PROJECT 2: REACHABILITY QUERIES IN DIRECTED GRAPHS

System and device programming

A.A 2019-2020

Professor

Stefano Quer

Team

Luca Barco (s276072)

Stefano Bergia (s276124)

Giuseppina Impagnatiello (s270325)

Summary

0. Folder structure

In the project directory you will find two sub-directories containing the source code corresponding to the **two different versions** of the algorithm that we implemented.
Both of them can be run passing *the input Graph file*, *the number of labels* and the *input Query file* as parameters via command line.

N.B. *We had some encoding issues with the benchmark files and couldn't read them correctly; we solved this problem manually copying their content on new files. Please if you encounter any problem with them, see the benchmark folder in the following github repository. It contains the files we used to test our code, so they should correctly work.*

0.1. Github repository

We provide you with a link to a Github repository: **Q2-GrailParallelImplementation**, containing a Visual Studio project that allows you to run a complete program with both the implementations. This can be done through a command line interface that allows you to choose the implementation and the input files, either from the benchmark graphs or from a generation on-the-fly mechanism (using the GraphGenerator-StQ program).

1. Introduction: Grail approach and why two different versions

When we started working on this project, we decided to take into account both the **memory consumption** and the **execution time**, especially considering the size of the larger graph in the benchmark suite, and so we ended up implementing two different versions.
The first one is **main-memory based**: it is faster, but also the most memory consuming.
The second one is **file-based**: it is slower because of the disk overhead, but allows to reduce the memory consumption, so it fits better for very large and/or dense graphs.

At the end of this README file, we provide you with some memory-vs-time statistics on the benchmark suite, both for the in-memory and in-file version.

2. Reading the graph input file

2.1. Information storing

In the **in-memory version**, first of all we initialize an ADT graph G with the **GRAPHinit** function: it will be used to store all the information relative to the graph reading.
Both the two classical ways of storing data (adjacency list and matrix) are provided, but after some experiments on time execution, both on the benchmark test suite and on the on the fly generator graphs, we ended up choosing the **adjacency list** that has proven to be faster in most of the cases.

In the following table, you can see the time required to read the graph and create the labels for both list and matrix implementations. In most of the cases, even if we provided small graphs as input, the list works better than matrix. Furthermore, the matrix is generally more memory consuming.

File	#Nodes	#Edges	List	Matrix
500-749.gra	500	746	0.019s	0.025s
500-1022.gra	500	1018	0.029s	0.022s
500-2473.gra	500	2473	0.036s	0.044s
500-5070.gra	500	5070	0.068s	0.087s
500-12976.gra	500	12975	0.16s	0.18s
500-18121.gra	500	18123	0.21s	0.27s
500-23685.gra	500	23684	0.37s	0.38s
500-28360.gra	500	28361	0.36s	0.41s
500-30499.gra	500	37365	0.43s	0.55s
500-36893.gra	500	45429	0.54s	0.69s
500-109462.gra	500	109465	1.33s	1.47s
500-123702.gra	500	123691	1.62s	1.52s
1000-495918.gra	1000	495845	7.15s	6.83s

For the **File version,** the representation is totally different; the graph is never completely stored in memory but it is accessed each time directly on the .gra file. To do so, the graph is scanned a first time to create an index file, which stores the following information for each node: its index (they are stored in order), its offset in the .gra file with respect to the beginning of the file, the size in character in the .gra file and its number of children.
The index file will also store the labels computed using the GRAIL algorithm.
These operations are done by calling the function **generateIndex**.

2.2. Threads synchronization

In the **in-memory version**, the graph reading is done through the **GraphParallelRead** function, that sets up 5 threads (the number of threads can be changed in the main.cpp source code, up to 8 threads) calling the **ThreadReader** function for all of them and waiting for their termination.
Each thread reads its own block of the graph input file (defined by an offset and a size) and, for each row, it inserts the parent node in a buffer and loops taking each child and inserting the edge parent-child in the graph structure.

In this case the **File version** behaves in a similar way; the function **generateIndex** creates 2 threads (just two because it is empirically faster, but also in this case the number of threads

can be changed) and splits up the graph file in equal parts. Each thread executes the function **parallelRead** which reads one of the batches of the file computed in the previous step.

2.3. Finding the roots

For the **in-memory** version, the ThreadReader function also provides a mechanism to find the graph roots: in the graph structure we have a bitmap with a bit set to zero for each node; then during the reading, the **setBitmap** function sets to 1 the bit associated with each child node, indicating that it is not a root (the bitmap writing operation is protected with a mutex). In the end, the bits still set to 0 are the roots.

This process is exactly the same for the **File version**.

3. Creating labels

3.1. Threads synchronization

In the **in-memory version**, the creation of the labels is made through the **RandomizedLabelling** function: it initializes a matrix in the graph struct, that will be the storing point for all the labels, creates d threads (where d is the number of labels, passed as input from the user) and waits for their termination.
Each of the threads has its own tread_t_label structure, containing a bitmap for the visited nodes and the right part of the labels r (initialized to 1). The thread then executes the **singleIndexRandomizedLabelling** function which randomizes the order of all the nodes of the graph, looks for the roots and calls the RandomizedVisit on them.
The latter is a recursive function that implements the pseudo-code provided in the GRAIL paper:

- if the current node has been already visited,it returns the node left label.
- else, it gets the number of children, randomizes their order, and recurs on each of them. When all the children return, it calculates the **current node label** (based on r and on the minimum of the children left label), stores it in the graph structure and returns the node left label.

The **File version** is the same from a logic point of view, the big main difference is that the index file is unique for each thread, and the labels for each node are stored on the same record. So it is required to implement some mutual exclusion technique to access the records in the index file. This is done using an array of **struct file_sync**, one for each thread. This struct contains a DWORD to store the index of the node that is being occupied by the corresponding thread, a semaphore to ensure mutual exclusion and another DWORD to store the number of threads currently waiting for that node to be available.
When a thread wants to block a node, it calls the **lockIndex** function which accesses the file_sync array (the array is also protected by a mutex) and checks if there is any thread blocking the record. If nobody is locking that part of the file, then the thread writes the index in its own file_sync and continues its execution. If the record is locked, the id of the thread locking it is retrieved and the current thread updates the count of thread waiting on the “id_thread” file_sync, and waits on its semaphore.
When the locking thread wants to release the node, it calls the **unlockIndex** function, which sets the locked record index in the current thread to -1 and releases the semaphore for an amount equal to the number of threads waiting on it (it then sets the number of waiting threads to 0). At this point one of the waiting threads will lock the index for itself and the others will be locked again, thanks to the while(TRUE) cycle in the lockindex function.

4. Resolving queries

4.1. Threads synchronization

In the **in-memory version**, the queries resolution is solved through the **QueryResolutionSetup** function: it chooses an appropriate number of threads, based on the number of queries to be solved, creates them and waits for their termination.
Each of the threads then executes the **ResolveQueries** function that reads a single query from the input file (the access to the file pointer is protected with a mutex, to reach a better thread synchronization also in reading operations).
In the first place we implemented a draft version with one **queryBuffer** for each thread: it would have served to store a certain number of the file rows, in order not to access the file too many times; after some trials, however, we found out that the buffer memorization took much than the multiple access to the file, so we ended up with a queryBuffer of length 1.
Each of the threads then calls the **Reachable** function, that does the setup of the query parameters and calls the RecursiveReachable function: given a source and a destination node, it returns 1 if they are reachable, 0 otherwise.
The resolution is made through the **GRAIL** algorithm: if the destination labels are not contained in the source one, it return 0 immediately; else it verifies the reachability recurring on the source node's children (assuming each of them as next source node), until either the source and destination node are the same, or all the possibilities have been explored without success.

For the **File Version** the logic is again the same, the difference is that of course the graph has to be accessed using directly the files. This caused threads to spend too much time waiting for the file to be accessible, slowing down query resolution time. This is why in this case the **queryBuffer** mentioned before becomes useful: in particular we have noticed through trial and error that using a buffer of 20 queries and 2 threads resulted in the best performance.

5. Statistics

We present you some statistics on the **memory consumption** and **execution time** for the benchmark testsuite.
The tests are all made on the same Windows10 machine, having the following technical specifications:
Operating system: Windows 10 on 64 bit
Processor: i7 7500U CPU @ 2.70 GHz - 2.90 GHz x64
RAM: 16 GB

IN-MEMORY IMPLEMENTATION: SMALL_DENSE

File	#Nodes	#Labels	#Queries	Graph Reading Time	Creation Labels Time	Query resolving time	AVG query resolving time	Memory used
arXiv_sub_6000-1	6000	3	12000	0.556s	0.173s	3.469s	0.000012s	7.5MB
citeseer_sub_10720	10720	3	22000	0.375s	0.282s	5.398s	0.000003s	6.2MB
go_sub_6793	6793	3	15000	0.126s	0.193s	3.912s	0.000002s	3.2MB
pubmed_sub_9000_1	9000	3	18000	0.341s	0.249s	4.435s	0.000002s	5.7MB
yago_sub_6642	6642	3	12000	0.376s	0.191s	3.245s	0.000002s	5.6MB

IN-MEMORY IMPLEMENTATION: SMALL_SPARSE

File	#Nodes	#Labels	#Queries	Graph Reading Time	Creation Labels Time	Query resolving time	AVG query resolving time	Memory used
agrocyc_dag_uniq	12684	3	24000	0.176s	0.289s	5.993s	0.000002s	4.1MB
amaze_dag_uniq	3710	3	8000	0.074s	0.122s	2.106s	0.000018s	2.0MB
anthra_dag_uniq	12499	3	24000	0.179s	0.271s	5.995s	0.000368s	4.0MB
ecoo_dag_uniq	12620	3	24000	0.172s	0.270s	6.024s	0.000352s	4.1MB
human_dag_uniq	38811	3	60000	1.971s	2.093s	14.705s	0.000047s	9.9MB
kegg_dag_uniq	3617	3	7000	0.069s	0.127s	1.770s	0.000020s	2.0MB
mtbrv_dag_uniq	9602	3	18000	0.133s	0.241s	4.431s	0.000005s	3.2MB
nasa_dag_uniq	5605	3	12000	0.079s	0.163s	3.029s	0.000002s	2.6MB
vchocyc_dag_uniq	9491	3	20000	0.149s	0.235s	4.864s	0.000004s	3.4MB
xmark_dag_uniq	6080	3	12000	0.087s	0.186s	3.310s	0.000003s	2.6MB

IN-MEMORY IMPLEMENTATION: LARGE

File	#Nodes	#Labels	#Queries	Graph Reading Time	Creation Labels Time	Query resolving time	AVG query resolving time	Memory used
uniprotenc_22m.scc	1595444	3	1000000	12.289s	40.574s	290.426s	0.000034s	375MB
cit-Patents.scc	3774768	3	3000000	139.592s	96.962s	852.691s	0.000081s	1.7GB

FILE IMPLEMENTATION: SMALL_DENSE

File	#Nodes	#Labels	#Queries	Graph Reading Time	Creation Labels Time	Query resolving time	AVG query resolving time	Memory used
arXiv_sub_6000-1	6000	3	12000	10.107s	8.600s	297.471s	0.020249s	2.2MB
citeseer_sub_10720	10720	3	22000	54.362s	6.090s	23.689s	0.000571s	3.6MB
go_sub_6793	6793	3	15000	2.177s	2.267s	16.439s	0.000002s	2.3MB
pubmed_sub_9000_1	9000	3	18000	6.355s	5.780s	19.517s	0.000558s	3.0MB
yago_sub_6642	6642	3	12000	6.586s	5.846s	14.458s	0.000899s	2.2MB

FILE IMPLEMENTATION: SMALL_SPARSE

File	#Nodes	#Labels	#Queries	Graph Reading Time	Creation Labels Time	Query resolving time	AVG query resolving time	Memory used
agrocyc_dag_uniq	12684	3	24000	2.406s	3.857s	24.392s	0.000274s	4.3MB
amaze_dag_uniq	3710	3	8000	0.711s	1.024s	33.833s	0.008342s	1.8MB
anthra_dag_uniq	12499	3	24000	2.155s	3.751s	25.698s	0.000315s	4.1MB
ecoo_dag_uniq	12620	3	24000	2.426s	4.017s	25.135s	0.000344s	3.9MB
human_dag_uniq	38811	3	60000	6.786s	39.307s	70.200s	0.000731s	10.5MB
kegg_dag_uniq	3617	3	7000	0.544s	0.945s	41.670s	0.011522s	1.8MB
mtbrv_dag_uniq	9602	3	18000	1.713s	2.632s	18.815s	0.000347s	3.4MB
nasa_dag_uniq	5605	3	12000	1.152s	1.302s	12.200s	0.000363s	2.2MB
vchocyc_dag_uniq	9491	3	20000	1.643s	2.669s	20.551s	0.000338s	3.3MB
xmark_dag_uniq	6080	3	12000	1.235s	1.455s	12.676	0.000489s	2.1MB

FILE IMPLEMENTATION: LARGE

File	#Nodes	#Labels	#Queries	Graph Reading Time	Creation Labels Time	Query resolving time	AVG query resolving time	Memory used
uniprotenc_22m.scc	1595444	3	1000000	247.266s	367.259s	940.383s	0.001058s	385.7MB
cit-Patents.scc	3774768	3	3000000	3325.133s	3292.448s	2131.500s	0.001800s	854.0MB

P.S. The benchmark graph uniprotenc_100m.scc is not present in these statistics, because it seems to be corrupted, as you can see in the following screenshot:

6. Conclusions

For small Graphs the in-memory implementation is far better than the File implementation, while for large graphs the File implementation is still slower but it doesn’t require the huge amount of memory the in-memory representation does.
The long computation time needed by the File version to compute the labels can be mitigated by the fact that the labels are stored on the index file, and therefore they can be used immediately to resolve the queries.