Software Development Project - Graph Shortest Paths
Fall 2016-2017

Vrachas Christos
1115201300024

Papatsoris Ioannis
1115201300137

1 Compilation - Execution - Unit Testing

Type make for compilation

How to run: **./gsp <threads>**
where <threads> is the number of threads to be used for query processing, defaults to 6.
We recommend adjusting this value according to your machine for best performance.

Run with input redirection: **./gsp <threads> < <graph_file> < <workload_file>**
Make sure the last line on <graph_file> is an *S*. If you manually concatenate the two files in one and run **./gsp <threads> < <input>**, there needs be a newline after the *S*.

The GoogleTest library was used for some unit testing, which is included in the tests/googletest folder. To perform the tests, run **make** inside the tests/ folder and **./check**. Make sure to change the defined value INITIAL_MAX_LIST_NODES in Graph.hpp to 1 and MAX_NEIGHBORS in ListNode.hpp to 3.

2 General Optimizations

Although the use of a Hash Table regarding lookup about whether an edge between two nodes already exists has been suggested by the instructor, it turned out to hinder performance and is not used, but it still exists commented out within the code.

During the development of the first part, we tried two optimizations on the bidirectional algorithm. For the first one, we would expand the side having the fewest number of nodes on its fringe, instead of moving on both sides simultaneously. Later on, we thought about taking into consideration not just the number of nodes on each fringe, but also the number of the grandchildren of them. By these two additions, the search is not be fully balanced, but it is much faster.

For the explored set of the search, we use an array in which we mark which nodes have been explored with a visit_version_ID. Instead of clearing the structure between searches, we simply use a bigger ID for the next search.

3 Part 2

3.1 Static Graphs

For the static graph case, we implemented a set of extra structures: Strongly Connected Components, Hypergraph (Directed Acyclic Graph), and the Grail Index. After many tests, we have concluded that in order to gain more on query answering, the number of intervals per node, should be **five**. Moreover, if the answer to a Grail query about whether node A is connected with node B is MAYBE, we also perform the mirror query, which is whether node B is connected with node A. Finally, there is a probability for the initial graph, to be a DAG. In such case, there is no need to create a hypergraph, which is a somewhat costly operation, but the initial graph is used directly, for the Grail Index. Whether the graph is a DAG or not, is spotted when we get the total SCCs of the graph. If the number of SCCs matches the initial graph's number of nodes, then there is no need to build the Hypergraph. In such case, on the later algorithms creating the Grail Index, we take care of self-cycles.

3.2 Dynamic Graphs

Three different approaches were tested for the Connected Components Update Index. One of them was based around maintaining a list for each initial component, that would include each component it connected with. Every time 2 components had to be connected, 4 memory copy operations were required to update the lists. With an additional optimization, the lookup about whether 2 components were connected would be accomplished in constant time or in $O(m)$ at worst, where m is the number of components connected to the given component. However, in practice the worst case would never happen even in the largest dataset, so it was always constant time.

One other approach was to hold an array of size equal to the number of initial components and assign a shared ID number across connected ones, resulting in constant time for connection lookup. When connecting 2 components, if one of them was not already connected with any different one, the connection would also be made in constant time. If not, the most naive way to update the structure would require $O(n)$ time where n is the number of initial components, which could be improved on with various optimizations. But in practice, the second case would only occur 23 times out of 1 million in the largest dataset.

The real problem with the above approaches is that they cannot be tweaked so that they can be shared between various versions of the graph and be used in parallel, without becoming memory inefficient. For each batch, that would require V times more memory, where V is the number of different graph versions in the current batch. The second approach was altered to work in parallel, and while time performance was identical to the final approach, memory usage on the largest dataset was off limits. Although the main thread could check whether 2 components are connected right away without assigning the job to the Scheduler, thus eliminating the need of multiple graph versions, it would give up completely on parallelism and perform worse than the third approach, which is presented bellow:

Here is the final approach, which can operate in parallel between multiple versions of the graph efficiently.

Algorithm 1 CC Update Index

```
procedure ConnectTwoComponents
    $a \leftarrow min\{c1, c2\}$
    $b \leftarrow max\{c1, c2\}$
    while $update[a]$ NOT EMPTY and $update[a].version \leq current\_version$ do
        if $update[a].next == b$ then
            return 'Already connected'
        $tmp \leftarrow a$
        $a \leftarrow min\{update[tmp].next, b\}$
        $b \leftarrow max\{update[tmp].next, b\}$
    $update[a].id \leftarrow b$
    $update[a].version \leftarrow current\_version$
    return 'Connection successful'
```

We hold an array of *initial_ number_ of_ connected_ components* size. Each cell represents a component and holds an integer which is the next connected component that the current one is linked with, with all of the cells being empty at the beginning. Considering that the two components $c1$ and $c2$ we want to connect do not seem to be already connected from the CC Index, we consult the Update Index and connect them in the above manner.

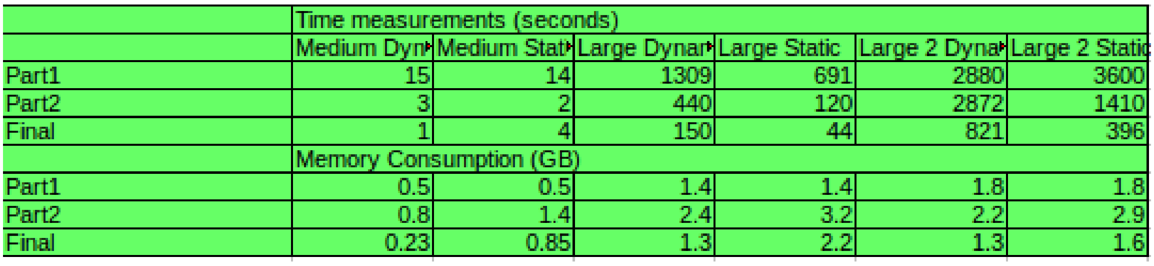
To check if two components are already connected using the update index, we use the same iteration in the above algorithm, but without altering the index.

While index rebuilding according to a metric regarding Update Index usage was implemented, it did not improve performance in any of the approaches, even when rebuilding was performed at every batch and without even taking into account the extra time required for rebuilding. As a result, no rebuilding is done at all. This can be changed by altering the THRESHOLD defined value in CC.hpp file (float between $[0, 1]$)

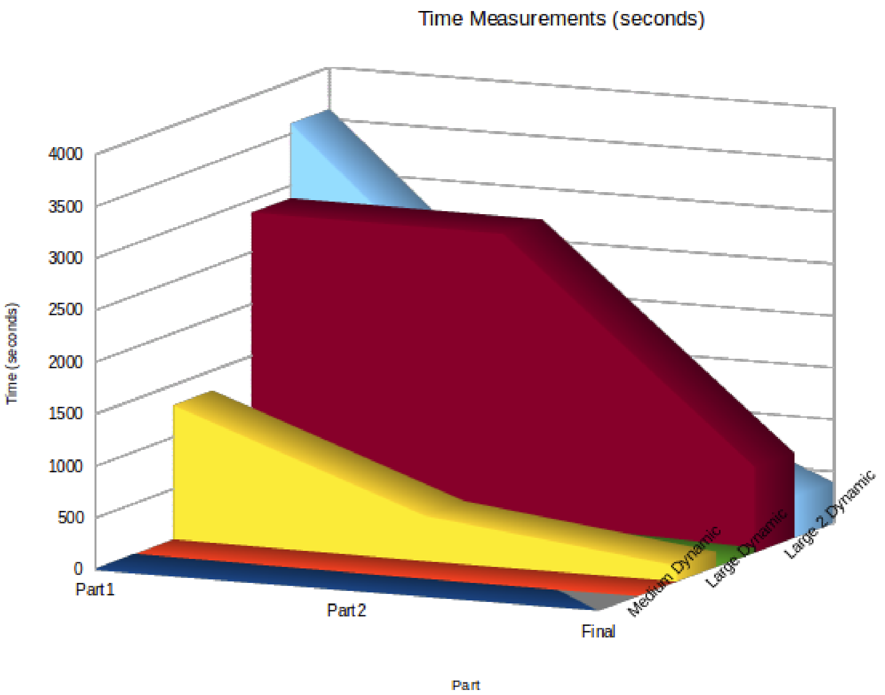
4 Part 3

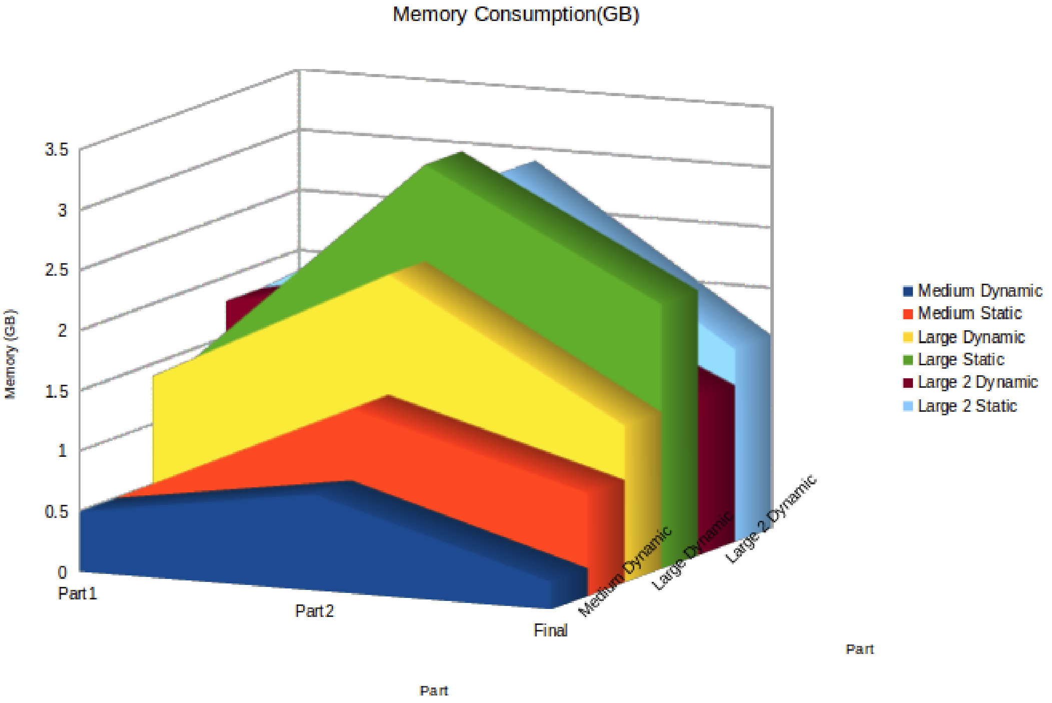
The distinction between static and dynamic jobs is accomplished with the use of inheritance from a generic Job class and polymorphism. In order to avoid multiple mutex locks and unlocks, each thread is assigned a number of jobs, which is given by the formula $\lfloor \frac{\#jobsperbatch}{\#threads} \rfloor$. The first thread might get some extra jobs. To perform searches in parallel, each thread has to use its own explored set.

5 Time measurements and memory consumption



	Time measurements (seconds)					
	Medium Dyn	Medium Stat	Large Dynar	Large Static	Large 2 Dyna	Large 2 Static
Part1	15	14	1309	691	2880	3600
Part2	3	2	440	120	2872	1410
Final	1	4	150	44	821	396
	Memory Consumption (GB)					
Part1	0.5	0.5	1.4	1.4	1.8	1.8
Part2	0.8	1.4	2.4	3.2	2.2	2.9
Final	0.23	0.85	1.3	2.2	1.3	1.6





5.1 System setup

Intel Core i5-4670 @ 3.4Ghz (x4 cores)
8GB RAM - 128GB SSD