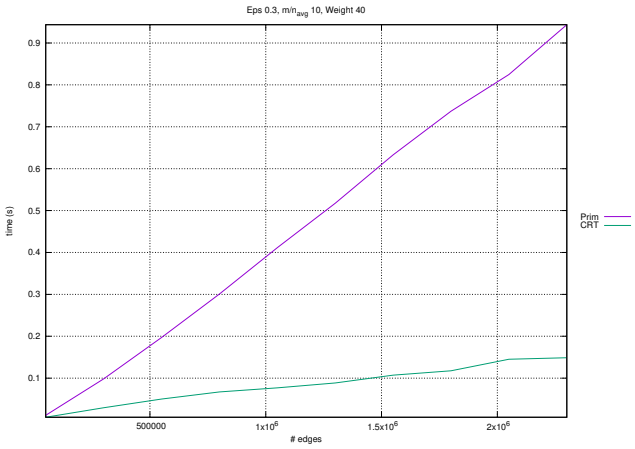


(b) *Relative error.*

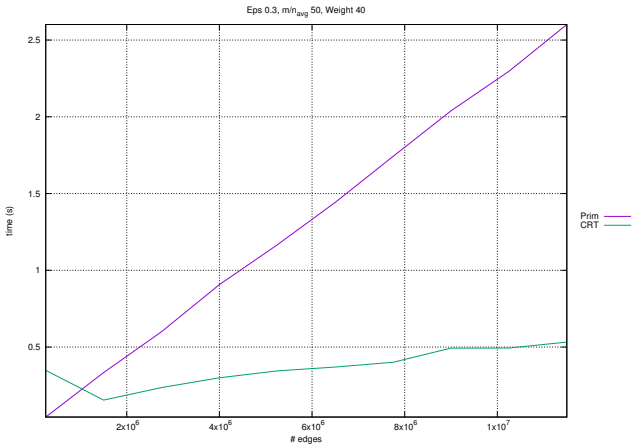


(c) *Absolute error.*

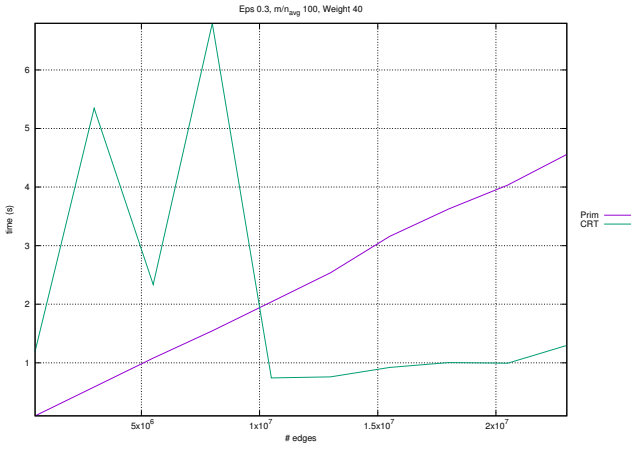
Fig. 5.1: Sublinearity of the CRT algorithm.



(a) $d \simeq 20$

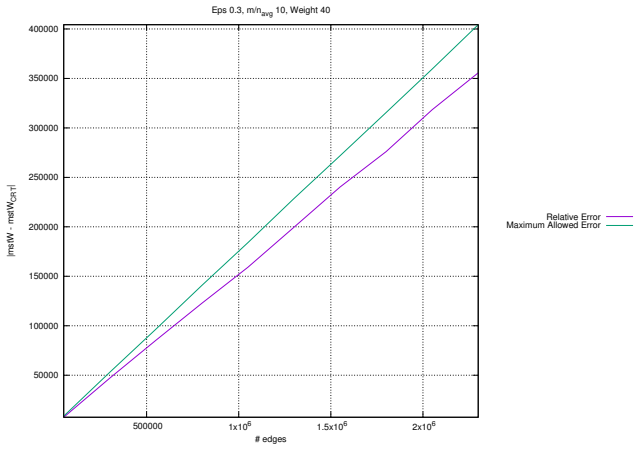


(b) $d \simeq 100$

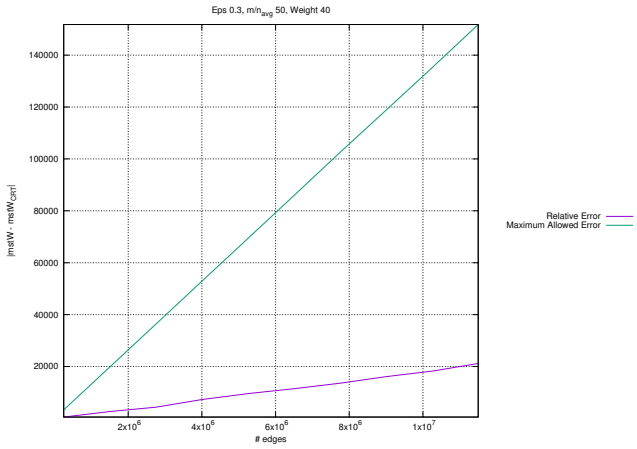


(c) $d \simeq 200$

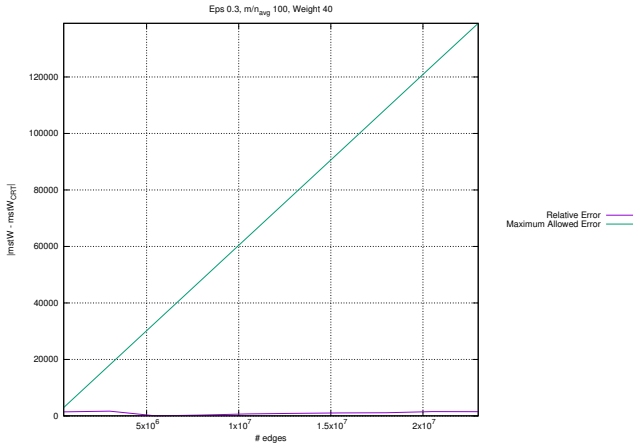
Fig. 5.2: Behaviour for the increase of d.



(a) $d \simeq 20$

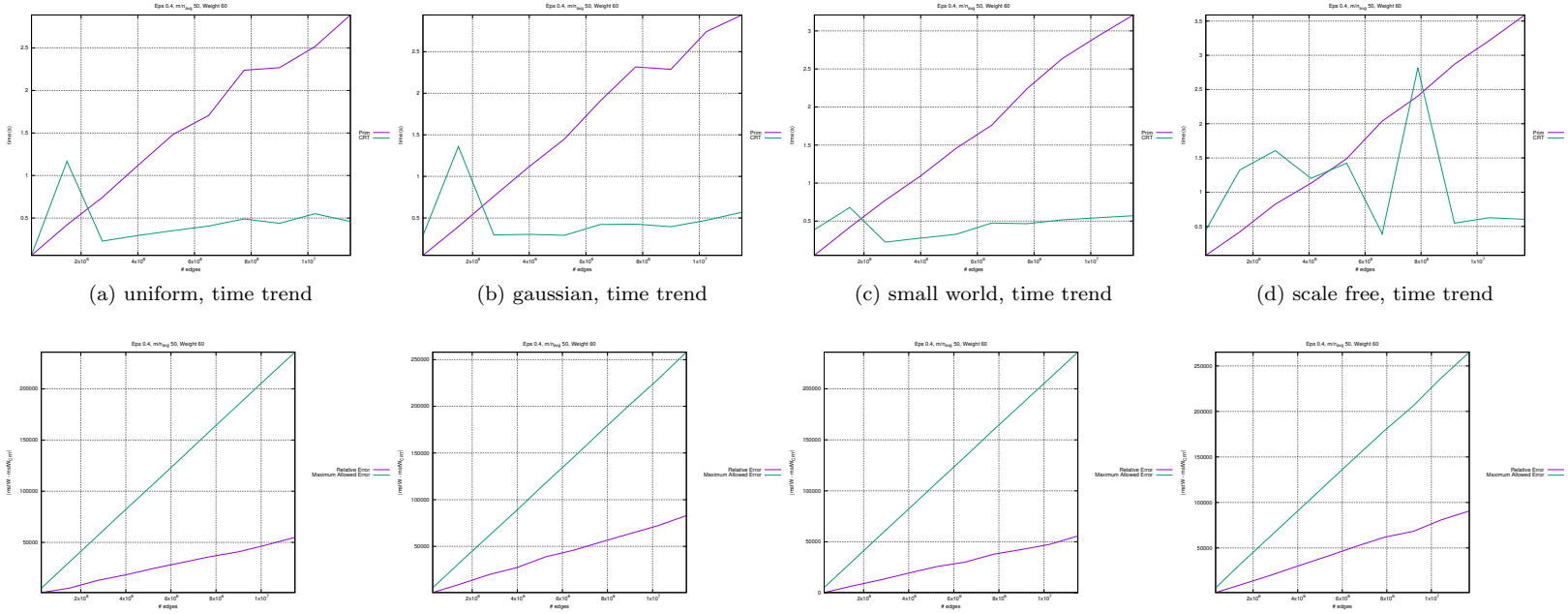


(b) $d \simeq 100$



(c) $d \simeq 200$

Fig. 5.3: Behaviour for the increase of d.



(a) uniform, time trend (b) gaussian, time trend (c) small world, time trend (d) scale free, time trend

(e) uniform, relative error (f) gaussian, relative error (g) small world, relative error (h) scale free, relative error

Fig. 5.4: Different behaviours for different graph models; fixed values are $\varepsilon = 0.4$, $\bar{d} = 50$, $w_{\max} = 60$.

While in the corresponding Small-World graph and the corresponding Uniform graph the trend is similar to the Gaussian graph (Fig. **??**), 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. **??**). 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. **??**) with the corresponding (Fig. **??**), one can 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. **??**), while for the Small-World case the reader can observe (Fig. **??**) 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. **??**, for larger instances the CRT algorithm really 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 $\varepsilon = 0.4$

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

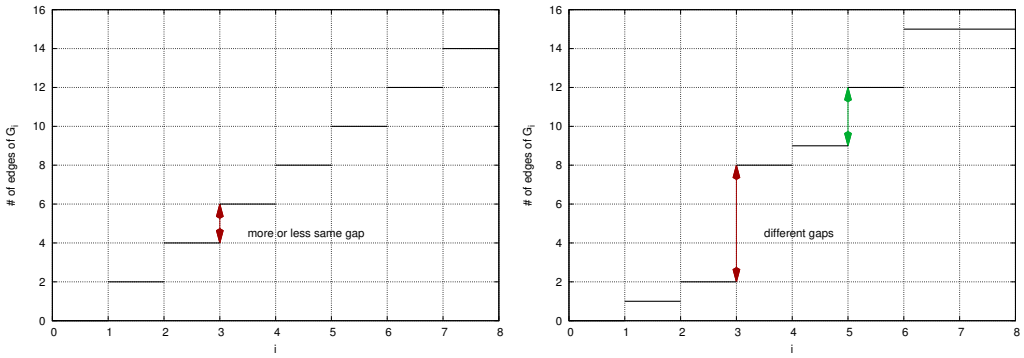
6. Final thoughts. As expected, a probabilistic algorithm like the CRT allows us to compute an approximation of the Minimum Spanning Tree weight in sublinear time on the number of edges, under certain conditions. Tunable parameters, that depends on ε, allows us to perform either a better or a worse approximation, 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.

6.1. Parallel implementation. We observe that the CRT algorithm lends itself very well to a parallel implementation. Indeed the majority of the algorithm's code is organized into independent sections, and in most cases they don't need to comunicate to each other. We also observe that three levels of paralellism can be achieved within the code. In the first level we parallelize each of the w independent calls to `approx-number-connected-components`; every call internally performs r independent BFSes from r different roots, that could in turn run in parallel, achieving a second level of parallelism. Moreover, considering that in the academic world there already exist different parallel implementations of the BFS algorithm, we can use one of them to perform an additional third level of paralellism.

To make it even more simpler, the number of different flows is known a priori, so a static pre-instantiation and a smart scheduling of the threads can be performed. At the first and second levels a *master-slave* model can be used in a *fork-join* structure, while in the third level a shared variable is needed between the different BFSes. As a final remark, parallelizing the BFSes could have too much overhead given that the algorithm is optimized to run them very fast and to stop the ones that seem to cost too much.

7. Future prospects. During an e-mail exchange with one of the original authors of [1], Dr. Ronitt Rubinfeld, another topic of discussion and study has emerged, about the distribution of the weight on the edges.



(a) Assuming a uniform distribution (b) Assuming a generic distribution

Fig. 7.1: Expected behaviour with different laws for the distribution of weights.

In our code, the generation of random graphs only assume a uniform distribution of the weights on the edges, i.e. a given edge can have an integer weight $k \in [1, w]$ with probability $\frac{1}{w}$. That implies a linear growth of the dimension of $E(G_i), \forall i \in [1, w]$, namely the set of G_i's edges; this is well depicted in figure [**?**], where, as i grows, the size of $E(G_i)$ increases at each step of a quantity "near" $\frac{|E(G)|}{w}$, and it is more true as $|E(G)|$ is big for the law of large numbers. On the other side, having a generic law of distribution for the edges weight implies having a different behavior as depicted on [**?**].

This difference could be of interest because it means that the input of different subsequent iterations of `approx_num_connected_comps` will have a non regularly increasing size, or even the same size for some calls; it can be easily shown indeed that the function in [**?**] is nondecreasing. Since the cost of those calls finally determines the overall cost, we might argue that this could lead to a minor difference, but at the same time think that only with another set of tests we could conclude something relevant about this observation.

REFERENCES

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

[2] Beman Dawes, David Abrahams, Rene Rivera, Boost C++ Libraries. http://www.boost.org/.

[3] Fisher, Ronald A.; Yates, Frank (1948) [1938], Statistical tables for biological, agricultural and medical research (3rd ed.). London: Oliver & Boyd. *pp. 2627.*

[4] Durstenfeld, R. (July 1964), "Algorithm 235: Random permutation". *Communications of the ACM.*