Advanced Algorithms 2019/2020

Advanced Algorithms project

Eleonora Auletta 94811 | Miguel Cóias 97137 | João Tavares 89630

---

1 Introduction

Network analysis is an important subject with varied applications such as social networks, road network planning or telecommunications. More often than not, these applications are associated with the need to analyze graphs for which even polynomial time algorithms are not adequate. Furthermore, the sheer size of these large graphs can also be a source of problems, if inadequate data structures are employed to store them in memory.

The main goal of the project is to attenuate the problems associated with the factors we listed – that is, to find and implement suitable data structures to store and work with our graphs in memory and algorithms that are designed to be efficient and scalable for the different metrics implemented. Thus, approximated algorithms will be used and compared to an exact algorithm that would solve the same problem, if it is feasible to use it at all.

2 Background

2.1 Definitions

We will define a graph to be the pair $G = (V, E)$ of sets such that $E \subseteq V^2$. Depending on whether or not we consider directed or undirected graphs, we may consider E to be a set of ordered pairs or not, respectively. We will also define $n = |V|$ and $m = |E|$ for simplicity. The types of graphs we will restrict ourselves to in this project with are either unweighted (all edges have weight equal to 1) or weighted, where every edge in E has a correspondent positive weight given by an injective function $w(e) : E \to \mathbb{R}$. Distinguishing between these two variants allows us to save memory for unweighted graphs, even though they are a special case of weighted graphs.

Although all the graph data structures implemented in this project are valid for general graphs, we will specifically focus on sparse graphs, where $|E| = \mathcal{O}(|V|)$. This allows us to use data structures that are both efficient in memory and in performance for this specific type of graphs.

2.2 Implementation

Our Python package implements both undirected and directed graph data structures with unweighted or weighted variants. Since our focus lands on static representations of sparse graphs, we chose to use the compressed sparse row (CSR) format to store the graph data.

Essentially, a user can create a graph by passing a JSON file path or an equivalent dictionary (using the default translations done by Python's `json` module) as the argument to the class constructor, which is then parsed by a function that organizes its contents into two arrays (or three, in the weighted variant). To understand how the structure works, an example of an undirected and unweighted graph is given below.

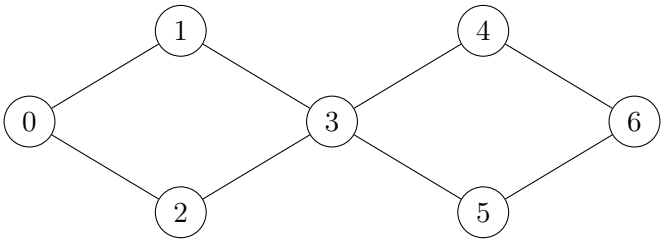

Figure 1: Example graph, with omitted weights.

This graph can be represented using the following adjacency list, in JSON format:

```
{
    "0": [1, 2],
    "1": [0, 3],
    "2": [0, 3],
    "3": [1, 2, 4, 5],
    "4": [3, 6],
    "5": [3, 6],
    "6": [4, 5]
}
```

It can be noticed that our parsing function does not support strings as labels, since this would require the use of an hash table, in addition to the arrays we mentioned.

Internally, this structure is rearranged into two arrays, `ind` and `adj`. In particular, `ind` stores the indices of `adj` where the neighbours of a vertex land. For instance, the neighbours of the vertex 0 in the example are `adj[ind[0]:ind[1]]`, considering that Python excludes the endpoint from array slices.

For weighted graphs, there is a third array `weights` that follows the same structure as `adj`, but instead of having the neighbours of a vertex it has, at each index, the weight of the edge that connects them. In this case, the JSON file should have the following format:

```
{
    "0": [[1, 2], [1, 1]],
    "1": [[0, 3], [1, 1]],
    "2": [[0, 3], [1, 1]],
    "3": [[1, 2, 4, 5], [1, 1, 1, 1]],
    "4": [[3, 6], [1, 1]],
    "5": [[3, 6], [1, 1]],
    "6": [[4, 5], [1, 1]]
}
```

and here the `WeightedGraph` and `WeightedDigraph` classes should be used instead.

The main differences between the undirected and directed graph classes consist in how the number of edges of the graph are computed, and how the iterator methods are designed. For undirected graphs, the number of edges m stored is half the size of the array `adj`, since in this case, edges are repeated by design, and iterating over them returns a list of edges (u, v) where $u \leq v$, in order to avoid listing repeated edges. When using directed graphs, the number of edges m is the size of `adj` and iteration over an instance of a directed graph returns the full list of ordered pairs (u, v).

The memory complexity of this representation is $\mathcal{O}(n + m)$, and it has the disadvantage of duplication in the case of undirected graphs, since an edge is internally represented as

two directed edges with equal weight, but reversed endpoints. Compression algorithms, or more compact data structures were not considered, since they could significantly impact the performance of the algorithms we implement, especially considering the limitations of Python and our limited knowledge about how to circumvent them.

One of the more important things to say is that parts of our project do **not** work with Windows, due to the way in which processes are created. At least, the algorithm implemented for betweenness centrality is confirmed to not work due to the subprocesses not being able to inherit a needed array from the parent process, which needs to be written to in order to avoid OOM issues from having an array for every subprocess. In Linux, processes are forked by default, which is what we want. Moreover, the fork method of creating processes can also be used in macOS machines. It's possible that there is a way to work around this on Windows systems, but we haven't been able to find it.

2.3 Functionality

Our `Graph` and `Digraph` classes implement the data structure described in the previous subsection, and provide the following basic API:

Method	Description
`neighbours(v)`	Returns array of neighbours of vertex `v`.
`order()`	Returns number of vertices.
`size()`	Returns number of edges.
`__iter__()`	Implements iterator functionality, i.e. `[e for e in G]`.

Both `order()` and `size()` methods have time complexity $\mathcal{O}(1)$. The method `neighbours(v)` has time complexity dependent on the outdegree of vertex `v`, since it is the length of the slice – if k is the outdegree of `v`, the method is $\mathcal{O}(k)$. The iterator method has time complexity $\mathcal{O}(m)$, since it iterates through every edge once and only once.

In order to run the project, one should open a Python terminal in the root folder of the project. For example, in Bash:

```
(aalg) user@computer:~$ cd aalg-graphs/
(aalg) user@computer:~/aalg-graphs$ python
Python 3.6.9 (7.3.1+dfsg-4, Apr 22 2020, 05:27:13)
[PyPy 7.3.1 with GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
   information.
>>>> from structs.graph import Graph
>>>> G = Graph('examples/unweighted.json')
>>>> [e for e in G]
[(0, 1), (1, 5), (1, 6), (1, 7), (2, 3), (2, 6), (3, 4), (4,
   5), (5, 6), (6, 7)]
>>>>
```

All the graph data structures are in `structs`, and all algorithms are in `algorithms`. Some graph examples are included in the folder `examples` and the benchmark/testing scripts are in `benchmarks`. The folder `hash` contains mostly hash functions used for the average path length part of the project.

3 Metrics

3.1 Betweenness centrality

The betweenness centrality of a vertex v of a given connected graph G is a measure of how frequently the vertex v lands on the shortest paths between other vertices of G. To formalize this intuitive concept, we first need to define a number of objects. We will define σ_{st} as the number of shortest paths between $s, t \in V(G)$, and $\sigma_{st}(v)$ as the number of shortest paths between $s, t \in V(G) \setminus \{v\}$ that pass through v.

In order to define the betwenness centrality, the authors of [1] further define S_{st} as the set of shortest paths p_{st} between s and t, and S as the union of S_{st}, for all $s \neq t$. Then, for any vertex v, let $T_v \subseteq S$ be the set of all shortest paths that v is internal to, which can be formally defined as $T_v = \{p \in S : v \in \text{int}(p)\}$. In this framework, the authors then define the normalized betwenness centrality as

$$b(v) = \frac{1}{n(n-1)} \sum_{p_{st} \in S} \frac{\mathbb{1}_{T_v}(p_{st})}{\sigma_{st}} \tag{1}$$

where $\mathbb{1}_{T_v}(p_{st})$ is the indicator function of T_v. In order for this definition to also work for undirected graphs we can say that $p_{st} \neq p_{ts}$, otherwise we would need to multiply $b(v)$ by 2.

Currently, the state-of-the-art exact algorithm to find the betwenness centrality of a vertex is Brandes' algorithm, which finds $b(v)$ in $\mathcal{O}(nm)$ time for unweighted graphs and in $\mathcal{O}(nm+n^2 \log n)$ for weighted graphs, for which we also provide a parallelized implementation for unweighted graphs. Brandes' work centered on his definition of dependency, which he defined as

$$\delta_{s*}(v) = \sum_{t \in V(G)} \delta_{st}(v) \tag{2}$$

where $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. The main contribution of his seminal paper was the recursive formula he found for dependency:

$$\delta_{s*}(v) = \sum_{w | v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s*}(w)) \tag{3}$$

where $P_s(v)$ is the set of predecessors of v in the shortest paths from s to v.

The high computational cost of this exact algorithm is unfeasible for large graphs, hence our work in betweenness centrality will be focused around an approximated algorithm, we will call `vcbc`.

3.1.1 Theory

The main results used are from Vapnik-Chervonenkis theory, which is a key component of statistical learning theory. We will state the main results without proofs which can be found, for the most part, in the paper we base ourselves on for this section. Essentially, the definitions, lemmas and theorems we will now state are used to obtain the number of samples necessary to obtain a (ε, δ) approximation of $b(v)$ for all $v \in V(G)$. In what follows, let S be a domain and R a collection of subsets of S. The online version of the paper contains a nice example in figure 2 that illustrates the two following definitions well.

Definition 1. *Let $B \subseteq S$. The projection of R on B is the set $P_R(B) = \{B \cap A \mid A \in R\}$. If $P_R(B) = 2^B$, that is, $P_R(B)$ is the power set of B, we say that B is shattered by R.*

Definition 2. *The Vapnik-Chervonenkis dimension of R, $\mathrm{VC}(R)$, is the cardinality of the largest subset of S shattered by R.*

Now, let $X = (X_1, ..., X_k)$ be a collection of independent and identically distributed random variables on S, sampled according to a distribution ϕ defined on S, and let $\phi(A)$ be the probability that a sample from ϕ belongs to a set $A \subseteq S$. We will use the empirical average $\phi_X(A) = \frac{1}{k}\sum_j \mathbb{1}_A(X_j)$ of $\phi(A)$ as an unbiased estimator for $\phi(A)$.

The following definition and theorem will be essential to the algorithm.

Definition 3. *Let R be a collection of subsets of S and ϕ be a probability distribution on S. For $\varepsilon \in (0, 1)$, an ε-approximation to (R, ϕ) is a multiset C of elements of S such that $\sup_{A \in R} |\phi(A) - \phi_C(A)| \leq \varepsilon$.*

Theorem 1. *Let R be a collection of subsets of D with $\mathrm{VC}(R) \leq d$, and let ϕ be a distribution on D. Given $\varepsilon, \delta \in (0, 1)$, let C be a collection of $|C|$ points from D sampled according to ϕ, with $|C| = \frac{c}{\varepsilon^2}(d - \ln \delta)$, where c is an universal positive constant. Then C is an ε-approximation to (R, ϕ) with probability at least $1 - \delta$.*

This theorem will be used to find the required sample size in order to find an approximation that fits our (ε, δ) criteria. In the following section we will discuss the universal positive constant c and possibilities for an upper bound of d. To put into context how the previous definitions and theorems are useful in computing an approximation of the betweenness centrality of all vertices of a graph, let $R = \{T_v \mid v \in V(G)\}$ and notice that R is a collection of subsets of S as defined on the previous section.

3.1.2 Implementation

The following lemma is essential to find the sample size we need to obtain the desired precision for the betweenness centrality of all vertices. The vertex diameter, $\mathrm{VD}(G)$, is the size of the longest shortest path on S.

Lemma 1. $\mathrm{VC}(R) \leq \lfloor \log_2(\mathrm{VD}(G) - 2) \rfloor + 1$.

When the graph is unweighted, $\mathrm{VD}(G)$ is equal to $\mathrm{diam}(G) + 1 = \max_{u,v \in S} d(u, v) + 1$. This is used to obtain a quick approximation, since we can use a simple algorithm that ensures we have an upper bound, called `diam2approx`. This algorithm does breadth-first search on an unweighted graph from a random source vertex and sums the largest two distances found. It's easy to see why the approximation d_G lands in the range $\mathrm{diam}(G) \leq d_G \leq 2\,\mathrm{diam}(G)$, which is good enough for our purposes while, at the same time, being relatively fast to obtain.

In the case the graph is weighted, finding a good upper bound for $\mathrm{VD}(G)$ is a difficult problem – the size of the largest weakly connected component of G is an obvious, but very loose upper bound. In what follows, we will focus on unweighted graphs, for which the problem is more tractable. Then, the sample size r we will use is

$$r = \frac{c}{\varepsilon^2}\left(\lfloor \log_2(d_G - 2) \rfloor + 1 - \log \delta\right) \tag{4}$$

The universal constant c is estimated to be around $\frac{1}{2}$, as per the authors (which refer to [2]). This fixed value is what we will use in practice. Furthermore, the pseudocode in the next page explains the general process done by our algorithm.

Algorithm 1 VCBC with G = (V, E), with ε, $\delta \in \mathbb{R}$ (small)

```
function VCBC(G, ε, δ)
    bc ← zeros(|V|)
    diam ← diam2approx(G)
    r ← ⌈1/(2ε²) (⌊log₂ (diam − 2)⌋ + 1 − log δ)⌉
    for k = 1 to r do
        u, v ← randomVertex(G)
        dist, parents, sigma ← BFS(G, u, v)
        t ← v
        while t ≠ u do
            prob ← [σ_uk / σ_ut for k in preds[t]]
            z ← randomVertex(preds[t], weight=prob)
            if z ≠ u then bc[v] = bc[v] + 1/r
    return bc
```

3.2 Average path length

The average path length of a given unweighted graph G is a measure of the average number of steps along the shortest paths for all possible pairs of nodes. To define this all we need to use the concept of distance between to vertices. We define $d(v, u)$ to be the length of the shortest path between v and u and the average path length as

$$l(v) = \frac{1}{n(n-1)} \sum_{u \neq v} d(u, v) \tag{5}$$

Using the convention that $d(u, v) = \infty$ when there is no path from u to v, this concept is only useful when the graph G is connected.

There are a few algorithms to find the average path length based on computing the shortest path length for every pair. This approach scales badly, especially if the graph is dense.

In order to find this measure more efficiently we use the HyperBall algorithm which uses HyperLogLog counters in order to approximate the path lengths. This algorithm has a worst case runtime of $\mathcal{O}(DV^2)$ where D is the diameter of the graph.

A HyperLogLog counter is a vector M of size $m = 2^b$ used to approximate the number of distinct elements n in a stream. Given a counter we estimate n as:

$$E = \frac{\alpha_m m^2}{\sum_{k=0}^{m-1} 2^{-M[k]}}$$

The HyperBall algorithm works by reframing a graph as a stream into each vertex. So each vertex will have it's HyperLogLogCounter used to estimate the number of elements in $N(v, d) := \{u | d(v, u) \leq d\}$

3.2.1 Theory

The theoretical results needed are all from the HyperLogLog counters. We will state all the main results without proof. These are presented in detail in the original paper.

The core idea is that by hashing a value the number of zeros from left to right in it's binary representation is random with distribution $P(Z = k) = \frac{1}{2^k}$. Therefore by storing the maximum number of zeros found z we can get a rough estimate of the number of distinct

elements by calculating 2^z. In order to obtain a better approximation we split the stream in $m = 2^b$ substreams, indexed by the first b bits of the hashed value, and calculate the harmonic mean times a empirically found constant. The maximum number of zeros will be stored in a HyperLogLog counter which is an array with m elements.

Theorem 2. *The estimate E obtained by HyperLogLog using $m \geq 3$ is asymptotically almost unbiased in the sense that*

$$\frac{1}{n}\mathbb{E}_n(E) = 1 + \delta_1(n) + o(1)$$

where $|\delta_1(n)| < 5 \cdot 10^{-4}$ as soon as $m \geq 16$.

3.2.2 Algorithm

To work with these HyperLogLog counters we need a few auxiliary methods. Add which given an element from the stream will add it to the counter. Size, which returns the current estimate for the number of distinct elements given a coutner M. Union, which given two counters will merge them together taking the maximum for each element and therefore estimating the number of distinct elements that have been added to both of the counters.

We define $h_b(x)$ as the first b bits in $h(x)$, and $h^b(x)$ as remaining set of bits. As previously stated each element of the counter corresponds to a substream indexed by the first b bits of the hashed value, that is by $h_b(x)$. We also need to define another function, $\rho^+(x)$ which corresponds to the number of leading zeros plus one. That is, $\rho^+(010001) = 2$ and $\rho^+(100001) = 1$ for example.

Algorithm 2 HyperLogLog with h: $\mathbb{D} \to 2^\infty$ and α_p given according to our c

```
1: function ADD(M: counter, x: item)
2:     i ← h_b(x)
3:     M[i] ← max(M[i], ρ+(h^b(x))
4: function SIZE(M: counter)
5:     Z ← Σ_{j=0}^{p-1} 2^{-M[j]}
6:     return α_p p^2 / Z
7: function UNION(M: counter, N: counter)
8:     for i < p do
9:         M[i] ← max(M[i], N[i])
```

The HyperBall algorithm simply uses $|V|$ HyperLogLog counters each of which we will use to estimate $N(v, d)$ for $v \in V$ and $d \geq 0$. Each counter will be initialized with the hashed index of it's corresponding vertex, so that initially it is the estimate for $N(v, 0)$

In order to approximate $N(v, d+1)$, we will iterate over the neighbours of v and union, in an auxiliary counter a, the counter $M[v]$ containing the counters of its neighbours. At the end of this process we estimate $N(v, d+1) - N(v, d)$ as delta $=$ size(a) $-$ size(d) and update the estimate for the sum of distances by adding $(d+1) \times$ delta to it.

When the sum of all the deltas for every $v \in V$ is 0 we know that the counters have not been updated and therefore will never be because we are only doing unions not adding new items. Once we reach this state we can halt execution and return our estimate for the average path length. We know for a fact that in at most V cycles this will happen, however we can also show that there will be at most $D = \text{diam}(G)$ cycles. The diameter is defined as the maximum eccentricity which implies that for any given $v \in V$, $N(v, D) = V$, so after D steps every counter will be equal terminating execution.

Inside this loop there are two nested loops which go over at most $|V|$ elements. Because every operation done inside this loop is $\mathcal{O}(1)$ we have that the worst-case time complexity of HyperBall is $\mathcal{O}(DV^2)$.

Below we have the pseudo-code for HyperBall as is presented in the original paper with the required steps to calculate the average path length filled in.

Algorithm 3 HyperBall with $G = (V, E)$ and $h : V \to 2^{32}$

```
1:  function HYPERBALL(G, h)
2:      n ← |V|
3:      for v ∈ V do
4:          add(c[v], h(v))
5:      sum ← 0
6:      for r = 0 to n do
7:          delta ← 0
8:          for v ∈ V do
9:              a ← c[v]
10:             for u ∈ N(v) do
11:                 a ← union(c[u], a)
12:             c'[v] ← a
13:             delta ← delta + size(c'[v]) - size(c[v])
14:         if delta = 0 then
15:             break
16:         sum ← sum + r · delta
17:         for v ∈ V do
18:             c[v] ← c'[v]
19:     return sum / (n(n−1))
```

3.2.3 Implementation

Following the original implementation of HyperLogLog counters we used $b = 5$ and 32 bit hash functions. This proved to be both enough to calculate the metric with a reasonable amount of precision as well as efficient enough by keeping the constant low.

Working with graphs of up to 10^7 vertices we had to keep memory usage as efficient as possible. Especially if we want to parallelize the algorithm as we did, as it is obvious that each process would need it's own set of counters. Besides the memory needed to store the graph, we must also store the values of the counters themselves. Hyperball requires us to store the counters corresponding to distance d in order to calculate those for distance $d + 1$ so we only needed to have at any given time $2V$ counters in memory.

For this application we cannot have only V counters because we cannot update them during the cycle. We could allocate a new set of counters and at the end of the cycle assign them to other the V counters. In order to not have to be constantly be allocating more memory we chose to have $2V$ counters $M[2][V]$, and use the set of counters corresponding to the parity of d, i.e. $M[d \mod 2]$. This allowed us to minimize the memory usage as well as save some time in comparison with the pseudo-code which at the end of a cycle will assign the new values into the old counters.

We can now estimate our memory usage as $m \cdot 2V \cdot s$ where m is the number of elements in a counter, V the number of vertices and s the number of bits occupied by a single element.

Because we are using 32 bit hash functions and because we chose to use the array library for python, which is known to be memory efficient, we can estimate s to be at most 32 bits.

The implementation of the hyperloglog functions as fairly straight forward as mostly equal to the pseudocode. The function `size` that iterates over the 32 elements and performs $O(1)$ operations so `size` is $O(32) \sim O(1)$. The function `add` consists only of $O(1)$ operations being $O(1)$ as well. The `union` function iterates over the 32 elements of the counter performing a max operation, which is $O(1)$, being $O(32) \sim O(1)$ as well. The function zero also has a loop with at most 28 cycles so it's runtime is $O(28) \sim O(1)$

Now let's decompose the code we have for hyperball: The allocation of counters is linear so it should take $O(2V)$. The loop for initializing the counter only performs add operations which we have determined to be $O(1)$ so the loop is $O(V)$. As for the main loop as we have seen before is shoul be worst case $O(DV^2)$ as long as all the operations inside are $O(1)$ which as we have seen they are.

We also implemented a parallel version of this which uses the `multiprocessign` library to run several instances of this hyperball algorithm we have just implemented. This parallel function will consume large amounts of memory because, due to our lack of experience with these techniques, we were not able to implement a version which shares the graph in memory.

4 Experimental evaluation

4.1 Betweenness centrality

Our implementation `vcbc` uses multiprocessing in order to obtain a large boost in performance. Given our rudimentary knowledge about parallelization, the performance of this algorithm may be suboptimal in practice, compared to other implementations done by more experienced programmers using other low-level programming languages.

One key thing to note is that our algorithm is not fully parallel, since a shared array `bc` is used to mitigate the memory constraints from keeping the results of all processes in memory. Since this involves using a lock to avoid race conditions, it is slower than a fully parallel version, but requires much less memory to run – furthermore, this will **not** work on Windows, since we cannot fork a process in that platform. We observed that for larger graphs and smaller values of ε the effects of this mitigation aren't as notable, but we suspect that this system was switching to swap memory with the fully parallel version.

For this section, we chose three graphs from KONECT with varying sizes: the US power grid (4941 vertices and 6594 edges), Amazon (MDS) (334863 vertices and 925872 edges) and Hyves (1402673 vertices and 2777419 edges). Furthermore, we also generated a handful of random connected undirected graphs ranging from 100 to 50000 vertices and around 1000 to 550000 edges. Since we could not find enough connected graphs on KONECT with 10^4 to 10^5 vertices, we used two of the random graphs. The first has 11120 vertices and 106378 edges and the second has 49056 vertices and 416086 edges (we'll call them 11120_106378_5 and 49056_416086_6 for "convenience").

All tests are run on a laptop with a quad-core Intel Core i5-9300H CPU and 8 GB of RAM, with PyPy 7.3.1 (based on Python 3.6.9) and using Debian Bullseye. Furthermore, the running times presented are averaged over three runs.

By running the module `benchmark.vcbc_test`, the obtained results should be somewhat reproducible on a similar system. The results are organized in the table below and plotted with a logarithmic ordinate axis in figure 2. We did not run the algorithm for Hyves with $\varepsilon = 0.02$ because we expected that three runs would take almost three hours, given the

running time for $\varepsilon = 0.04$.

Graph	$\varepsilon = 0.1$	$\varepsilon = 0.08$	$\varepsilon = 0.06$	$\varepsilon = 0.04$	$\varepsilon = 0.02$
US power grid	0.53757	0.83267	1.24306	2.12777	7.02754
11120_106378_5	1.59019	2.30290	4.21617	8.90452	32.68625
49056_416086_6	9.24373	12.81338	22.03346	47.06658	188.07697
Amazon (MDS)	37.80916	59.78469	105.69812	228.188820	990.20482
Hyves	168.52913	231.90643	386.84655	864.34706	–

Table 1: Running time, in seconds, of `vcbc` for various values of ε

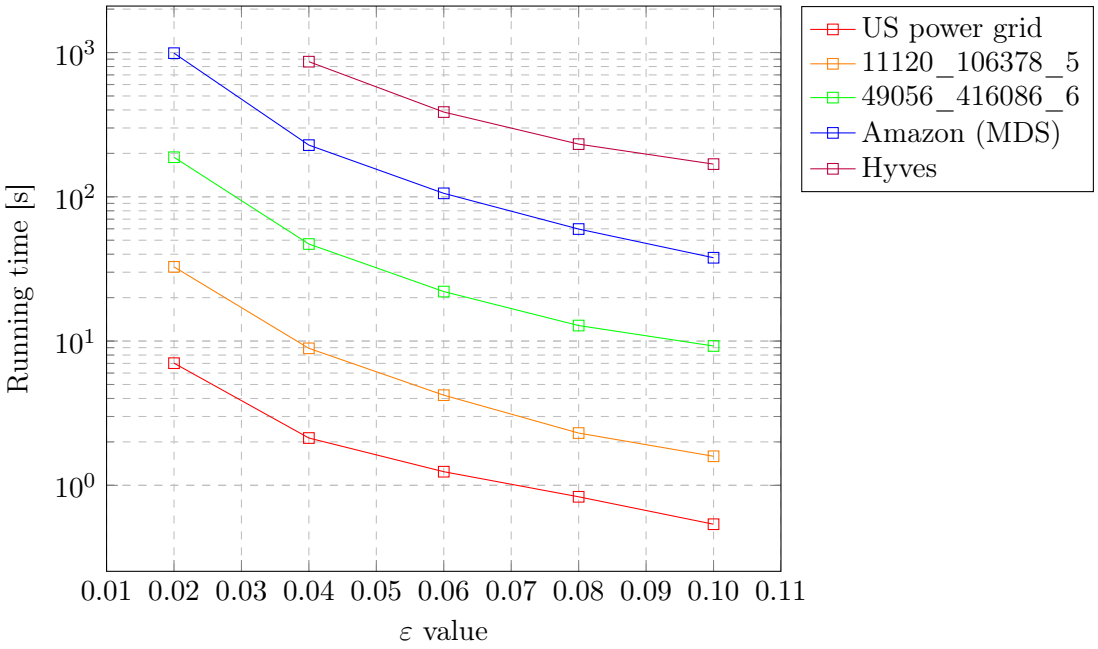

Figure 2: Running time of `vcbc` with fixed $\delta = 0.1$

The running times obtained are consistent with what we expected; for a single iteration, the algorithm calls breadth-first search on the graph with two random endpoints, which has time complexity $\mathcal{O}(n+m)$. Given how r scales with ε and the diameter of the graph, these running times seem to be what would be expected, even though our method of testing does not use the same diameter approximation for different values of ε, generating a new one every time instead. We did not change this since it reflects how the algorithm performs in practice.

Although the version of breadth-first search used here supports obtaining multiple shortest paths, this does not significantly impact the worst-case performance of the algorithm, since we only compute and store the array of predecessors of each vertex of the graph. That is, we never explicitly compute the paths, only the predecessors of each vertex, thereby avoiding a possible exponential time problem.

The remainder of the algorithm has been explained in the previous section, which is a loop over the shortest path between two randomly selected vertices. The number of iterations of that loop can be as high as the maximum possible length of the shortest path between two vertices in a graph, which is n. All operations used in that loop can be done on constant or amortized constant time (in the case of appending to a list); therefore, the asymptotic worst-case complexity of one iteration of the algorithm is $\mathcal{O}(n+m)$.

We can then conclude from this analysis and previous knowledge about the value of r that the asymptotic running time of our `vcbc` algorithm is $\mathcal{O}(r(n+m))$. One key takeaway from this expression is that $T(G, \frac{\varepsilon}{2}, \delta) \approx 4T(G, \varepsilon, \delta)$, with $T(\cdot)$ being the running time of the algorithm. This is exactly how we estimated that running `vcbc` for Hyves with $\varepsilon = 0.02$ would take around three hours, knowing the running time for $\varepsilon = 0.04$.

In order to confirm our theoretical observation for the asymptotic running time, we gathered a set of 99 random connected undirected graphs with diameter 6 and measured the execution time for each of them, using fixed values of ε and δ. The results are plotted in figure 3 and, as expected, the execution time appears to vary linearly with respect to $n + m$. The black line included is the linear regression line computed using the pairs in the plot, and takes the form $T(n+m) = 4.92056 \cdot 10^{-5}(n+m) + 0.07825$.

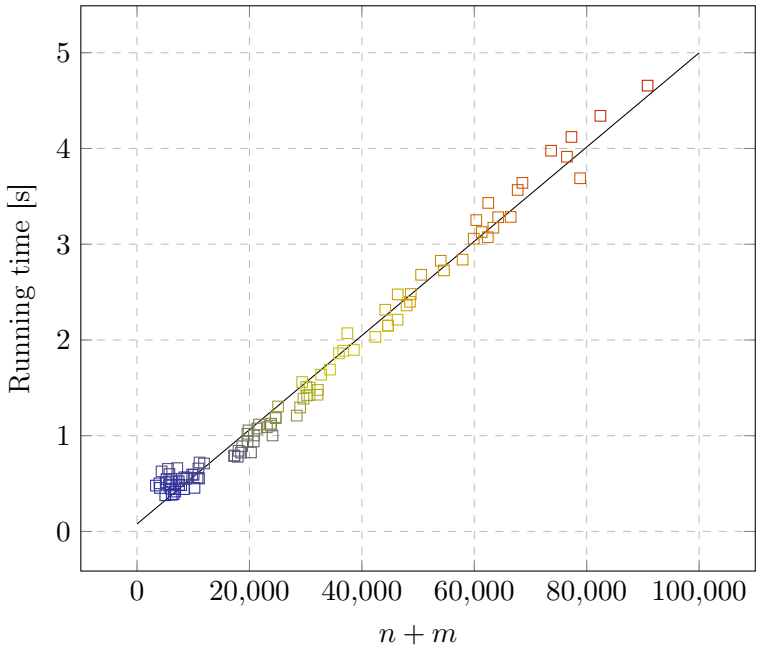

Figure 3: Running time of `vcbc` with fixed $\varepsilon = 0.05$ and $\delta = 0.1$, for random connected undirected graphs with diameter 6

4.2 Average path length

Due the nature of the algorithm used we need to perform several approximations with different hash functions to average out and get closer to the real average path length. So we used multiprocessing to dramatically decrease the running time. In order to avoid race conditions, deadlocks and whatnot we parallelized each instance of `hyperball`. We could obtain better performance by paralellizing the union operation for the hyperloglog counters but due to the scheme we chose to represent the counters it became very dificult to implement.

All tests were run on a Intel Core i7-7500U CPU and 8 GB of RAM, with PyPy 7.3.1 and using Manjaro. We experimented with linear hash functions, Jenkins 96 bit mixing functions, `randint` from the python standard library, the operating system's `urandom` and also linear hash functions with coprime coefficents. All of them ran in approximately the same amount of time, the slowest being `urandom` due to the OS latency.

We performed several tests and measured the variance of the sample. The highest

variance was by far that of the families of linear hash functions. For all others the variance was not significantly different. We believed that the family of colinear functions would result in a better behaviour of the hash function and therefore a better estimate. The experimental results decisively show that not only doesn't it give better estimates, but the variance also seems to increase. In fact, it appears that this decision of sampling from this specific subset of linear functions has even created a bias on the estimate with its results consistently tending to the wrong value. These results are highlighted in the next table.

As for the graphs, we chose from KONECT 2 different graphs with various sizes and diameters: the US power grid and PGP, as well as 2 of the random graphs we created. The results are summarized in table 2.

Graph	`random`	`urandom`	`mix`	`linear`	`colinear`
US power grid	18.7565	18.9118	18.6355	21.7369	26.0076
	4.2570	4.2417	4.0193	21.1001	20.4937
PGP	7.5277	7.5397	7.4378	7.7817	11.0333
	1.6313	1.6607	1.5715	8.0171	22.8603
26988_314915_5	3.6017	3.5598	3.6132	3.6672	4.7758
	0.7207	0.7382	0.7275	3.7002	3.7813
37129_301192_6	4.0258	3.9951	4.0444	4.1198	5.6803
	0.7730	0.8272	0.7823	4.8652	4.9080
49056_416086_6	4.0792	4.0872	4.0988	3.9350	5.2056
	0.8549	0.8405	0.7661	4.2846	3.8832

Table 2: Average value and variance over 700 runs of `hyperball` for each graph

To analyze the worst case we used SageMath to generate graphs. Because we want to explore the upper bound $\mathcal{O}(DV^2)$, we generated several random graphs with different diameters. In order to this we created a script that using the SageMath function `randomGNP(n, p)` with n being the order of the wanted graph, and p the probability that an edge $e \in V^2$ exists. This script samples n from $[10, 10^4]$ and p from $[10^{-4}, 10^{-2}]$ and use `randomGNP(n, p)` to generate a graph. These bounds were chosen for two reasons: firstly, because computing the diameter from bigger orders was very expensive and we wanted a huge amount of graphs to analyze. Secondly, because we were only interested in connected graphs using a smaller p made it very hard for the random graph to be connected.

The running time of `hyperball` for each of these graphs resulted in the plot in figure 4. We chose to plot the running time as a function of V^2 because if the worst time was as predicted, we should see for each different diameter that it would be bounded above by a linear function of slope proportional to D. This seems to be indeed the case. However, we can also notice that as the diameter increases the less time it takes to run. This initially seems to contradict the upper bound we found, although all it seems to mean is that the upper bound of V for the number of neighbours of any given vertex is too rough. When the diameter increases we expect to have a sparser graph. Which, in turn, means a smaller number of adjacent vertices. This, however, cannot be translated into an explicit relation which could be used to tighten the worst case bound.

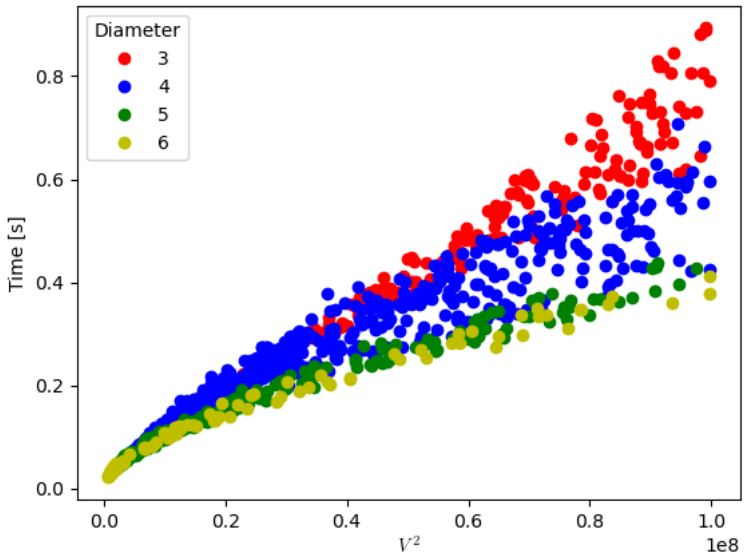

Figure 4: Running time of `hyperball`

5 Conclusion

Taking into account the introduction of our project, we believe that we have done a reasonable job in regards to using efficient data structures to store our graphs, even if we did not explore compression or more compact data strucutres. Furthermore, the algorithms we used also scale decently well with the increasing graph sizes, and the implementations used seem to be in line with the known theoretical bounds. We don't think that the performance of our algorithms could be substantially improved from our versions, while keeping the same graph data structures and using Python (or a variant like PyPy).

One of our main takeaways is that we will not be using Python again if the objective is to have high performance code – even if it's possible to do, it can often be harder and more unpredictable than doing it in a lower level language.

With this project, we experienced the main pitfalls of programming in Python and learned how to work around some of them. Moreover, our skill at reading scientific papers and interpreting their results also improved vastly from this experience, which will be a valuable skill going forward.

References

[1] Matteo Riondato and Evgenios M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, page 413–422, New York, NY, USA, 2014. Association for Computing Machinery.

[2] Maarten Löffler and Jeff M. Phillips. Shape fitting on point sets with probability distributions. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, pages 313–324, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[3] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA '07: Proceedings of the 2007 International Conference on Analysis of Algorithms*, 2007.

[4] Stefan Heule, Marc Nunkesser, and Alex Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the EDBT 2013 Conference*, Genoa, Italy, 2013.

[5] Philippe Flajolet and G. [Nigel Martin]. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182 – 209, 1985.

[6] Paolo Boldi and Sebastiano Vigna. In-core computation of geometric centralities with hyperball: A hundred billion nodes and beyond, 2013.

[7] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.0)*, 2020. `https://www.sagemath.org`.

[8] US power grid network dataset – KONECT, September 2016.

[9] Pretty Good Privacy network dataset – KONECT, April 2017.

[10] Amazon (MDS) network dataset – KONECT, September 2016.

[11] Hyves network dataset – KONECT, October 2016.