Data Structure Contest Report - Spring 2022

Farouk Jeffar

Filippo Maria Pedini

Politecnico di Milano

High Performance Computing: Graph and Data Analytics

Guido Walter Di Donato

June 30, 2022

Introduction

The contests consisted of working with various graph representations and evaluating their performance. The goal of the challenge was to implement a graph data structure to store a graph, populate it with real graph data (in our case we used datasets provided by the Linked Data Benchmark Council (LDBC)). The initial implementation was an adjacency list that used two lists as physical implementation. The goal was to improve memory usage, population time, DFS and BFS time using any graph representation except the adjacency matrix, and it was required that the code be delivered should interface with the GraphAlgorithm class which exposes three methods (populate, get_neighbors, finished)

Code Analysis

Our Approach

Our approach consisted of analyzing the code before starting to implement or modify the pre-existing code. The analysis phase was about spotting possible weaknesses in the code and areas of improvement. This was done by understanding the decisions that were made for the pre-existing implementation and challenging them by doing research and by referring to various benchmarks and current best practices related to graph representations. By looking at benchmarks, we got an initial idea about what could be improved and what tracks should be explored in order to gain more performance. The usage of the Adjacency List lists data structures and edge storing structures were initial improvement tracks that according to literature were not as optimal and could definitely be adjusted or modified completely to improve the overall performance.

Getting neighbors

The get_neighbors method plays a particularly key role in the DFS/BFS execution. It is the function that feeds neighbor edges given a vertex. This means that improving get_neighbors would directly improve DFS/BFS performance. In the pre-existing implementation the method is to use an iteration over a list in order to get the neighbors. In the case of lists this can be quite heavy since it involves iterating over a big part of the list to retrieve neighbors and lists are certainly not the best choice for such an operation. Moreover, the get_neighbors method is expected to return an iterable object of pairs, while the default implementation that was provided used a double list method meaning that the structures did not match. This mismatch in data format impacts negatively performance since the data needs to be adapted before it is fed to the BFS/DFS function, this opens the way to the theory that if the graph representation matches the get_neighbors method, then the performance will increase since there is no overprocessing of data meaning, in theory, a reduction not only in overall computing time but also in memory usage.

Lists and vectors

The first clear thing that struck our minds whilst analyzing the code was the use of two lists as representation for the Adjacency List, we documented thoroughly this choice of implementation and quickly realized that even if it was fully functional it was inefficient. The inefficiency was caused by using two lists which increases memory usage, and also by the fact the lists are not the most efficient data structure in the C++ programming language. So, in theory, if tuples or pairs were used, it would be possible to use one list and thus save memory. Moreover, our past programming knowledge alerted us that vectors might be faster than lists, and by doing some research we found out this was indeed the case. According to Baptiste Wicht from dzone.com:

- std::vector is insanely faster than std::list to find an element
- std::vector always performs faster than std::list with very small data
- std::vector is always faster to push elements at the back than std::list
- std::list handles large elements very well, especially for sorting or inserting in the front

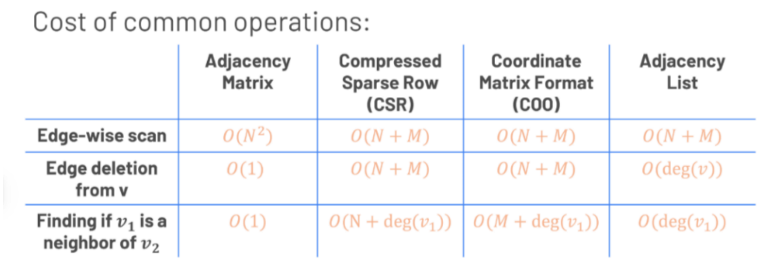
https://dzone.com/articles/c-benchmark-—stdvector-vs#:~:text=std%3A%3Avector%20is%20insanely,or%20ins

Therefore, the conclusions we got from our code analysis are that lists should be replaced and that edges should be stored in data structure that facilitates the integration with the GraphAlgorithm class.

Implementation

Adjacency List

In our implementation we decided to stick to the adjacency list graph representation, not only because we learned from the theoretical lectures of this Passion in Action Course that it was overall the most efficient one, but also because our researches on the CSR representation showed that although comparable, it has problems with insertion of new nodes/edges after the representation has been finalized.



Cost of common operations:

	Adjacency Matrix	Compressed Sparse Row (CSR)	Coordinate Matrix Format (COO)	Adjacency List
Edge-wise scan	$O(N^2)$	$O(N+M)$	$O(N+M)$	$O(N+M)$
Edge deletion from v	$O(1)$	$O(N+M)$	$O(N+M)$	$O(\deg(v))$
Finding if v_1 is a neighbor of v_2	$O(1)$	$O(\mathrm{N}+\deg(v_1))$	$O(M+\deg(v_1))$	$O(\deg(v_1))$

Figure 1: Graph representations comparison

According to the image above, the adjacency list algorithm is a better fit in our case.

Vectors and Pairs

The first modification was to remove two lists and use one list of tuples. This implementation would decrease the memory usage and population time, but this would drastically worsen DFS/BFS because it would need to iterate over all the tuples to find each individual neighbor which result in a complexity of O(n^2). This means that tuples are not the best solution because we cannot access neighbors directly. This is when we had the idea of using a list of list of pairs, this solution would help us not only reduce memory usage and population but also the BFS/DFS time since by navigating to an index on the list we would get directly the list of neighbors by iterating over the list. However, this implementation of lists of lists of pairs only improved memory and population time but the BFS/DFS time increased compared to the default implementation. Since we already discovered that vectors are faster than lists, we switched to a vector of vectors of pairs and in the get neighbors function we returned the full vector corresponding to the current vertex directly without having to iterate, being C++ optimized for moving vectors as opposed to lists and because get_neighbors expects an iterable object of pairs which matches perfectly with our graph representation which is vector of vectors of pairs. After running this implementation, the results were very ahead compared to the other methods we tried for the evaluation criteria (Population, Memory, BFS, and DFS) and we will showcase them in detail in the next section.

Sorting

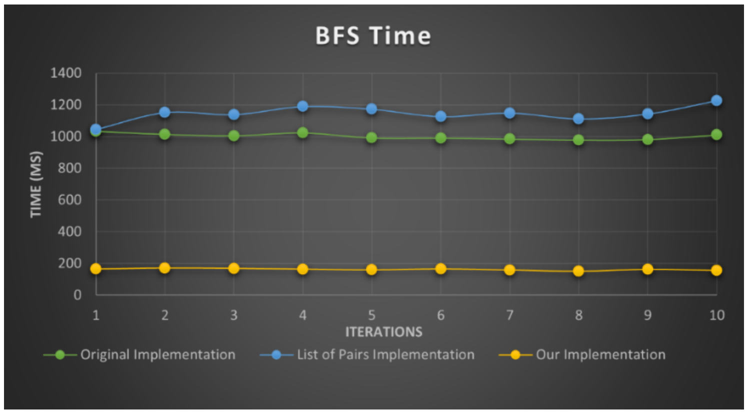
To try to improve the results even more, we experimented with sorting. Given our code, the most sensible sorting location would be after the population since sorting before has no effect at all. So, after experimenting with sorting we found the same conclusion across all methods we experimented (our main implementation, default implementation, and list of lists of pairs). The findings for sorting were that population increased drastically, memory usage remained on par with the un-sorted version, but the DFS/BFS times marginally decreased. We saw that the population time increase was too big compared to the time gained in DFS/BFS, thus we decided to keep the un-sorted version of a vector of vectors of pairs as representation of an adjacency list as our implementation for the challenge.

Results

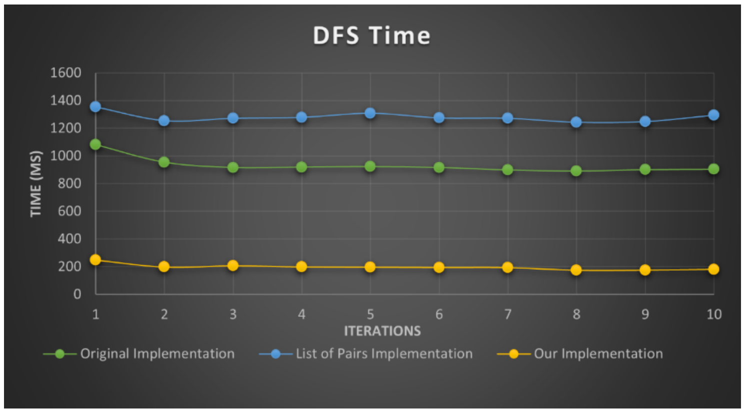
In this chapter we show a very graphical representation of the results we obtained with our implementation of the adjacency list, the plots compare the results of all the meaningful metrics of the benchmark, obtained with the three implementations (Original implementation, List of pairs, our implementation). In particular the plots refer to the Wiki-Talk Graph and the Cit-Patents Graph, Dota-league Graph was not included due to the lack of memory.

Results key points:

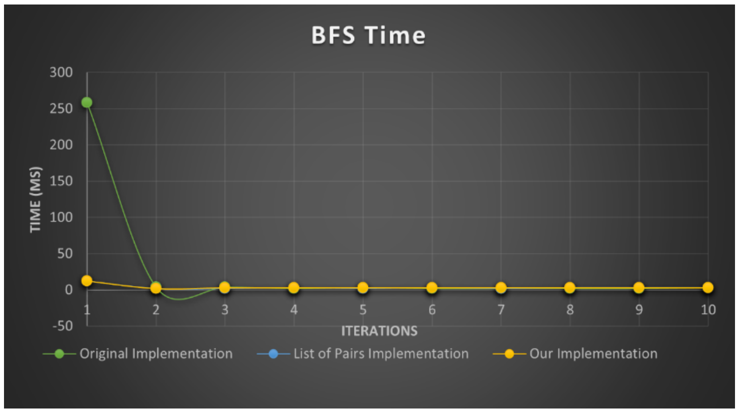
- 2.4x less time in population on average
- 91% less memory usage on average
- 6.2x less time in BFS on average
- 4.7xlesstimeinDFS on average



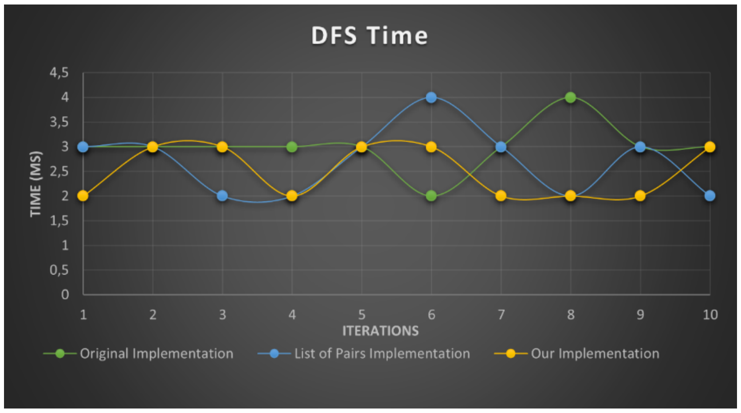
BFS Time, WikiTalk Graph



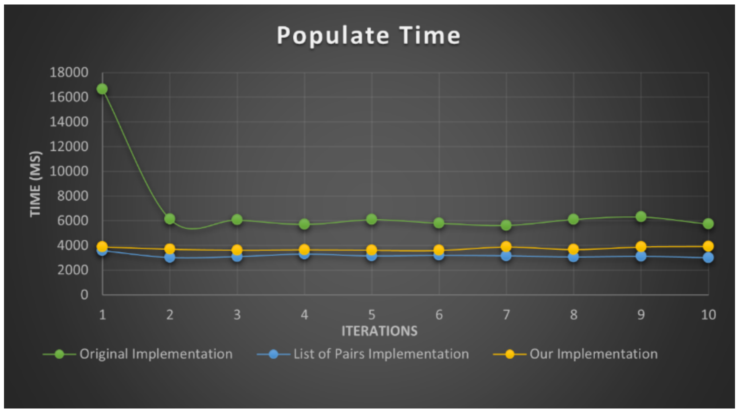
DFS Time, WikiTalk Graph



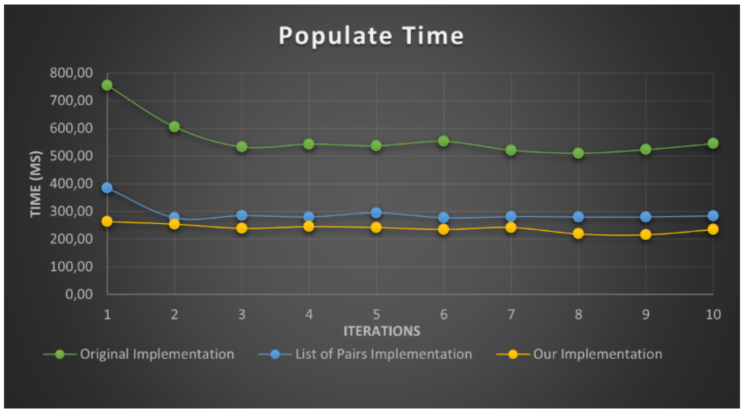
BFS Time, CitPatent Graph



DFS Time, CitPatent Graph



Populate Time, CitPatent Graph



Populate Time, WikiTalk Graph

Conclusion

In our educational path, we came across graphs in all our major courses as computer scientists. Nevertheless, we never got the chance to really work with graphs practically. Furthermore, by implementing and experimenting with them in order to find an efficient representation, we are confident in saying that our graph skills moved from the theoretical aspect to a practical. We noticed this particularly towards the ending phases of the challenge, where we would effectively analyze a representation or an improvement idea before implementing since we were accustomed to how graph works and what parameters impact their performance.