POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTING AND TELECOMMUNICATION

Institute of Computing Science

Report

GRAPH ALGORITHMS

Adam Korba, 151962

Teacher

PhD. Grzegorz Pawlak

POZNAŃ 2022

Contents

Chapter 1

Introduction

Graphs are mathematical data structures that consist of vertex set - $|V|$ and edge set - $|E|$. They are used as a mathematical abstraction of more complex structures and are useful in solving many real-world problems. This report focuses on the properties of digital representations of these structures and implementing them in the context of simple algorithms. Four different types of graph representation considered are:

- Neighborhood matrix (vertex matrix)
- Incident matrix (egde matrix)
- Edge list
- List of incidents

The time needed to check the existence of edges was measured, and the report compares performances of these graph representations, time and space complexities.

In addition, a topological sort algorithm (topsort) was implemented. Different possibilities for choosing suitable graph representations are discussed.

Chapter 2

Exercises

2.1 Random graph generation

2.1.1 undirected graph

To get test objects it was required to generate random graphs with n vertices. My approach was to generate a set of vertices and a set of all possible edges. Then vector holding these edges is randomly shuffled and at the end, we take only the first 60% of edges as it is our desired saturation. Then a graph is saved to a text file.

2.1.2 directed acyclic graph

Topological sort is an algorithm that only works on directed acyclic graphs (DAGs in short). My approach to generate randomly these types of graphs was to first create a set of vertices and then, in the loop generate arcs until the desired amount is created (saturation is 0.3 so we need 30% of all possible arcs. Arcs are created by randomly choosing two vertices out of the set, it is checked if a given arc exists and if it is the case the new one is discarded. Additionally, every time an arc is added algorithm traverses the graph looking for cycles. If none are found arc can be kept. The resulting set of vertices and arcs is saved to a text file.

2.2 Exercise 1 - graph Representations

2.2.1 neighborhood matrix (vertex matrix)

This graph representation works on the adjacency matrix (A). We store the $|V|$ x $|V|$ matrix. Every possible entry $A_{i,j}$ is a number, where 1 means that there exists an edge between i and j and 0 if there is no such edge. There is also a possibility to represent directed graphs where -1 means that the arc ends at j and 1 means that it starts at i. Space complexity is $\mathcal{O}(|V|^2)$ so for graphs with small saturation this representation wastes memory for storing zeros.

2.2.2 incident matrix (egde matrix)

This representation works on an incidence matrix that stores one edge per column. It is required to look at the entire column to decide which vertices some edge connects. It means that if in the n-th column on the i-th and j-th row there is a 1 there is an edge between i and j. Once again it is not difficult to represent directed graphs with this method by writing -1. This representation has really bad space complexity $\mathcal{O}(|V| * |E|)$, so it is almost unusable for graphs with a large number of edges.

2.2.3 edge list

This representation is similar to a mathematical graph because it contains a set of vertices and a set of edges. Set of edges contains `std::pair<int, int>` objects, where the first number is starting vertex and the second is the ending vertex (in an undirected graph the order of these numbers doesn't matter). It has space complexity of $\mathcal{O}(|V| + |E|)$

2.2.4 list of incidents

This structure consists of linked lists for every vertex v where the contents of these lists are vertices adjacent to v, these lists are also called adjacency lists. This implementation has a space complexity of $\mathcal{O}(|V| + |E|)$

2.2.5 comparision

In the experiment, random graphs were generated with a number of vertices ranging from 200 to 2000, with 50 differences between test objects. The resulting time was the time needed to check all possible edges divided by the number of vertices. Tests running too long were skipped.

The result of experiment 1 is presented in Figure 2.1. The best graph representations for these given tasks turned out to be Incidence Matrix (IM) and Incidence List (IL). When it comes to IM it has a querying time complexity of $\mathcal{O}(1)$ which gives results instantly. IL allows doing this operation in linear time which is also very efficient. Although it is a little slower incidence list takes much less memory space so it is often a better option.

The edge list and incident matrix both turned out to be inefficient in the given task. The reason for this is that in the worst-case checking for the existence of an edge requires for enumerating all edges. It is quite bad if we take into account that the tested graphs had 0.6 saturation which means there were a lot of edges.

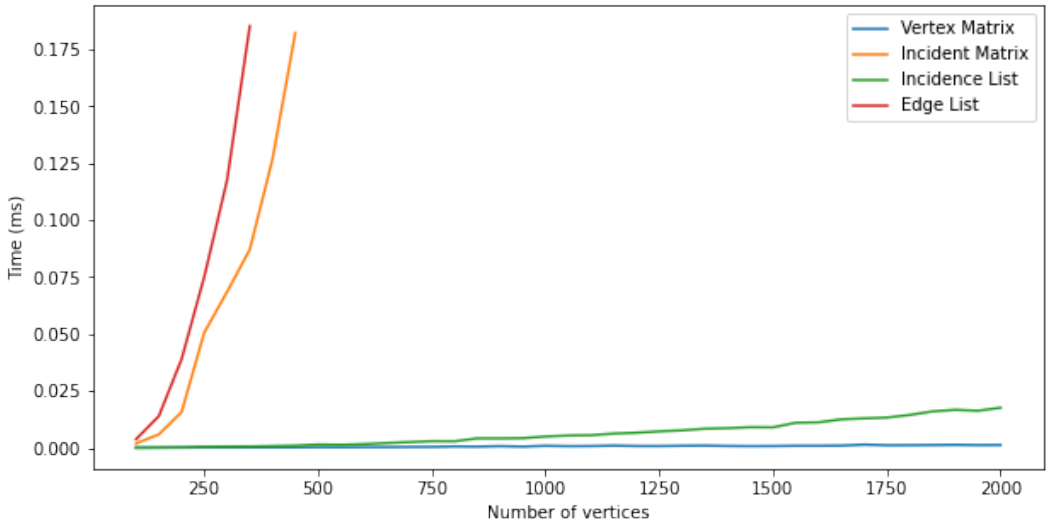

FIGURE 2.1: Time needed to check existence of an edge

2.3 Exercise 2 - Topological Sort Algorithm

2.3.1 Introduction to problem

Topological sort is an ordering of vertices in a way that for every arc u -> v, vertex u comes before v in the ordering. Real-world examples of usage of this algorithm are for example dependency management on a computer system. So if we have a graph of dependencies then we can decide which ones should be installed first. It is only possible to create such ordering on Directed Acyclic Graphs (DAG), for example, take the cyclic graph of dependencies where A depends on B, B depends on C, and C depends on A. It is impossible to decide which one should be installed first because all depend on each other. It is worth mentioning that there might be many solutions to topsort.

2.3.2 Brief details of implementation

The algorithm to solve topological sort is very similar to Depth First Search (DFS), where traversal is implemented with recursion, but there is also an additional stack where vertices are added in the recursive callback.
The best graph representation for this task turns out to be **adjacency list**, as it allows for fast querying for neighbors in the directed graphs. This operation is essential in all graph traversal algorithms. With this representation algorithm should have a worst case of $\mathcal{O}(|V| + |E|)$.
Adjacency matrix would also be a viable option but a slightly worse one, as it requires more memory and would be slower in finding all neighbors of some vertex, this would produce a worst case of $\mathcal{O}(|V|^2)$, so not bad if a number of vertices is small.
When it comes both to **edge matrix** and **edge list**, the algorithm would work very slowly for bigger numbers of edges as finding neighborhoods would be time-consuming.

2.3.3 Performance of the algorithm

The algorithm was tested for DAGs with a number of vertices ranging from 200 to 2000. The difference between data points is 100. Saturation of graphs is equal to 0.3 In Figure 2.2 one can see that performing topological sort for adjacency list is indeed very fast. Even for large graphs with many vertices and edges.

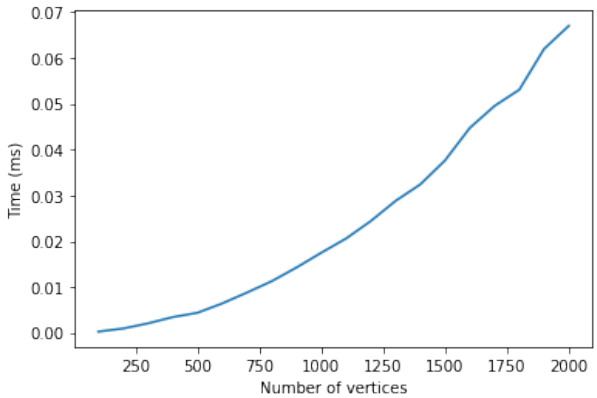

Figure 2.2: Time needed to perform topsort

Chapter 3

Conclusions

In computer science, we often find ourselves in a situation of Time-Space Trade-Off, which means that when solving we have to trade between space in memory and processor time. Often to have the fastest solution we need to use a lot of o space. And in the opposite direction if we want to use less space we have to accept that algorithm might work slower.

This is the situation one faces when needing to choose one of the graph representations to solve the problem. Experiment one showed that you can't do better than the adjacency matrix when it comes to the time needed to query edges from a graph, however, this solution requires a lot of space. Because of that, it is more reasonable to choose adjacency lists that are shown to be a great balance between space and time complexity.

Edge list and edge matrix did poorly in the experiment but they also might have their use cases. For example, it is easy to count the number of edges in these representations, by taking the length of arrays holding them.

When it comes to the problem of topological sorting adjacency list was again the best to solve it. Because of its nature, it is very easy to check neighbors of a certain vertex you just need to check the contents of a list belonging to a certain vertex. Implementation of DFS and BFS is trivial in the vertex list. And as the algorithm I used depends on BFS it was obvious to use this one.

Adjacency matrix would work well too, but finding the neighborhood of a vertex would be an operation with complexity $\mathcal{O}(|V|)$, which isn't bad, however, it would be on average slower than adjacency lists. For edge matrix and edge list finding neighborhood requires checking all edges which is very inefficient.

Poznan University of Technology
Faculty of Computing and Telecommunication
Institute of Computing Science

Typeset using LaTeX.