Dijkstra's was a heapy

Garrenlus de Souza
Informatics Institute
Federal University Of Rio Grande do Sul - UFRGS
Porto Alegre, Brazil
gsouza@inf.ufrgs.br

I. Introduction

A. Dijkstra's Algorithm

As pointed out in class, the twist that makes the Dijkstra algorithm so interesting is how it capitalizes on the selection of the next node to be explored (i.e.: the next node to have its neighbours added to the frontier). This selection is done by optimizing the distance from a given starting node to the one known to be the closest and ends up providing interesting insights about what can be expected depending on the current goal and employed data structures. The algorithm, depicted as follows relies heavily in three operations.

Algorithm 1 Dijkstra

Require: a graph $G = (V, E)$ weighted by c_e for the edges $e \in E$ and a vertex $s \in V$.
Ensure: d_v is minimal from s to any other vertex $v \in V$.

```
0: d_s ← 0; d_v ← ∞; ∀v ∈ V
0: visited(v) = false, ∀v ∈ V
0: Q = ∅
0: insert(Q, (s, 0))
0: while Q ≠ ∅ do
0:    v ← deletemin(S)
0:    visited(v) = true
0:    for u ∈ N+(v) do
0:       if ¬visited(v) then
0:          if d_u = ∞ then
0:             d_u ← d_v + d_vu
0:             insert(Q, (u, d_u))
0:          else if d + v + d_vu < d_u then
0:             d_u ← d_v + d_vu
0:             update(Q, (u, d_u))
0:          end if
0:       end if
0:    end for
0: end while=0
```

These operations are $insert$, $deletemin$ and $update$. The first inserts a tuple into Q, that tuple references a node u and a distance d_u. The next, $deletemin$ extracts from Q the node with the minimum distance to s that is still in Q. The next and last, $update$, does as suggested and updates the distance associated with u in Q to the value d_u.

These operations were implemented under the invariants and constraints of a min-heap. That way insertions must take at most $log_k(n)$ for k being the arity of each node, and n the amount of nodes in the tree at the moment of insertion. The $update$ operation must respect the same constraint as well. The $deletemin$ operation on the other side must be performed at $O(1)$, constant time.

II. Methodology

For the current study, the applied methodology consisted primarily into the analysis of running time and key data-structure related statistics made over a reference implementation of the $Dijkstra$ algorithm written in C++.

A. Min-k-Heap

This reference implementation was based in [1] and goes further on providing ways to more efficiently "heapify" a given array of elements from some type T using $Floyd's$ algorithm[] which is bound to linear time on the number of nodes, more efficient than the canonical approach of inserting the n elements into the structure, that ends up pushing the time complexity numbers against the $nlog(n)$ upper bound in the worst-case. Unfortunately, this extra functionality was not needed in the experiments, but for sure rendered itself as an important contribution for future use.

B. The Statistics

The main goal was to evaluate how $|V|$, $|E|$ and k affected the time needed to traverse a connected component completely, as well as the number of interchanges and updates employed. There was some effort on making sure that (u, v) paths were actually being correctly evaluated and optimized, but there was no further analysis over the eventual time numbers drawn from such experiments. The source code is available at [2] under a private repository.

C. The setup

In order to properly evaluate the aforementioned statistics some directed graphs were generated. This was done by defining the amount of vertices and edges only to then derive the independent probability for each of the $2 * \binom{n}{2}$ edges to be active in the resulting graph (Notice we are taking into account edges uv and vu). In order to provide a more comprehensive view of how each one of these characteristics affected running times for the traversing operation, the following values for i and j were used. For simplicity the amount of edges and vertices were defined over powers of 2, so the numbers below are in logarithmic scale. The amount of edges m is given by

TABLE I
GRAPH GENERATION PARAMETERS (I)

i	j	ρ
41	14	0.0055246089233887625
42	14	0.00781297686626381
43	14	0.011049217846777525
44	14	0.01562595373252762
45	14	0.02209843569355505
46	14	0.03125190746505524
47	14	0.0441968713871101
48	14	0.06250381493011048
49	14	0.0883937427742202
50	14	0.12500762986022096
51	14	0.1767874855484404
52	14	0.2500152597204419
53	14	0.3535749710968808
54	14	0.5000305194408838
55	14	0.7071499421937616
56	14	1.0000610388817677

$2^{i/2}$ in the table and n is given by 2^j. The values for k were drawn from the $[2, 29]$ range and for each test, 30 randomly selected nodes were chosen as starting points (the s node in the $Dijkstra$ algorithm). Each traverse was then timed. It must be brought to the reader's notice that these numbers are astonishingly big, for reference 2^{44} is over 16 trillion, so there would be 2^{22} edges, something around 2 million edges.

TABLE II
GRAPH GENERATION PARAMETERS (II)

i	j	ρ
56	14	1.0000610388817677
56	15	0.250007629627369
56	16	0.06250095368886854
56	17	0.015625119210199052
56	18	0.003906264901218037
56	19	0.000976564362648702
56	20	0.0002441408578308657
56	21	0.00006103518535384433

The attentive reader may have noticed that in the first table the values for j did not actually vary, staying in 15 for all the scenarios. These values were used for the first batch of experiments. For the next batch the latter table was used instead so a complete graph could be investigated (The machine that hosted the experiments was bounded at 32GB in terms of memory resources, this explains why in Table I i goes only so far as to reach 58).

For each generated graph, 30 source nodes were randomly selected and each one was then fed into the Dijkstra's implementation along one each possible value for k. In turn for each of these executions, statistics such as the amount of interchanges, updates, insertions and removals as well as the time spent were recorded. The amount of individual cases somewhat surpasses what one could expect to be relevant for presentation when it comes to variety when it comes to the distribution of results. Therefore, only some specific case studies got the spotlight given how representative they are over the whole set of data.

III. RESULTS

The set of test graphs is partitioned between real-world routing graphs of United State's localities and the graphs randomly generated described in the previous session.

A. Routing graphs

The only case studies for the routing graphs are New York and the whole United States. The time durations depicted below are in microseconds.

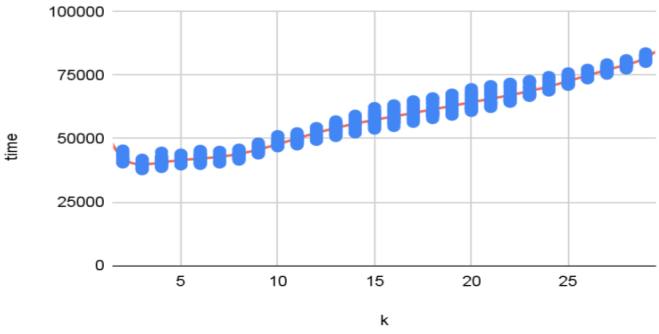

Fig. 1. The variations in time given for the New York instance using different values for k, the arity of the min-heap used in Dijkstra's algorithm.

The blue dots in the previous figure correspond to, in the case of a given k, to all the rooting nodes chosen at random for this specific graph. In this case the value that minimizes time is $k = 4$. Notice that as k grows, the time needed grows as well. One other important number in this context is the amount of interchanges in each graph traverse. This information is conveyed by the figure that follows.

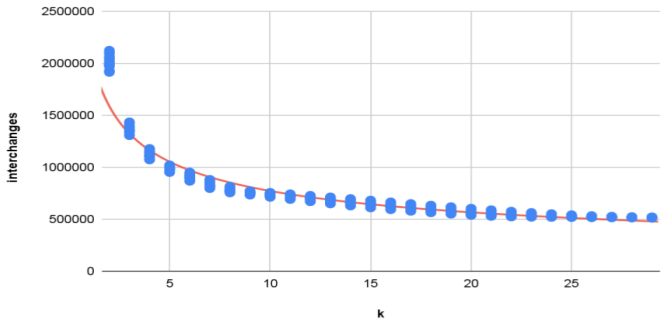

Fig. 2. The interchange numbers found for the New York instance using different values for k.

This interchanges swap elements inside the heap structure in order to keep things in check when it comes to the data-structure invariants. Notice how in both of the previous figures the data points are really close together. Also, notice how the number of interchanges decay resembling a steep logarithmic curve, just as expected, with the greater values in the end that corresponds to the smaller k.

For the whole country the overall behavior presents itself to be more pronounced when it comes to the variation in runtime as k grow large. It is worth highlighting that the reduction is

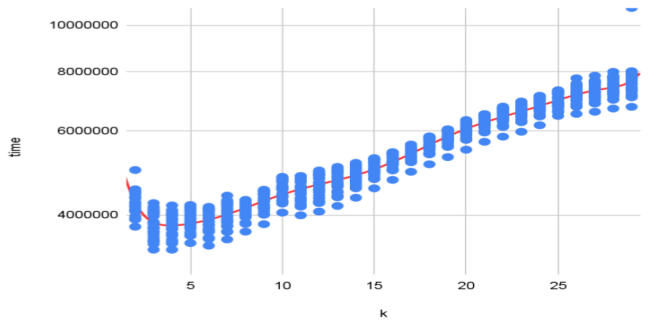

Fig. 3. The variations in time given for the whole USA instance using different values for k.

more expressive when it comes to the proximities of $k = 4$, the value that minimizes the runtime for the given case, something that happened at $k = 3$ in the case of New York. Just like the previous case study, the number of interchanges decrease seemingly logarithmically as function of n and k.

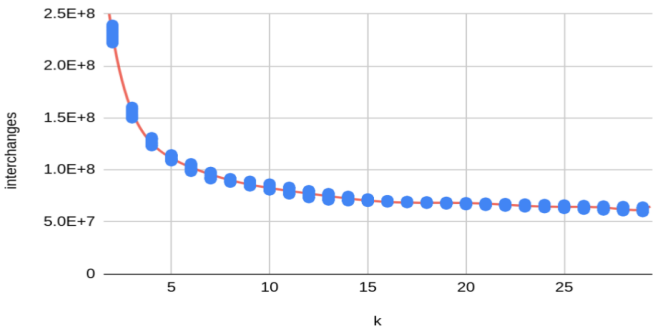

Fig. 4. The number of interchanges performed over the USA routing graph as k ranges from 2 to 29.

Notice how even though different values were used to start the graph traversing, the amount of interchanges necessary does not seem to be seriously affected as the dots fall into a really narrow range as can be seen in Fig. 4.

B. Random graphs

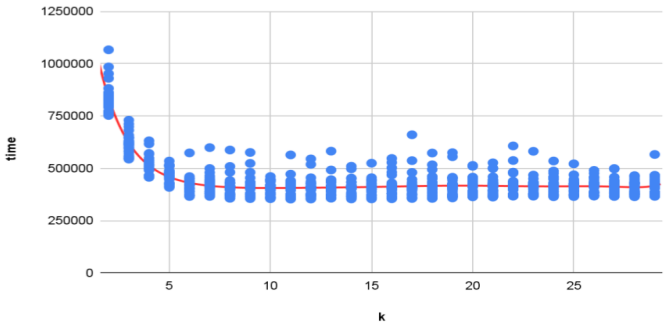

Fig. 5. Time as a function of k for the randomly generated complete graph with 2^{14} vertices.

For our next case study in Fig. 5, the complete graph with 2^{14} vertices, the time curve as a function of k deviates greatly from what was seen with the routing graphs, the time spent to traverse the graph drops steeply from $k = 2$ all the way to $k = 6$ and practically stabilizes itself around the same range until 29, slowly but surely increasing as k goes up.

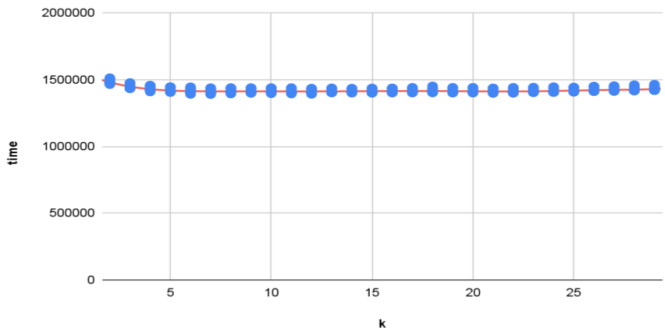

Fig. 6. Time as a function of k for a randomly generated graph with 2^{18} vertices and a $\rho \approx 0.0039$.

The next relevant case is the one if Fig. 6. In this experiment the variation of time between different starting nodes is astonishingly small, as well as the variation between different ks, notice how it drops (time) close to $k = 5$ and then steadily increases by a considerably small amount as k approaches 29.

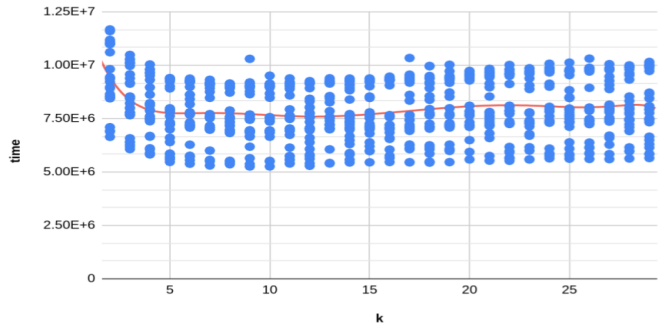

Fig. 7. Runtime as a function of k for the randomly generate graph with 2^{21} nodes and $\rho \approx 0.00006$.

In Fig. 7 we can see something quite interesting. This is the graph with the most vertices for all the ones generated (and further analyzed), with 2^{21} nodes (nodes and vertices are being used interchangeably). Notice that in this case the amplitude is way more significant than in the previously discussed scenarios. Actually, this graph is quite curious in the sense that the optimal k seems to be around $k = 9, 10$, and as you can see by the trending line (a polyfit with $d = 10$), it grows slowly as k gets larger.

Inf Fig.8 we have a comparison between the presented graphs in terms of average runtime, variance and coefficient of variation. Notice how stable is the coefficient of variance for the routing graphs that have respectively 22.26% and 23.49% for New York and the US as a whole. Something curious but

which was not further investigated, so it might as well be the result of sheer coincidence.

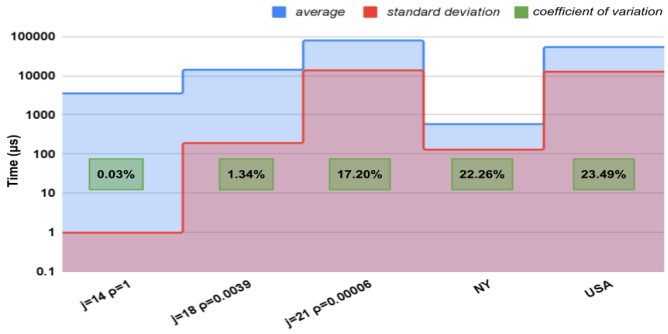

Fig. 8. Average, standard deviation and coefficient of variation for the routing graphs and some of the randomly generated ones.

IV. Discussion

One interesting aspect observed in the amount of updates performed for each simulation is that this value actually varied between simulations that kept the same graph and starting node, something that after some discussion with the professor came to light as a result of ties between nodes when it comes to sifting up or down through the heap (and the possibility of multiples exploration routes, which in turn introduces the aforementioned variation in the amount of updates performed).

This could be tackled by setting up a tie break in the process of finding out if some given node has children with (in the case of a min-heap) a lower key or if some node has a parent with a higher key. One possible criteria for such decision would be the node index. Such modification would not impact at all in the asymptotic time complexity but ended up being left out of the current implementation as to prioritize the completion of a deliverable version of the whole assignment.

V. Conclusion

There are a few interesting conclusions that can be drawn from the presented experiments. One of them is that the approached routing graphs and randomly generated graphs cannot be matched against each other as a result of how they were built. The routing graphs have notoriously lower node degree when compared to the others as the latter's was defined as an arbitrarily high number as to suit the demand for density probably exceeding what is actually reasonable for a routing graph.

Another interesting point that must be made relates to how each starting node could have its own optimal value for k, this was considered only at the later stages of this work, would it have happened otherwise, a more clever and widely distributed range could have been used, drawing potentially more interesting insights on the behavior of the algorithm in terms of interchanges and runtime for values of maybe hundreds or even thousands for k.

The growth in runtime as k increases could be very well attributed to the linearly increasing work needed to test for an interchange as a node needs to sweep along the key values of its children to decide if an interchange (being it a sift up or sift down) must be performed to restore the heap invariant.

One interesting detail is that up until a point while running the experiments, a curious behavior showed up with the first (and the first only) graph traverse using the Dijkstra's algorithm. It was desperately slow, taking something around 2 to 3 times the amount of every other nodes that would be (randomly) selected next. Regardless of the node. This was attributed to memory hierarchy related issues, something related to temporal or even spatial locality as it was the first time for the current graph in which its edges and vertices would be needed. The proposed workaround was to make one more evaluation at the end, discarding the numbers for the first one, this way reducing the noise in the final analysis, which is available at [3].

Another fact that might be relevant is that for some of the cases the machine running the evaluations was being concurrently used (in order to complete some of the other author's assignments) which might as well have affected the overall performance in some sense, though great distortions were not observed.

References

[1] Ernst E. Doberkat, "An average case analysis of floyd's algorithm to construct heaps," *Inf. Control*, vol. 61, no. 2, pp. 114–131, may 1984.

[2] "Dijkstra reference implementation," https://github.com/GarrenSouza/inf05016-dijkstra, Accessed: 2022-21-12.

[3] "Dijkstra experimental data," https://docs.google.com/spreadsheets/d/1SImaYMa9_7KOOMiVLokmVfdyopIzT1zAMWiraIKiekw/edit?usp=sharing, Accessed: 2022-21-12.