CS5234 Final Report

This report is an extension on an algorithm of the paper: *Chowdhury, Rezaul & Ramachandran, Vijaya. (2005). External-memory exact and approximate all-pairs shortest-paths in undirected graphs. 735-744. 10.1145/1070432.1070536.*

Our extension modifies the algorithm AP-BFS, a cache-oblivious algorithm presented in the paper to solve the all-pairs shortest path (APSP) problem. Our extension makes it more efficient by bounding the complexity of the iterations of a loop, in a step of the algorithm.

We also implemented in Java the following algorithms: AP-BFS, our extension of AP-BFS and a APSP solving algorithm relying only on MR-BFS and compared them on different graphs.

Table of Contents

The All-Pairs Shortest Path problem (APSP) 2
The AP-BFS algorithm 2
MR-BFS 2
Incremental-BFS 2
AP-BFS 4
Key ideas 4
Full algorithm 5
Our extension of the AP-BFS algorithm 5
Description of our extension of AP-BFS 5
Added Complexity of our method: 8
Correctness of our method: 8
Full algorithm of AP-BFS extension 8
Benchmark of our extension of AP-BFS versus AP-BFS 8
Benchmark on graph 1 (sparse) 10
Benchmark on graph 2 10
Benchmark on graph 3 (dense) 11
Conclusion of the benchmarks 11
Conclusion 11
References 12

The All-Pairs Shortest Path problem (APSP)

The All-Pairs Shortest Path Problem consist in finding the shortest distance between every pair of nodes of the graph. It can used to compute the diameter of the graph.

The algorithm AP-BFS is a cache-oblivious algorithm to solve the APSP problem for undirected, unweighted graphs.

Other algorithms to solve the APSP problem were presented on the paper. We decided to focus our extension on AP-BFS.

The AP-BFS algorithm

This article first provides a cache-oblivious algorithm, denoted **AP-BFS**, to solve the APSP problem for **unweighted, undirected graph**. The result of this algorithm can then be used to find the diameter of the graph.

The key idea of this algorithm is that it performs BFS in an I/O-efficient way for each vertex of the graph, by computing a BFS of a node (with a cache-oblivious algorithm) and then use the result to compute more efficiently the BFS of other nodes. To do so, the authors introduce the following algorithms:

MR-BFS

The algorithm **MR-BFS** [1], which is a BFS algorithm that works by iteratively constructing Layers $L(i)$(nodes at distance *i* of the source node *s*), with the relation:

$$\begin{cases} \mathrm{L}(0) = \mathrm{s} \\ L(i) = Neighbors\big(L(i-1)\big) - L(i-1) \cup L(i-2) \end{cases}$$

It has an I/O complexity of $O\big(V + sort(E)\big)$.

It is the algorithm in Week’s 10 lecture, so we did not delve into many details.

Incremental-BFS

The algorithm **Incremental-BFS**, which take as an input a node *u*, the BFS result of a node u, and a node *v* (the source node), and use it to compute more efficiently BFS(*v*).

Let’s note $L_u(i)$ the set of nodes at a distance *i* of *u*, and $A_u(i)$ the set of the adjacency lists of these nodes.

The key idea of **Incremental-BFS** is that for an unweighted undirected graph if you consider three vertices *u, v, w*, then:

$$(1) d(u,w) - d(u,v) \leq d(v,w) \leq d(u,w) + d(u,v)$$

So, if you want to compute $d(v,w) = i$, there exist a j such that the adjacency list of *w* is in $A_u(j)$.

The equation (1) allows us to determine the following inequality:

$$i - d(u,v) \leq j \leq i + d(u,v)$$

This inequality gives us a possibility to optimize the **MR-BFS** algorithm: By running it, for a layer $\mathrm{L(i)}$, instead of accessing the adjacency lists of all the neighbors of $L(i-1)$ to construct it, you can only consider the neighbors of nodes of $L(i-1)$ which have their adjacency list in $A(j)$, with $j \in$

$\left[max[0, i - d(u,v)]; min[|V| - 1; i - 1 + d(u,v)]\right]$. This allows us to reduce the number of nodes to consider at each iteration to compute L(i)

For example:

Let's imagine we computed $L_v[2]$, computed the BFS for u, and want to compute $L_v[3]$ for the following graph:

Since $d(u,v) = 1$, we have $j \in [1;4]$. So we will consider the lists of adjacency lists $A[2], A[3]$ and $A[4]$, and see if the adjacency list of the nodes in $L_v[2]$ belong to one of these lists. If so, we will add it to $L_v[3]$.

We start with $L_v[3] = \{\}$

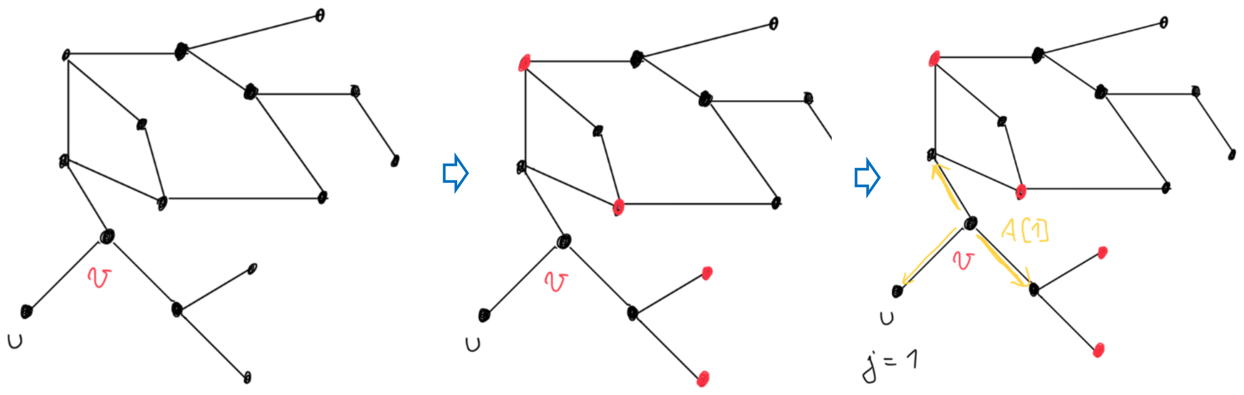


Figure 1 - Example Incremental-BFS (1/3)

We first consider the nodes in $L_v[2]$ (in red). Then, we consider the list of adjacency lists $A[2]$. No nodes of $L_v[2]$ have their adjacency lists in $A[2]$, so we don't add anything to $L_v[3]$.

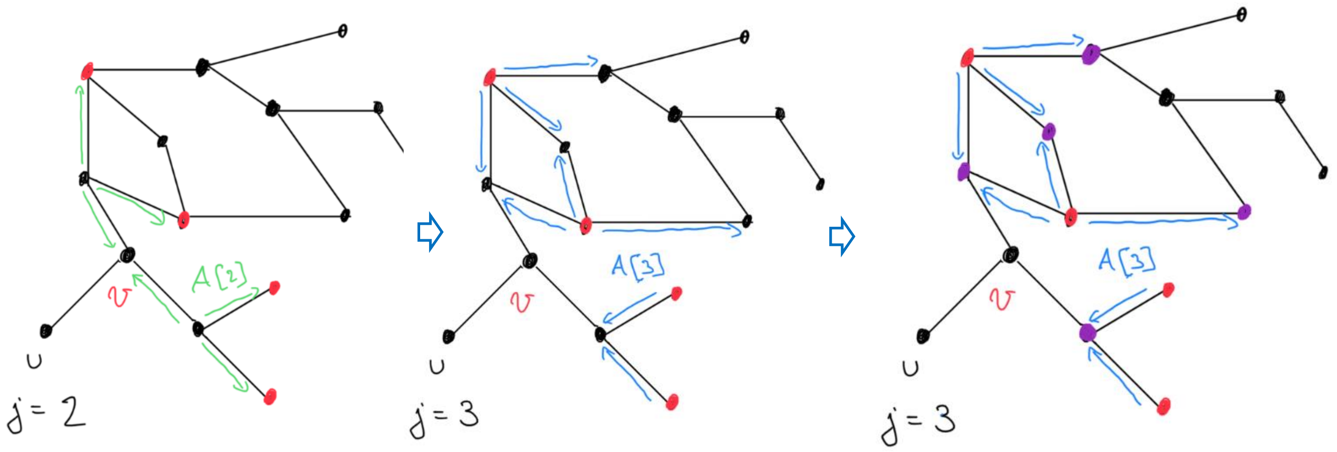


Figure 2 - Example Incremental-BFS (2/3)

Then, we do the same for $A[2]$. We can see that no nodes of $L_v[2]$ have their adjacency list in $A[2]$, so we don't add anything to $L_v[3]$.

We now consider $A[3]$. We can see that the four nodes in $L_v[2]$ have their adjacency list in $A[3]$. We can then add all the nodes of their adjacency lists to $L_v[3]$ (The nodes in purple).

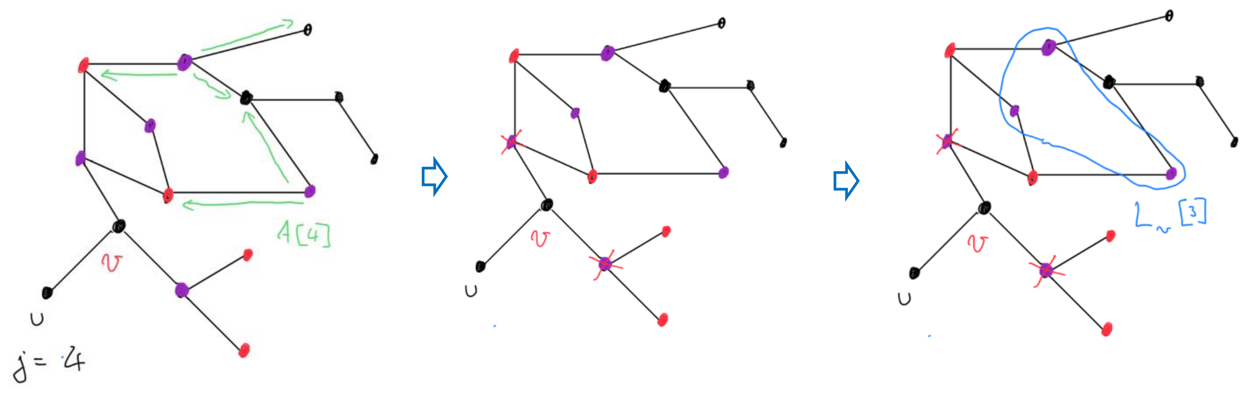


Figure 3 - Example Incremental-BFS (3/3)

We now do the same for $A[4]$, and no node is eligible.

The rest of the MR-BFS can now resume, by removing the duplicates in $L_v[3]$, and subtracting the nodes present in $L(2) \cup L(1)$. We end up with the nodes circled in blue for $L_v[3]$.

The **Incremental-BFS** algorithm has an I/O complexity of $O\left(\frac{E}{B}d(u,s) + sort(E)\right)$.

AP-BFS

Key ideas

As for the general algorithm **AP-BFS**, which is the one solving the AP-BFS problem by relying on the first two:

A key idea is that the algorithm will use **MR-BFS** to compute the BFS of a node. Then, it will iteratively construct the BFS of the other vertices, by using the previously determined vertices, with **Incremental-BFS**.

Something to consider is the order in which we compute the BFS of the vertices. The authors use for this the order inferred by the Euler Tour[1] of a spanning tree of the graph to determine this order. For an unweighted, undirected graph, each vertex appears at least once in a Euler Tour of a spanning tree of this graph, providing an order on the set of the vertices.

[1] A Euler Tour of a graph is a path that traverses each edge exactly once. In the context of a tree (for example for a Spanning Tree of a graph), it can be seen as a depth first traversal of the tree, beginning at the root, and finishing at the root. (Erik, P., & Lai., S. K. (s.d.). Lecture Notes in Advanced Data Structures. http://courses.csail.mit.edu/6.851/spring07/scribe/lec05.pdf)

For example, we consider the following tree, its Euler tour, and the ordering of the nodes from this Euler Tour:

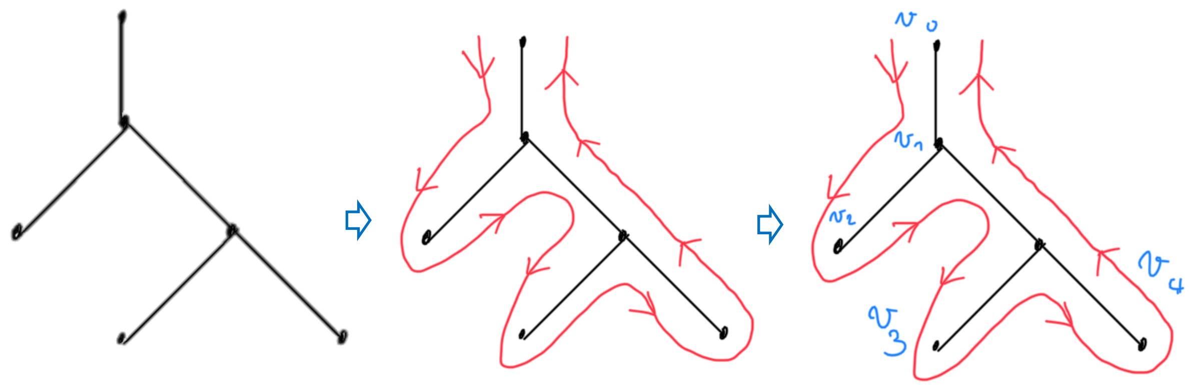


Figure 4 - Euler Tour of a Tree and the resulting ordering of the nodes

Full algorithm

The algorithm **AP-BFS**, does the following steps:

1. Finding a spanning tree T of the graph, construct its Euler Tour mark the first occurrence of each vertex in the Euler Tour.
2. Run **MR-BFS** on the first marked vertex "v_0".
3. For every other vertex in the order, they appear in the Euler tour, compute its BFS with **Incremental-BFS**, by using the result of the previous iteration's BFS (or the result of **MR-BFS** for the first iteration).

The overall complexity is $O\big(V \times sort(E)\big)$ I/Os and it needs $\Theta(V^2)$ space to store the all-pairs shortest paths.

It is possible to use these results to compute the diameter of a graph $O\big(V \times sort(E)\big)$ I/Os and $\Theta(V)$ space.

Other algorithms were introduced in the paper and explained in our first and second report and lecture, but we focused on this first algorithm for our extension. Therefore, they were omitted from this report.

Our extension of the AP-BFS algorithm

Description of our extension of AP-BFS

We decided to focus on the algorithm AP-BFS and see if we could optimize it, and we noticed the following:

The algorithm uses a Euler Tour of a spanning tree of the graph, to determine an order in the vertices.

It will then run a *MR-BFS* (K. Munagala and A. Ranade's Algorithm [1]) on the first node.

Then for every other node, in the order determined, the algorithm will run the algorithm *Incremental-BFS*, by relying on the result of the previous iteration's BFS to shorten the results.

Incremental-BFS is a variation of *MR-BFS* algorithm, with a complexity of $O\left(\frac{E}{B}d(u,v) + sort(V)\right)$, with v the node for which the BFS is computed.

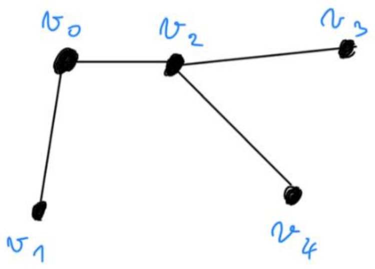
To optimize each iteration of *Incremental-BFS*, it is necessary, for each node v, to find a node u such that $d(u, v)$ is minimal.

Since we consider an unweighted, undirected graph, for every node v, there exists a node u such that $d(u, v) = 1$, resulting in always having the range of possible values of j to: $j \in [i - 2; i]$.

The only case this might not be true is that if a node has no edges, then the BFS problem for this node is trivial. So, we can assume a node has at least a neighbor.

Considering this, we found a simple way to construct a better iteration order than the order of the Euler Tour of a spanning tree of the graph, for the AP-BFS algorithm, that bounds the complexity of each iteration of the Incremental-BFS algorithm from $O\left(\frac{E}{B}d(u, v) + sort(V)\right)$ to $O\left(\frac{E}{B} + sort(V)\right)$.

For example, let's consider the following graph (we took a tree out of simplicity):



This would give us the following iterations in which we will run $IncrementalBFS(G, u, v, d(u,\cdot))$, to compute the BFS of v with the results of the BFS of u, represented by $d(u,\cdot)$.

Iteration	**u**	**v**	$\boldsymbol{d(u, v)}$
1	v_0	v_1	1
2	v_1	v_2	2
3	v_2	v_3	1
4	v_3	v_4	2

A better order (with the same graph labels) would be:

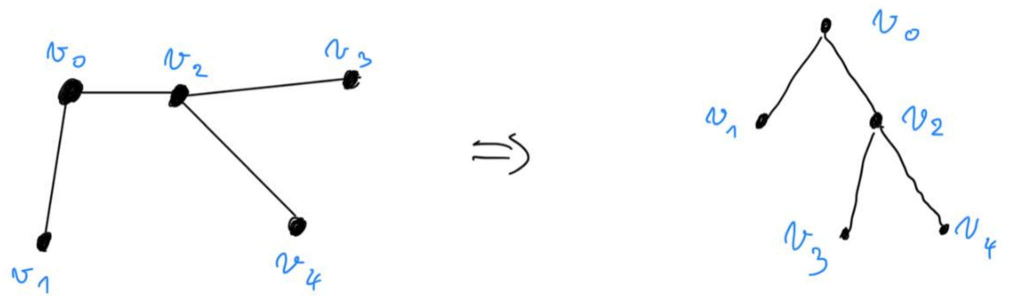
Iteration	**u**	**v**	$\boldsymbol{d(u, v)}$
1	v_0	v_1	1
2	v_0	v_2	1
3	v_2	v_3	1
4	v_2	v_4	1

Such order would allow us to reduce, for every iteration i, the range of values for j from $[i - 1 - d(u, v); i - 1 + d(u, v)]$ to $[i - 2; i]$, since we insured $d(u, v) = 1$.

To construct such order, we do the following: We run a DFS or BFS on the graph (starting from a random node) and whenever a node is first found, we add it to a tree, with its predecessor as its parent in the graph.

For the remaining of this report, we will call such tree, the **plan** of the algorithm.

For example, the previous graph would give us the following "plan" for every iteration:



This plan consists essentially in traversing the graph, starting from a node, and where we only consider the first appearance of a node to determine its parent in the "plan" tree. This results in a complexity of $O\big((V + E/B)log_2V + sort(E)\big)$ with a DFS cache-oblivious algorithm [8]. It can even be merged with the first BFS (running MR-BFS) done on the first node "v_0", to further reduce the complexity of the algorithm. In our algorithm, this node "v_0" would be chosen as random.

Then we run the algorithm $IncrementalBFS[G, v, v.parent, d(v.parent,\cdot)]$, starting from the root. We will always have $d(v, v.parent) = 1$ since the graph is undirected and these nodes are neighbors.

It has the following advantages:

- **It bounds the complexity of computing $\boldsymbol{L_i(u)}$** from $O\left(\frac{E}{B}d(u,v) + sort(E)\right)$ to $O(\frac{E}{B} + sort(E))$.

  While this does not change the overall I/O complexity of $O\big(V \times sort(E)\big)$ for the whole algorithm, since this step of the algorithm is asymptotically negligible in front of computing the first BFS of the algorithm running MR-BFS, we still feel it is a significant change, as the Incremental-BFS algorithm will be run $|V| - 1$ times. We decided to implement the AP-BFS algorithm and our extension to compare them by benchmarking them on different graphs.

  It does not change the spatial complexity $\Theta(V^2)$ either.

- **It can be parallelized:**

  It is easy to extrapolate a plan to parallelize the iterations of Incremental-BFS, while still maintaining the best-case scenario for each iteration ($d(u, v) = 1$): Each branch of the tree (and the jobs that result) can be sent to a different machine, along with the necessary BFS results.

  Since the plan is a tree, the critical path depends on the total height of the tree, which can be bounded by the diameter of the graph.

  The work is the complexity of the algorithm AP-BFS.

  AP-BFS algorithm could also be parallelized, by running AP-BFS on a subset of the vertices of the graph.

- **It is easier to understand:**

  This last part is highly subjective, but worth noting in our opinion.

  The intent and benefits behind such order is, in our opinion, easier to understand than the order provided by the Euler Tour of a spanning tree of a graph. While the Euler Tour of a spanning tree is a valid order, it is not a straightforward method to iterate over nodes.

Our idea on this extension came while trying to understand if the Euler Tour of the tree had useful properties, and why it was chosen.

Added Complexity of our method:

The construction of the "plan" of iterations can be done with a classical DFS algorithm in $O(V+E)$ compared to a complexity of $O(min\{V+sort(E), sort(E)\times log_2 log_2(V)\})$ to find a spanning tree [2]. The Euler Tour is constructed cache-obviously in $O\big(sort(V)\big)$ [7], and marking the nodes is done in $O\big(sort(E)\big)$ I/Os. Resulting in a total complexity for creating an order out of the Euler tour of a spanning tree of the graph of:

$$O(min\{V+sort(E)+sort(V), sort(E)\times log_2 log_2(V)+sort(V)\})$$

Our method if run with a DFS can be done cache-obviously in $O\big((V+E/B)log_2 V+sort(E)\big)$ [8]. If run with a BFS (with MR-BFS for example) can be done cache-obviously in O(V + sort(E)) I/Os if the MR-BFS algorithm is adapted to store one of the parents of a node. This would allow us to merge the first and second step of the AP-BFS algorithm (The second step is running the MR-BFS algorithm on the first node).

Excluding this part, the rest of the algorithm has a complexity of $O\big(V\cdot sort(E)\big)$ I/Os.

We store the same amount of data as the original method when computing the plan, hence extra space used by our extension is the same as the AP-BFS method, at O(V).

For our implementation however, we decided to do this with a DFS for this part, for two reasons:

- It is simpler to implement
- We did not implement the best method highlighted in the paper to cache-obviously compute the Euler Tour of a spanning Tree of a graph, so we had to leave this part of the algorithm out of the benchmark, meaning the quality of our implementation for this part of the algorithm has no impact on the benchmark anyway.

Correctness of our method:

Our method finds a different ordering compared to find a spanning tree, followed by finding a Eulerian tour, then mark ordering. Both provide an ordering, which can be viewed as a "meta parameter" of MR-BFS and Incremental-BFS. The correctness follows from MR-BFS and Incremental-BFS, which we do not change. Notice DFS clearly visits all vertices in the graph, so the following BFS will be performed from each v ∈ V [G].

Full algorithm of AP-BFS extension

1. Create a spanning tree out of the graph the following way: Start at a random node and traverse the graph (BFS or DFS). When a node is first encountered, add it to the tree, with its predecessor as parent in the tree.
2. Run MR-BFS on the first node
3. For every node of the tree (except the root node), run:
   Incremental-BFS(Graph, node, node.parent, BFS results of node.parent)

Step 1 and 2 could be merged.

Benchmark of our extension of AP-BFS versus AP-BFS

We implemented AP-BFS and our extension in Java, as well as its parallelized version, using the library JGraphT [3].

Source code is available on GitHub (It was packaged for the benchmark only, not in a way to be reused as a third-party library).

To do so, we also implemented MR-BFS, Incremental-BFS and an algorithm to generate the Euler Tour of a Spanning Tree of a Graph (The benchmark measurements did not consider this last algorithm, nor the creation of our "plan" with a DFS of the algorithm, as we did not have time to implement the algorithms used by the paper studied, that came from another paper [2], and it is not the focus on your extension).

We ran the 2 algorithms on the same 3 graphs, randomly generated, with the Erdős–Rényi model [4][5], with the generator provided by JGraphT [6]. We ran the tests 3 times, for the following tree seeds: "100L", "200L" and "300L", we used seeds to have reproducible results.

We computed 3 graphs $G(n, M)$ (n: number of nodes, M: number of edges), with the following parameters:

Graph	$\boldsymbol{n = \lvert V \rvert}$	$\boldsymbol{M = \lvert E \rvert}$
1	1 000	50 000
2	1 000	200 000
3	1 000	400 000

The first graph is sparse, the last one dense, and one in between.

We restricted our benchmark to connected graphs, for ease of implementation of the algorithms. This is reasonable, as connected graphs can generally provide worse performance, and people usually care about worst case performance of deterministic algorithms. Consider the case where the graph is not connected. For Dijkstra's algorithm, if u, v are not in the same connected component, $d(u, v) = inf$. Most of the time, other APSP/SSSP algorithms do the same.

If the graph contains several connected components, once one can recognize the connected components, and reduce the APSP problem to several smaller problems of APSP, which shall yield shorter running time (or less I/Os for external memory model) overall.

To generate connected graphs, we added edges to nodes of each connected component until the graph is connected. We used the ConnectivityInspector class from JGraphT [9] to do so.

We measured the time taken to compute all the BFS for these 3 graphs (excluding the part where we compute the order in which we will run the BFS), for the following algorithms:

- The Floyd-Warshall algorithm implemented by JGraphT [10], to have a point of comparison to judge the quality of our implementations of MR-BFS and Incremental-BFS. This algorithm has a complexity of $O(V^3)$ operations. Since the number of edges does not influence on the complexity, we expect it to have more constant results on the benchmarks, compared to our implementations.
- Only the MR-BFS algorithm (starting from each node), as a point of comparison of our implementation of the Incremental-BFS algorithm, compared to the MR-BFS algorithm.
- Using the AP-BFS algorithm,
- Using our extension of the AP-BFS algorithm

The last 3 algorithms are external memory algorithms, so we are interested in measuring the I/O. However, there are multiple "levels" of I/O: It can mean the I/O between the L3 cache and the RAM (external memory), the RAM (cache) and Disk (external memory). In our benchmarks, with the size of our graph, the only I/O measurement that made sense is the I/O between RAM and lower-level caches, as the graph was small enough to be stored fully in RAM.

Since it is difficult to measure this I/O, we decided to measure the execution time of each algorithm on the same graph, on the intuition that it would be roughly proportional to the I/O cost.

Benchmark on graph 1 (sparse)

We obtained the following measurements (in ms):

Implementation	**Duration: Seed 100**	**Duration: Seed 200**	**Duration: Seed 300**	**Average**
Floyd-Warshall	2352	2384	2375	2370,33
MR-BFS only	3621	3337	3669	3542,33
AP-BFS	6893	6507	7022	6807,33
AP-BFS extended	6917	6526	6892	6778,33

In that benchmark, all the algorithms duration had the same order of magnitude.

Floyd-Warshall algorithm performs much better, probably because its implementation is better. There is also less variation on its results, since its complexity does not depend on the number of edges.

The implementation relying on MR-BFS only performs significantly better than our AP-BFS implementations. This highlights that the quality of our AP-BFS implementations is probably not enough to be able to compete with our implementation of MR-BFS, with some overhead that penalizes it.

Our extension of AP-BFS performs marginally better (to the extent that it is difficult to draw a clear conclusion that our extension is better).

Benchmark on graph 2

We obtained the following measurements (in ms):

Implementation	**Duration: Seed 100**	**Duration: Seed 200**	**Duration: Seed 300**	**Average**
Floyd-Warshall	3386	3291	3311	3329,33
MR-BFS only	42798	43693	45413	43968,00
AP-BFS	60798	59197	69394	63129,67
AP-BFS extended	60824	59250	69310	63128,00

In a denser graph, the Floyd-Warshall algorithm also performs much better (10x times less), and the difference is more and more stark: Since its complexity ($O(V^3)$) doesn't depend on the number of edges, this difference in result is not surprising, and only increases as the number of edges increase.

The difference between the "MR-BFS only" implementation and the AP-BFS implementations is also significant, but slightly less than for a sparse graph, at around 30%. This highlights that our AP-BFS implementations' quality is probably not satisfying enough, resulting in overhead that cause this difference.

Both AP-BFS and AP-BFS extension implementations performed similarly.

Benchmark on graph 3 (dense)

We obtained the following measurements (in ms):

Implementation	Duration: Seed 100	Duration: Seed 200	Duration: Seed 300	Average
Floyd-Warshall	4221	3668	3698	3862,33
MR-BFS only	207527	231288	239375	226063,33
AP-BFS	288809	304030	311269	301369,33
AP-BFS extended	284111	301259	306183	297184,33

The Floyd-Warshall algorithm performs more than 50x better than the other implementations, as noted previously due the difference in complexity.

The MR-BFS only implementation is around 30% better than the AP-BFS implementations, just like for the previous sparser graph and probably for the same reasons (worst implementation of AP-BFS).

Lastly, the implementation of our extension of AP-BFS performed better than AP-BFS, but only marginally saw, probably due to the overhead of the implementation.

Conclusion of the benchmarks

From the benchmarks we can draw multiple conclusions:

It is difficult to test an external-memory model at our scale, especially since we do not have access to the I/O measurements. The duration of the algorithm, used instead, may not be the most appropriate replacement to measure an external-memory algorithm's performance.

Our implementation of AP-BFS (and its extension) has some overhead that makes it behave poorly compared to an implementation only relying on MR-BFS. It highlights the difficulty of implementing external-memory algorithm at our scale, and with Java.

Furthermore, it was not possible with just the benchmark results to draw a conclusion that our extension significantly improves the performance of the AP-BFS algorithm. Either the improvement due to the extension was not significant compared to the overhead of the rest of the algorithm, or our extension just does not provide a significant improvement. However, it is better than the original AP-BFS algorithm in some cases and does not perform worse in general, so our extension is valid. Further work, with better implementations and tests would however be needed.

Notice that Floyd-Warshall has a (time) complexity of $O(V^3)$, while AP-BFS has a (I/O) complexity of $O(V\ sort(E))$. Although they cannot be compared directly, this tells us that Floyd-Warshall's running time does not increase significantly when the graph becomes denser, while AP-BFS does. This implies that the Floyd-Warshall's advantage is larger when the graph is denser, when comparing with AP-BFS, which agrees with the results of the benchmark. The advantage may also come from the fact that Floyd-Warshall is designed under time complexity analysis and is written by a better coder.

Conclusion

In this report, we provided the details of an extension of the AP-BFS algorithm, a cache-oblivious algorithm to solve the APSP problem. Our extension bounds the complexity of a step of the AP-BFS algorithm, by providing a better order to run the successive BFS algorithms, relying on the particularity of the Incremental-BFS algorithm.

We implemented this algorithm and our extension, to compare it to an implementation relying only on the MR-BFS, and to a third-party implementation of the Floyd-Warshall algorithm, from JGraphT.

From our experiments, we notice that for the external memory model, it is easier to deal with theoretically. When one tries to implement them, as the assumption on computational operations are free may not be applicable or hard to control with normal computer settings, we do not observe a good result for the external memory algorithms. This may also be due to two other factors. Firstly, we cannot measure the number of I/Os directly, so we measure the time taken and assume it is proportional to the number of I/Os. This is true when all assumptions for the external memory model are held, but not necessarily true in practice. Secondly, the test cases may not be large enough to exploit the advantage of using better external memory algorithms. It is possible that all computations are done within the cache.

Nonetheless, AP-BFS extended does not perform worse, and sometimes perform slightly better than AP-BFS, which shows that our extension is valid, but would need a better implementation and more adapted tests to fully quantify the improvement it provides.

References

1. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In Proc. 10th SODA, pp. 687–694, 1999.
2. L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In Proc. STOC, pp. 268–276, 2002.
3. Dimitrios Michail, Joris Kinable, Barak Naveh, and John V. Sichi. 2020. JGraphT—A Java Library for Graph Data Structures and Algorithms. ACM Trans. Math. Softw. 46, 2, Article 16
4. Erdős, P.; Rényi, A. (1959). "On Random Graphs. I" (PDF). Publicationes Mathematicae. 6: 290–297.
5. Bollobás, B. (2001). Random Graphs (2nd ed.). Cambridge University Press. ISBN 0-521-79722-5.
6. JGraphT - GnmRandomGraphGenerator - https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/generate/GnmRandomGraphGenerator.html
7. L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In Proc. STOC, pp. 268–276, 2002.
8. Table 1 of: Arge, Lars & Bender, Michael & Demaine, Erik & Holland-Minkley, Bryan & Munro, J.. (2007). An Optimal Cache-Oblivious Priority Queue and Its Application to Graph Algorithms. SIAM J. Comput.. 36. 1672-1695. 10.1137/S0097539703428324.
9. JGraphT – ConnectivityInspector - https://jgrapht.org/javadoc-1.0.0/index.html?org/jgrapht/alg/StrongConnectivityInspector.html
10. JGraphT – FloydWarshallShortestPaths - https://jgrapht.org/javadoc-1.3.1/org/jgrapht/alg/shortestpath/FloydWarshallShortestPaths.html