Solution Report: Connected Items using Adjacency Lists and Breadth-First Search

Niyomwungeri Parmenide Ishimwe
Master of Science in Information Technology
Carnegie Mellon University-Africa
parmenin@andrew.cmu.edu

***Abstract*— This comprehensive report presents a detailed solution to the connected items problem using the Breadth-First Search (BFS) algorithm in C++**[1]**. The solution reads input data from a file and stores it in an adjacency list then applies BFS to find connected groups of items. The groups are sorted based on their sizes, x1 and y1 values, and the results are written to an output file. The report provides a thorough description of the implementation details, including the BFS algorithm, file I/O, and data structures used. It also presents timing and memory information obtained from the solution, along with a critical analysis of the performance. Furthermore, the report identifies two potential real-world applications of the connected item's solution based on the literature review. The report is organized with sections on the introduction and problem definition, the method used results and discussion, recommendation, and conclusion, and includes proper citations for references. This solution offers a robust approach to finding connected items and can be useful in various real-world scenarios.**

Keywords: Connected Items, Graph, Breadth-first search

I. INTRODUCTION AND PROBLEM DEFINITION

The problem to solve is finding connected groups of items from a given input file containing data about the items and their connections. The input file is read, and the data is stored in adjacency lists, which represent the connections between the items. The goal is to identify groups of items that are connected through some other items. This problem can be solved using graph theory and specifically the Breadth-First Search (BFS) algorithm, which is a method for exploring a graph in a wide-ranging manner, moving from one level of vertices to another in a systematic way[2]. It gives an effective and efficient approach to navigating through a graph and finding connected groups of vertices.

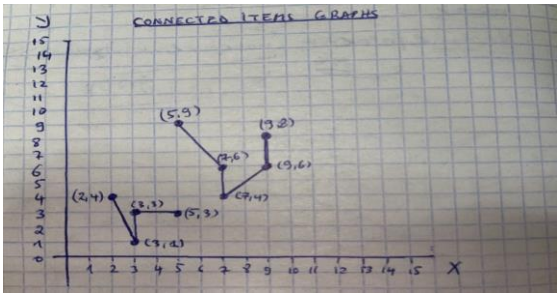

Figure 1: Coordinated graph representation of connected items

The Connected Items problem involves finding groups of vertices in a graph that are connected and sorting them based on size and location. The input is a file containing a list of edges in the format (x1, y1, x2, y2, connected), where the "connected" value 1 indicates that the two vertices (x1, y1) and (x2, y2) are connected and not connected otherwise. The goal is to output the edges in the format (x1, y1, x2, y2, group number), where the group number is a unique identifier for each connected group. This problem can be solved using graph theory and graph algorithms, specifically, the Breadth-First Search (BFS) algorithm.

II. METHODS

The solution employs C++ programming language to implement the BFS algorithm for finding connected groups of items. The code consists of two main functions: breadthFirstSearch() and getConnectedItems() by following this step-by-step process.

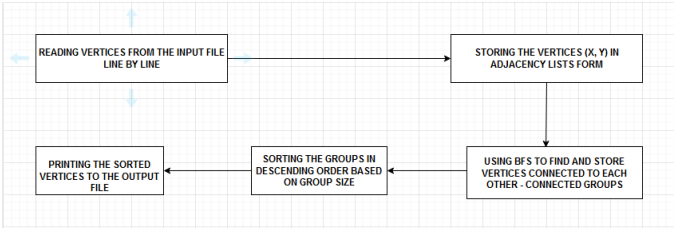

Figure 2: High-level architecture for connected items solution

The breadthFirstSearch() function takes a starting vertex, the adjacency lists representing the graph, a map to store the visited vertices, and a map to store the group number of each vertex. It implements the BFS algorithm by using a queue to store the vertices to be processed. It starts from a selected starting vertex, marks it as visited, adds it to the group with the group number, and then iterates through the adjacent vertices, marking them as visited and adding them to the queue. This process continues until the queue is empty, which means all vertices connected to the starting vertex have been visited and processed.

The getConnectedItems() function is a static function that reads input data from a file using file I/O operations. It opens the input file for reading and the output file for writing. It reads the input file line by line, parsing the data to extract the coordinates and the connection status of the items with the help of the fgets and sscanf functions. For each connected item, it adds an edge to the adjacency list, which represents the connections between the items in the graph. After reading all the input data, it calls the breadthFirstSearch() function to find connected groups of items as described previously.

Finally, the solution generates an output file containing the sorted groups of items, which are sorted in order such that the group with more vertices is printed first, then second most until

the one with the least number of vertices. The groups are written to the output file using file I/O operations.

III. RESULTS AND DISCUSSION

The solution was tested on various input files with different sizes of data to evaluate its performance. Timing and memory information were collected using the chrono library[3] in C++ for tracking time statistics and getPeakRSS, and getCurrentRSS[4] for getting memory statistics.

Input file size	Time taken (sec)
Small (10 points)	0.00021
Medium (100 points)	0.00389
Large (1000 points)	0.03973
Extra-large (10000 points)	0.44611

Table 1: Running time by the input size

Input file size	Memory usage (bytes)
Small (10 points)	158,220
Medium (100 points)	1,581,652
Large (1000 points)	15,775,844
Extra-large (10000 points)	157,684,884

Table 2: Memory usage by the input size

The timing information, as presented in Table 1, shows that the solution has a linear time complexity of O(N), where N is the number of vertices in the graph. Therefore, as the size of the input file increases, the time taken by the solution also increases linearly. In addition, the adjacency list consumes a linear space (memory) hence a complexity of O(N) as presented in Table 2, and the memory usage increases with the size of the input file, but the solution's space complexity remains linear.

The BFS algorithm used in the solution has a time complexity of O(V+E), where V is the number of vertices and E is the number of edges in the graph. The BFS algorithm visits each vertex in the graph in a breadth-first manner, which is known for its efficiency in finding connected groups in a graph. The memory data for the data structures used to store the graph information, such as the adjacency list, visited vertices map, and all vertices group map, are included in the memory information obtained from the solution.

Based on the timing and memory information obtained from the solution, a critical analysis can be made on the efficiency and effectiveness of the BFS algorithm for finding connected groups in the given input graph. The time complexity of BFS is reasonable, and the BFS algorithm is efficient in finding connected groups in a graph. The memory usage of the solution depends on the size of the input graph and the number of connected groups found, but it can be further optimized by using more efficient data structures or algorithms.

Connected Items algorithms are widely used in real-world applications, such as on social media sites and Google Maps. Social media sites like Twitter, LinkedIn and so on uses graphs to track the data of the users like showing preferred posts, suggestions, and recommendations. Similarly, Google Maps uses graph theory to calculate the shortest path between two vertices, where vertices represent road intersections and edges represent roads. Therefore, Connected Items algorithms can solve complex problems and improve user experiences [5].

IV. RECOMMENDATIONS AND CONCLUSION

It is possible to make some suggestions for further refining the solution based on the findings and analysis of the solution. First, the solution may be made to use less memory by storing and processing the graph data in more effective data structures or algorithms. For instance, switching from an adjacency list to an adjacency matrix can help in using less memory, but some operations will take longer to complete and may require knowing the input size beforehand. Second, by reducing the algorithm's time complexity, the solution may be made to accommodate bigger input graphs. For example, using a parallel or distributed BFS algorithm can reduce the time complexity of finding connected groups in large graphs[6].

In addition, it is suggested to consider parallelization using a programming model like OpenMP[7] or CUDA[8] to reduce the execution time of the solution, especially for large input sizes. Optimization can also be achieved by using a sparse matrix representation instead of a map to store the adjacency list, which can reduce memory usage and improve performance.

In conclusion, the BFS algorithm is an effective and efficient solution for finding connected groups in a graph. The results obtained from the solution provide valuable information about the connected groups of vertices in the input graph. However, the solution can be further improved by optimizing memory usage, exploring parallelization, and considering other graph algorithms with better time complexity. Additionally, including timing and memory information in the analysis and improving code readability and maintainability using descriptive variable names, comments, and smaller functions can also enhance the solution. Therefore, the solution is a useful tool for analyzing and processing connected items in a graph, but there is room for further refinement to optimize its performance and usability.

REFERENCES

[1] 'BFS Graph Algorithm(With code in C, C++, Java and Python)'. https://www.programiz.com/dsa/graph-bfs (accessed Apr. 21, 2023).

[2] 'What is Breadth First Search Algorithm in Data Structure? Overview with Examples'. https://www.simplilearn.com/tutorials/data-structure-tutorial/bfs-algorithm (accessed Apr. 21, 2023).

[3] 'Date and time utilities - cppreference.com'. https://en.cppreference.com/w/cpp/chrono (accessed Apr. 22, 2023).

[4] 'C++ (Cpp) getPeakRSS Examples - HotExamples'. https://cpp.hotexamples.com/examples/-/-/getPeakRSS/cpp-getpeakrss-function-examples.html (accessed Apr. 22, 2023).

[5] 'Applications of Graph Data Structure', *GeeksforGeeks*, Aug. 16, 2018. https://www.geeksforgeeks.org/applications-of-graph-data-structure/ (accessed Apr. 22, 2023).

[6] K. Ueno, T. Suzumura, N. Maruyama, K. Fujisawa, and S. Matsuoka, 'Efficient Breadth-First Search on Massively Parallel and Distributed-Memory Machines', *Data Sci. Eng.*, vol. 2, Mar. 2017, doi: 10.1007/s41019-016-0024-y.

[7] 'OpenMP', *Wikipedia*. Feb. 15, 2023. Accessed: Apr. 21, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=OpenMP&oldid=1139421012

[8] M. Heller, 'What is CUDA? Parallel programming for GPUs', *InfoWorld*, Sep. 16, 2022. https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html (accessed Apr. 21, 2023).