Practical Project
Graph Implementation in C

Enrique George Rodrigues 28602

Instituto Politécnico do Cávado e do Ave
Bachelor of Engineering (BEng) in Computer Systems Engineering
Advanced Data Structures Class
24 MAY 2024

Abstract. This document describes my solution to a computational problem of medium complexity, which includes the implementation of a graph in C, applying advanced concepts of graph theory and programming to solve the problem. The aim was to develop a solution capable of calculating the maximum possible sum of integers from a matrix of arbitrary dimensions, taking into account specific connection rules between the elements. For this implementation, a combination of a hash table and linked lists was used. These data structures were chosen to ensure both memory and performance efficiency.

Key-words: Programming, Graphs, Hash Tables, Linked Lists, C Language, DFS, Dijkstra's Algorithm

1 Introduction

This document describes the work done for the second practical assignment in the Advanced Data Structures class, part of my Degree in Computer Systems Engineering at the Polytechnic Institute of Cávado and Ave. This second practical assignment focuses on the implementation and manipulation of graphs, applying advanced concepts of graph theory and C programming.

The detailed specifications of the work can be found in the annex entitled "Enunciado do Trabalho Prático". The source code is available on Github, see https://github.com/Basiiii/Maximal-Graph-Sum-C.

The implementation of efficient data structures is fundamental to solving complex computational problems, especially in the context of graph theory. This document describes the solution developed for a problem that requires the construction and manipulation of a graph using the C language. The graph structure was built using an advanced combination of a hash table for the vertices with a linked list to control collisions and linked lists for the adjacent edges. This approach was chosen not only to optimise performance, but also to ensure efficient use of memory.

Within the graph structure is the hash table, where each entry represents a vertex of the graph. In the event of a collision, i.e. when multiple vertices are hashed to the same position in the table, a linked list is used to resolve conflicts.

Each vertex is represented by a structure containing a unique identifier and a linked list of adjacent edges. The edges, in turn, are represented by a structure that stores the destination vertex, the weight of the edge and a pointer to the next edge in the linked list.

In addition to the data structure, this work also implemented fundamental algorithms for manipulating the graph. Algorithms such as Dijkstra were used to find the shortest path between vertices, and a depth-first search (DFS) algorithm was used to explore the vertices and edges of the graph and calculate the path with the highest weight. These algorithms are essential for solving the given problem of connectivity within the graph.

Later in this document, I will explain the structure of the implemented code (Sect. 3), highlighting the main design decisions and the functionalities implemented. This approach aims to provide a robust and efficient solution to the proposed problem, using advanced concepts of graph theory and C programming.

2 Programming Style and Patterns

To ensure consistency, readability and maintenance of the code base, this project follows established programming standards. These standards are aligned with the Google Style Guide for C++ [1], adapted for C, promoting uniformity throughout the project.

2.1 Naming Conventions

Adopting a clear naming convention improves the clarity of the code. The following conventions were used throughout the project:

- **Functions**: `FunctionsLikeThis()`
- **Variables**: `variablesLikeThis`
- **Constants**: `CONSTANTS_LIKE_THIS`
- **Enums**: `EnumsLikeThis`
- **Structs**: `StructsLikeThis`

The use of completely capitalised names is avoided, except for constants, in order to clearly distinguish them.

2.2 Formatting Rules

Consistent formatting is essential for readability:

- **Indentation**: 2 spaces per indentation level, avoiding tabs.
- **Line Length**: Lines limited to a maximum of 80 characters.
- **Keys**: Opening braces are placed on the same line as the control statement.
- **Comments**: Descriptive comments are used to explain complex sections of code, following a consistent style.

2.3 Programming Practices

Following best practices ensures code quality:

- Use clear names for variables and functions, making the code self-explanatory.
- Write clear and concise code, avoiding unnecessary complexity.
- You follow the Google Style Guide for C++ [1] for general formatting and styling, ensuring compliance with industry standards.

2.4 Version Control

Git is used for version control, facilitating effective management of the code base:

- A new branch is created for each feature or error correction.
- Branches are created from the main branch for development, with descriptive names that reflect the functionality or error addressed.
- Commits are avoided directly on the main branch, ensuring stability.
- Functionality branches are regularly merged back into the main branch after thorough testing.

2.5 Documentation

Doxygen is used to document the code, automatically generating the API documentation:

- The Doxygen syntax for comments is followed, documenting functions, variables and blocks of code.
- The documentation includes short descriptions, parameter descriptions, return value descriptions and usage examples.
- Doxygen tags such as `@brief`, `@param`, `@return` and `@example` are used to structure comments correctly.
- Defines the author in file header comments for new files or significant sections of code, with modifications documented using the `@section Modifications` tag.

By following these programming standards and practices, the project maintains a high-quality code base, ensuring consistency and making it easier to understand, contribute to and maintain.

3 Graph Structure

In this section, I will describe in detail the structure of the implemented graph, addressing the design choices and justifying the decisions based on efficiency and scalability. The implementation was done in C, using a hash table to manage the vertices and linked lists for the edges, optimising both performance and memory usage.

3.1 Graph

Before discussing vertices and edges in depth, it is important to understand the basic structure of the implemented graph. The *Graph* structure is defined as follows:

```
typedef struct Graph {
        unsigned int numVertices;
        unsigned int hashSize;
        Vertex** vertices;
} Graph;
```

Code Snippet 1.1. Graph Structure

This structure stores the current number of vertices in the graph *(numVertices)*, the size of the hash table *(hashSize)*, and a pointer to the hash table containing the vertices *(vertices)*. The hash table facilitates quick access to the vertices, while the modular structure allows the graph to adjust dynamically to the growth in the number of vertices.

3.2 Vertices

The vertices of the graph are stored in a hash table. The choice of a hash table for the vertices was made with the aim of minimising access and search time. The hash table provides an average access time of $O(1)$, allowing for fast insertions, removals and searches, which is crucial for the graph's performance.

When several vertices are assigned to the same index in a hash table, this represents a collision. To manage these situations, a method known as separate chaining has been used, where each index in the hash table points to a linked list containing the vertices that have the same hash function value. This ensures that data access remains efficient even in the presence of collisions, keeping the overall structure of the hash table intact and allowing for quick insertions, removals and searches within the linked lists associated with each index.

```
typedef struct Vertex {
        unsigned int id;
        Edge* edges;
        struct Vertex* next;
} Vertex;
```

Code Snippet 1.2. Vertex Structure

3.3 Edges

The edges are stored in linked lists associated with each vertex. This approach was chosen because of the ease of insertion and removal. The use of linked lists allows for efficient edge insertion and removal operations. In dynamic graphs, where the links between vertices can change frequently, this flexibility is essential.

Compared to an adjacency matrix, linked lists use a space proportional to the number of edges ($O(n)$), making them more efficient for sparse graphs.

Linked lists also allow each edge to store a weight, which is essential for algorithms such as Dijkstra's, which need information about the weight of the edges.

Each edge is represented by a structure that includes the destination vertex, the weight of the edge and a pointer to the next edge in the linked list. This representation simplifies the implementation of graph algorithms and the manipulation of connections between vertices.

```
typedef struct Edge {
        unsigned int dest;
        unsigned int weight;
        struct Edge* next;
} Edge;
```

Code Snippet 1.3. Edge Structure

3.4 Hash Function

The hash function implemented applies a hashing technique designed to achieve an excellent statistical distribution. It ensures that each input bit influences approximately 50% of the output bits, aiming to minimise collisions by guaranteeing unique outputs for distinct inputs. This is achieved through efficient integer arithmetic and bitwise operations. The algorithm incorporates a carefully selected "magic number" ($0x45d9f3b$), chosen after extensive testing for performance optimisation.

This number improves the quality of the algorithm, bringing it closer to AES encryption hashes, albeit with potential marginal speed advantages. Although normalising the hash value to fit the size of the hash table can introduce collisions, these can be effectively mitigated with sufficiently large hash tables, or by using the separate chaining technique. Credit for this algorithm goes to *Thomas Mueller*, as discussed on Stack Overflow [2].

```
uint32_t Hash(uint32_t id, uint32_t hashSize) {
        // Inicialize position as given id
        uint32_t index = id;

        // Apply the hash function
        index = ((index >> 16) ^ index) * 0x45d9f3b;
        index = ((index >> 16) ^ index) * 0x45d9f3b;
        index = (index >> 16) ^ index;

        // Normalize the hash value to fit in the hash table
        index %= hashSize;

        return index;
}
```

Code Snippet 1.4. Hash Function

3.5 Big O Notation

The main operations in the graph are designed to be efficient, with the following Big O notation:

- **Vertex Insertion**: $O(1)$ in the average case, due to direct access in the hash table.
- **Vertex Removal**: $O(1)$ in the average case, using the hash table.
- **Edge Insertion**: $O(1)$ to add an edge to the beginning of the linked list of adjacent edges.
- **Remove Edge**: $O(1)$ in the best case, or $O(n)$ where n is the number of adjacent edges that need to be traversed to remove an edge.
- **Vertex Search by ID**: $O(1)$ in the average case, thanks to direct access to the hash table.
- **Edge Search**: $O(1)$ in the best case, or $O(n)$ where n is the number of adjacent edges that need to be traversed to search for an edge.

This structure and Big O notation are ideal for supporting efficient operations on variable-sized graphs, while maintaining the simplicity and scalability of the implementation.

4 Graph Preservation

This project includes functionalities for importing, exporting, saving and loading graphs, ensuring flexibility and efficiency in the manipulation of graph data.

4.1 Importing a Graph from a CSV File

Graphs are imported from CSV files with a specific format, where each line of the file represents a vertex and its edges. The format is as follows:

```
1;958;2
2;29;8
3;849;9;219;6;648;2
4;289;5;27;4;777;4;622;4
5;259;5;597;1
6;269;7
```

Each line contains the identifier of a vertex followed by pairs of values representing the destination vertices and the weights of the edges. For example:

```
5;259;5;597;1
```

This line indicates that vertex **5** is connected to vertex **259** with weight **5** and to vertex **597** with weight **1**.

To process the CSV file, we use a parser that analyses the entire file using the functions `strtok_s()` and `atoi()`. This method is highly efficient, allowing a random graph with a million vertices and a random number of edges to be analysed and loaded into memory in less than 2 seconds.

4.2 Graph export to CSV file

Exporting graphs to CSV files follows the format described above, making it easier to read and manipulate the data in other tools or for future uploads to the system.

4.3 Save and Load a Graph

In addition to importing and exporting in CSV format, the project also supports saving and loading graphs in and from binary files. This method is used to preserve the state of the graph efficiently and quickly, allowing for faster loading compared to reading CSV files.

Saving a graph is done by writing the data directly to a binary file. This approach ensures that all the details of the graph, including the vertices and edges with their respective weights, are saved in a compact form.

Loading the graph from a binary file involves reading the data directly into memory. This operation is optimised to ensure that large graphs can be loaded quickly while preserving all the structure and associated data.

These import, export, save and load functionalities provide flexible and efficient management of graphs, allowing easy manipulation and preservation of data.

5 Algorithms

The algorithms section presents a series of implementations for finding and analysing paths in graphs. The aim of these algorithms is to find all possible paths between specific vertices, as well as to calculate relevant information about these paths, such as the total weight of the edges. It also includes algorithms for calculating the shortest path *(minimum shortest path)* and the longest path *(maximum longest path)* between vertices.

5.1 Find All Possible Paths

This algorithm, implemented in the file `search.c`, uses a depth-first search (DFS) approach to find all possible paths from a source vertex to a destination vertex in a graph. During execution, it keeps a linked list of paths found and calculates the total weight of each path.

```
typedef struct PathNode {
        unsigned int* vertices;
        unsigned int* weights;
        unsigned int length;
        struct PathNode* next;
} PathNode;
```

Code Snippet 1.5. Path Data Structure Definition

The `FindAllPaths` function is responsible for coordinating the search, initialising the necessary parameters and calling `DepthFirstSearch` to start the depth search from the source vertex to the destination vertex. Each path found is added to the list of paths using the `AddPath` function. After execution, the `PrintPaths` function is used to print all the paths found, along with the total weight of each path.

```
PathNode* FindAllPaths(const Graph* graph, unsigned int src,
    unsigned int dest, unsigned int* numPaths) {
    *numPaths = 0;
    unsigned int pathCapacity = 10;
    PathNode* paths = NULL;
    bool* visited = (bool*)calloc(graph->hashSize,
        sizeof(bool));
    unsigned int* pathVertices =
    (unsigned int*)malloc(graph->hashSize *
        sizeof(unsigned int));
    unsigned int* pathWeights =
    (unsigned int*)malloc(graph->hashSize *
        sizeof(unsigned int));

    DFSContext context = { .graph = graph,
            .pathVertices = pathVertices,
            .pathWeights = pathWeights,
            .visited = visited,
            .pathIndex = 0,
            .paths = &paths,
            .numPaths = numPaths,
            .pathCapacity = &pathCapacity };

    bool success = DepthFirstSearch(&context, src, dest);

    if (!success) {
            FreePaths(paths);
            paths = NULL;
            *numPaths = 0;
    }

    free(visited);
    free(pathVertices);
    free(pathWeights);
```

```
31
32                  return paths;
33          }
```

Code Snippet 1.6. Function to Find All Paths

The algorithm uses a linked list of paths (`PathNode`) to store each path found. The `DFSContext` data structure is used to maintain the state of the depth-first search, including vertices visited, the current path index and the list of paths found.

5.2 Finding the Shortest Path with Dijkstra

This algorithm, implemented in the file `dijkstra-min.c`, uses Dijkstra's algorithm to find the shortest path from a source vertex to a destination vertex in a weighted graph. During execution, it maintains a minimal heap structure to manage the vertices to be explored and calculates the total weight of the shortest path.

```
1           typedef struct HeapNode {
2                   unsigned int vertex;
3                   unsigned int weight;
4           } HeapNode;
5
6           typedef struct MinHeap {
7                   HeapNode* nodes;
8                   unsigned int size;
9                   unsigned int capacity;
10          } MinHeap;
```

Code Snippet 1.7. Heap Data Structure Definitions

The `DijkstraShortestPath` function is responsible for coordinating the execution of the algorithm, initialising the necessary structures and calling auxiliary functions to insert vertices into the heap and extract the vertex with the lowest weight. After execution, the `PrintShortestPath` function is used to print the shortest path found, along with the total weight of the path.

The algorithm uses a minimal heap structure (`MinHeap`) to manage the vertices to be explored. The `HeapNode` structure is used to represent a node in the heap, with the vertex index (`vertex`) and the associated weight (`weight`). The `MinHeap` structure maintains an array of `HeapNode` elements and tracks the current number of elements in the heap (`size`) and the maximum capacity of the heap (`capacity`).

The `DijkstraShortestPath` function uses Dijkstra to calculate the shortest path from a source vertex (`src`) to a destination vertex (`dest`) in a weighted graph (`graph`). After execution, the shortest path is reconstructed using the `prev` array and stored in `path`, with its length stored in `pathLength`.

Explanation:

- **Data Structures:** `HeapNode` represents a node in the heap with a vertex and a weight, while `MinHeap` is a minimal heap structure for Dijkstra.
- **Auxiliary Functions:** `CreateMinHeap`, `InsertNode`, `ExtractMin` are auxiliary functions for manipulating the heap.
- **Dijkstra's algorithm:** The `DijkstraShortestPath` function implements Dijkstra's algorithm for calculating the shortest path in a weighted graph, using a minimum heap for optimisation.
- **Memory Release:** Dynamically allocated memory (`dist`, `visited`, `prev`, `minHeap->nodes`, `minHeap`) is freed at the end of the function to avoid memory leaks.

This version of the `DijkstraShortestPath` function provides robust error handling and clear documentation, ensuring that memory allocation failures are properly handled and that the function is reliable in a variety of scenarios.

5.3 Finding the Longest Path with Dijkstra

This algorithm, implemented in the file `dijkstra-max.c`, uses Dijkstra's algorithm to find the longest path from a source vertex to a destination vertex in a graph. During execution, it maintains a maximum heap structure to manage the vertices to be explored and calculates the total weight of the longest path.

```
1         typedef struct HeapNode {
2                 unsigned int vertex;
3                 unsigned int weight;
4         } HeapNode;
5
6         typedef struct MaxHeap {
7                 HeapNode* nodes;
8                 unsigned int size;
9                 unsigned int capacity;
10        } MaxHeap;
```

Code Snippet 1.8. Heap Structure Definition for Max Heap

The `DijkstraMaxPath` function is responsible for coordinating the execution of the algorithm, initialising the necessary structures and calling auxiliary functions to insert vertices into the heap and extract the vertex with the greatest weight. After execution, the `PrintLongestPath` function is used to print the longest path found, along with the total weight of the path.

The algorithm uses a maximum heap structure (`MaxHeap`) to manage the vertices to be explored. The `HeapNode` structure is used to represent a node in the heap, with the vertex index (`vertex`) and the associated weight (`weight`). The `MaxHeap` structure maintains an array of `HeapNode` elements and tracks the current number of elements in the heap (`size`) and the maximum capacity of the heap (`capacity`).

Explanation:

- **Data Structures:** `HeapNode` represents a node in the heap with a vertex and a weight, while `MaxHeap` is a maximum heap structure for Dijkstra.
- **Auxiliary Functions:** `CreateMaxHeap`, `InsertNodeMaxHeap`, `ExtractMax` are auxiliary functions for manipulating the maximum heap.
- **Dijkstra's algorithm:** The `DijkstraMaxPath` function implements Dijkstra's algorithm for calculating the longest path in a graph, using a maximum heap for optimization.
- **Memory Release:** Dynamically allocated memory (`dist`, `visited`, `prev`, `maxHeap->nodes`, `maxHeap`) is freed at the end of the function to avoid memory leaks.

This version of the `DijkstraMaxPath` function provides robust error handling and clear documentation, ensuring that memory allocation failures are properly handled and that the function is reliable.

6 Conclusion

In this work, I have successfully developed and implemented a solution to a challenging computational problem. This solution involves creating a graph in C, using advanced concepts of graph theory and programming to achieve the desired result. The main objective of this practical work was to develop a method capable of calculating the maximum possible sum of integers from a matrix of arbitrary dimensions, respecting specific rules governing the connections between the elements.

To do this, I used a combination of hash tables and linked lists. These data structures were chosen because of their ability to guarantee efficiency in both memory usage and performance. The hash table facilitated fast search operations, while linked lists allowed efficient traversal and manipulation of the connections between elements in the graph.

Through this implementation, I demonstrated my knowledge of the underlying theories and methodologies required to solve complex computational problems. The approach adopted not only achieves the desired functionality, but also optimizes the use of resources, making it suitable for real-world applications where efficiency is key.

In summary, this project has provided valuable insights into the practical application of graph theory and data structures, further improving my proficiency in advanced programming techniques. The solution developed here is a testament to my ability to tackle and solve demanding computational challenges effectively.

Acknowledgments I would like to express my sincere thanks to Professor Luís Ferreira for the guidance, support, and valuable feedback throughout the development of this project, which was fundamental in its realization.

References

1. Google Inc.: Google C++ Style Guide. `https://google.github.io/styleguide/cppguide.html` (2024), seen: 2024-05-02
2. Mueller, T.: What integer hash function are good that accepts an integer hash key? (2009), URL `https://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key`, seen: 2024-05-14

All links were last followed on May 24, 2024.