HELLENIC REPUBLIC
National and Kapodistrian
University of Athens

Software Development Project

Minimal Step Path

2016 – 2017

Contents

General Information

How to run

There is a makefile compile with make all . And run under ./MinPathGraph .

IMPORTANT : All the graph files should end with the letter S workload and result files excluded. Or there would be a segmentation fault .

Folder Structure

Dataset folder holds the all the datasets that are to be loaded. In the results folder are stored the txt files witch they hold the results of the program .

Set program Parameters

To run program Parallel go to the main function and uncomment the executeWorkLoadParrallel function and comment out the rest of the others.

To change metric go to the weaklyConnectedComponent header and change the Defined constant .

To change Tags Count go to the Grail header and change the defined constant.

To change Thread Count go to JobScheduler and change the Defined constant.

Some words about the develpment

Throughout the program the structure that was used the most was the dynamic hash, because of the small complexity that it offers. Even though, it is hard to find a hash function that distributes the keys uniformly Knuth multiplicated hash function and FNV hash was used, which gave satisfying results. With the program space memory was sacrificed in order to gain better running time. The indexes that were used improved the time by 3x, as shown from the measurements showcased in part four. The dynamic hash was used as a structure because time improvement was seen in addition to the ease of implementation. Although, the time needed for the creation of the indexes is significantly small compared to the workload the dynamic hash was chosen for the creation of these indexes so the best possible running time could be obtained for the program. Some problems were faced when the large data was ran. The ram memory needed to be expanded because four gigabytes was not sufficient. The minimum requirements for the large graph is memory of 6 gigabytes. It was attempted to abide by the prototype given as much as possible. The Tarjan algorithm that was given was completely inefficient in the recursive mode. Thus, it was switched to iterative. This switch was difficult, because of the uncommon placement of the recursive step. The grail index was tested through multiple tags. The improvement followed an algorithmic scale. The search for the distance between the two nodes within the graph showed better results using the bidirectional BFS instead of a common BFS search. Even though, bidirectional BFS and common BFS have the same complexity. The use of threads made huge improvements in the program’s run time. The graph was chosen to be global as we wanted to minimize the args passed to the threads. The debugging was fairly hard . Ass a suggestion for future projects we would say to give ass smaller graphs with more extreme cases so that we don’t need to wait long time to see the results .

Graph Struct

The graph structures consist of two subgraphs in which one depicts the outgoing edges from the nodes. There is, however, another one that depicts the ingoing edges from the nodes. Also, within the graph structure is encapsulated the strongly connected components, the weakly connected components, and the grail index. Each subgraph has a dynamic hash used as an inverted index for fast checking O(1) of duplicate edges. The inverted index is also used in estimating the strongly connected components in the get next neighbor function.

General Purpose Structures

The general purpose structures are a dynamic hash, list, stack, and a queue. More information can be obtained through the code

Strongly Connected Component

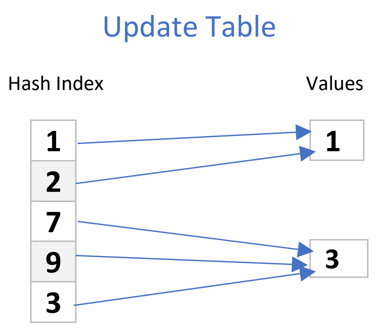
Tarjan algorithm was used in the iterative form. It was observed that, if it was left in the recursive form when given large sized—and even medium—data sets, stack overflow occurred.

Weakly Connected Component

The weakly connected components use an update index with complexity O(1). The update index is composed of a dynamic hash and a queue which helps in the deallocation of the values held by the dynamic hash. Otherwise the deallocation process would be of complexity $O(n^i)$ (i=2).

Update Index

The Update Index hash uses as keys the components id. When a new edge is inserted the nodes are checked under which connected component they belong to. If they belong to different components, two entries are created with their respective keys. One keys has the component id of the first node and the second key has the component id of the second node. A value is created for both of these entries by heap allocation of which the value is the lesser of the two components id. In this way, it can be checked if two separate components were merged into one. When merging occurs the new component always takes the id of the lesser of the two component ids.



Example of Update Index . If edge that connects components 7 and 1 was added then we change the value 3 to 1

The versioning of the Update Index was implemented with a Dynamic Hash that takes a string as a key witch is composed from the two components that the nodes to be insert are. For example if NodeS 2 and NodeF 3 witch they belong to components numbers 1 9 accordingly. A key “1-9” is created and for the value we store the current version. Then when we are going to consult the Update Index for those nodes we check the versioning hash and look if the version of connecting those two components is the same as the query version. Metric variable should be <1. Otherwise Component Update won’t happened .

Hyper Graph

The hyper graph consists of an array table of HyperGraphNodes and an array table which holds all the root nodes of the hyper graph (root nodes are nodes that don't have ingoing edges, and at least one outgoing edge). HyperGraphNode hold a Dynamic Hash with all the neighbors of the node so we can check double edges in O(1) and an array with all the neighbor nodes so we can get all the neighbor node fast if we search serially without having to search the dynamic hash . Searching serially can take long in the dynamic Graph as it might have many empty cells cause of bad hash.

Grail Index

The grail index consists of an array table of tags for each node of the hyper graph. Multiple tags are available. DFS search was used for the creation of the grail index. For more information about Grail R24.pdf.

Job Scheduler

The job scheduler is used to synchronizer the threads so the queries are execute in parallel and create the thread poll . Used mutexes are queueMutex to synchronize that each thread get atomically a job. BatchMutex and condition variable cond_batch_finished that help to block the and unblock the thread when a new batch of jobs is available. QueriesUsedUpdateMutex that syncronizes the QueriesUpdate counter. ParkedMutex as well condition variable cond_parked are used to implement a barrier like function . The given pthread_barrier function was not chosen due to the choice to keep the integrity of the prototype.

IMPORTANT: We chose to print the results not within the thread. Synchronizing them to print the results one at time in the correct order. But rather we passed the same table in each thread and let the thread printing their results in a cell of the table witch was given from the jobId and afterwards print the table from the main thread. That way we had much better running times of the program.

Performance Statistics

System Information

The tests were performed :

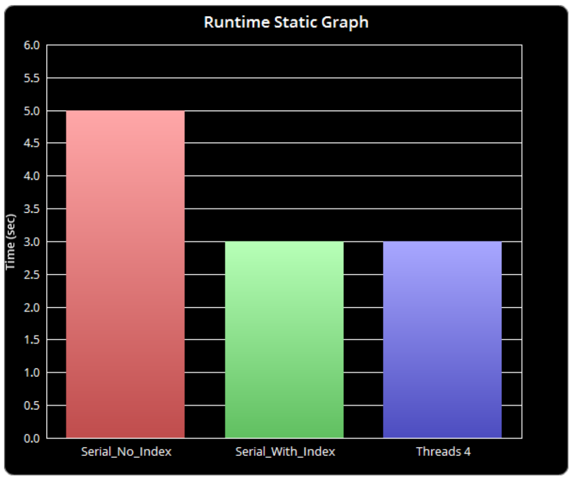
Operating System: Linux Mint 17 Qiana

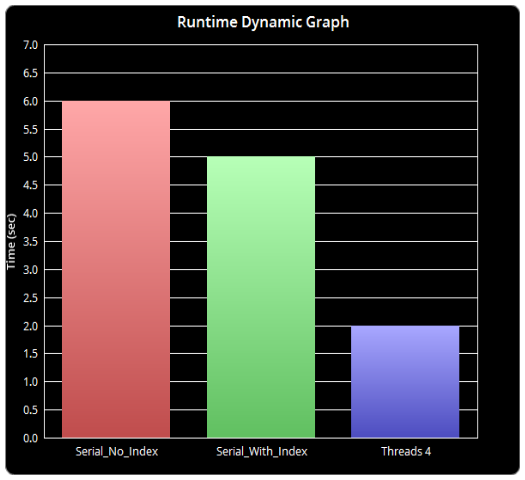
Processor: Intel Core i5 3230M

Ram Memory: 12 Gigabytes DDR3

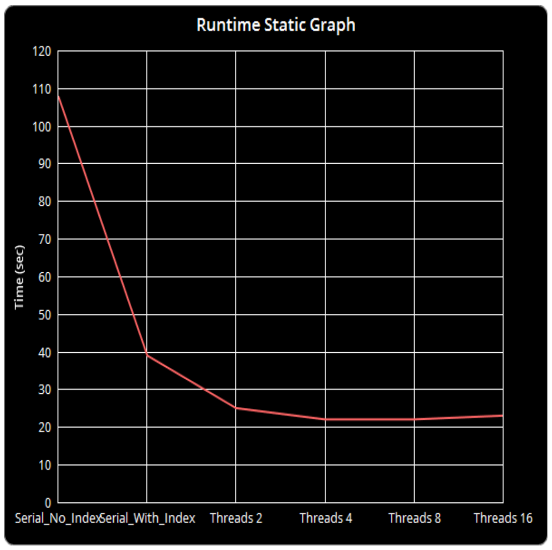
Performance Charts

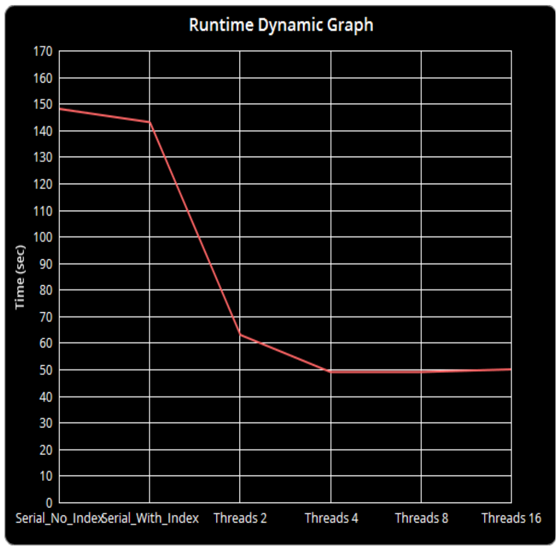
Small Graph

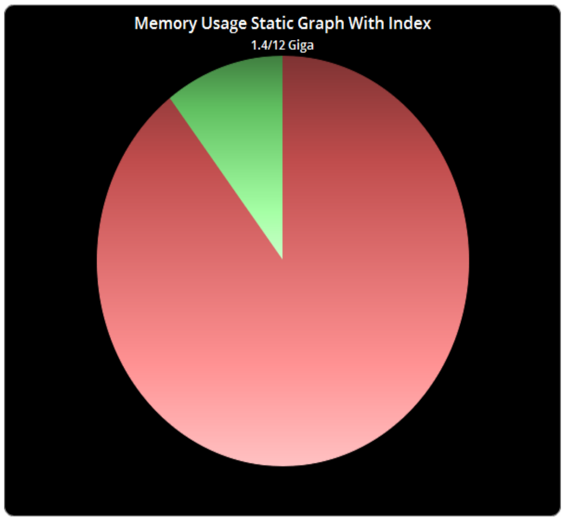


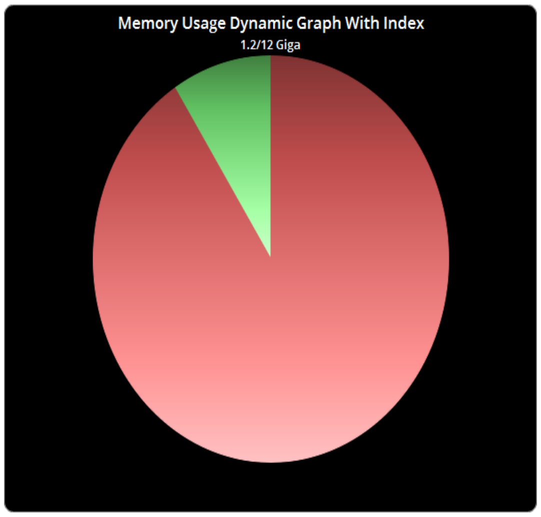


Medium Graph

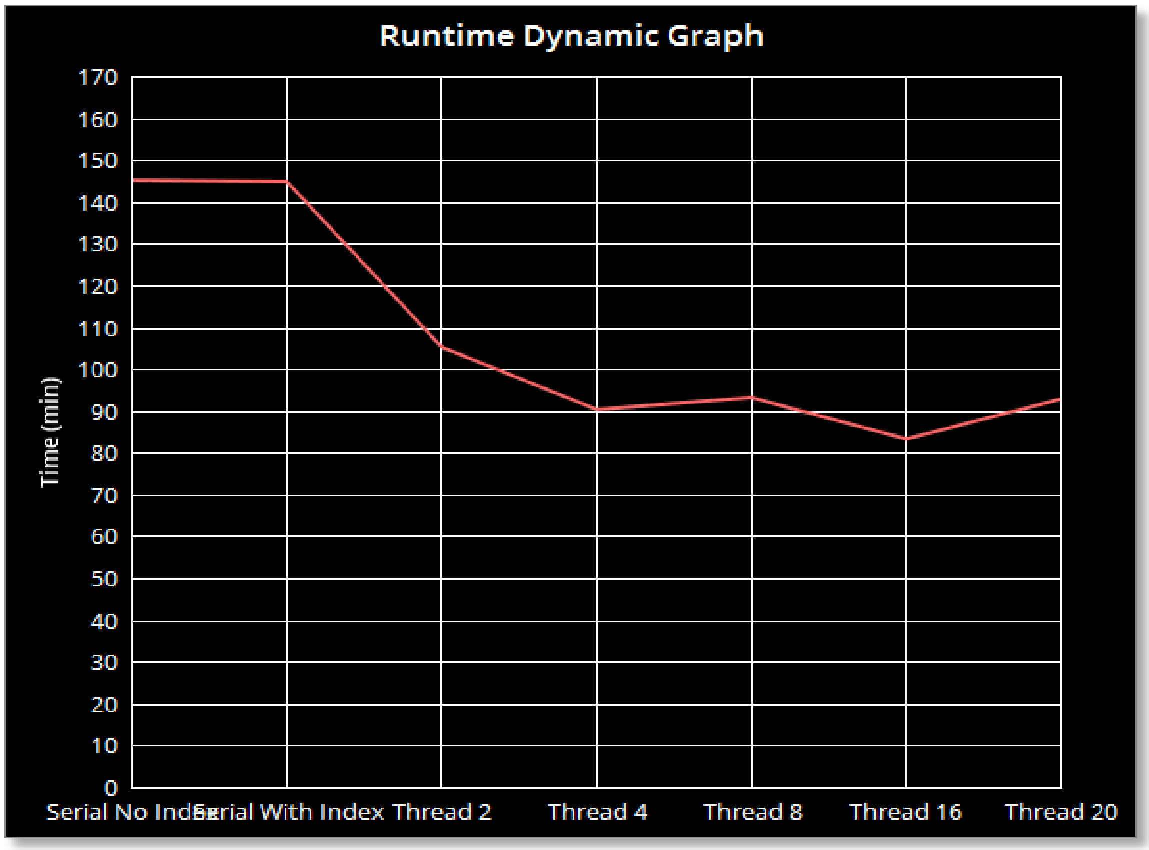


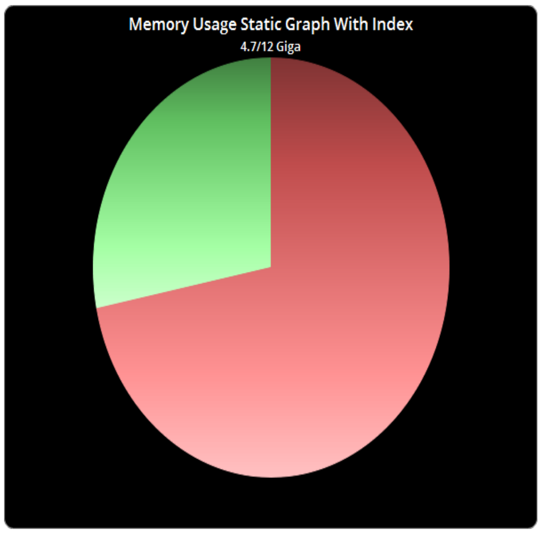


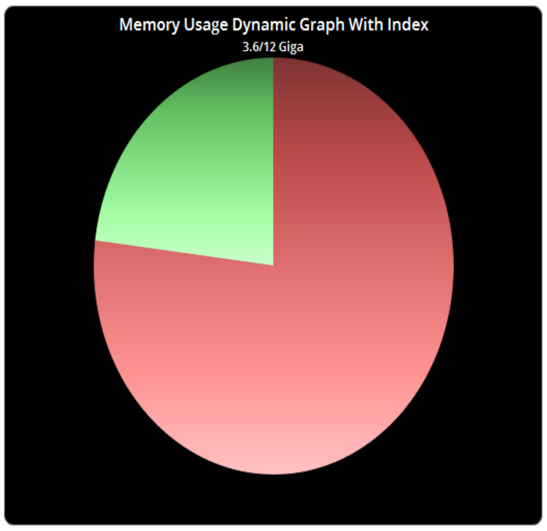

Memory Usage Static Graph With Index
1.4/12 Giga


Memory Usage Dynamic Graph With Index
1.2/12 Giga

Large Graph


Runtime Dynamic Graph
Time (min)
170
160
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
Serial No Index
Serial With Index
Thread 2
Thread 4
Thread 8
Thread 16
Thread 20





Conclusion

It was observed that the Strongly Connected Components incorporated with the Grail Index improved the programs runtime tenfold; The runtime was improved even more so once multithreading was combined, regarding static graphs. The only task at hand regarding the Dynamic graph and Indexes was to find the weakly connected component and employ multithreading. In such a case, significant improvement was observed. Contrary to popular assumptions an increased Number of tags for the grail did not speed up the program. In fact, the exact opposite occurred. Due to this, it was decided to keep the grail tags 3 and test the program.

- Note we didn't include the Static statistics for the large graph cause we ran out of time. This is left as an exercise for the reader.

Developers

Θεοδοσόπουλος Γεώργιος (Theodosopoulos Georgios)
AM 1115200900055

Φατούρος Κωνσταντίνος (Fatouros Konstadinos)
AM 1115200900234

Ζηκίδης Φωκίων (Zikidis Fokion)
AM 1115201100037