Cinex Search

Pedro Frazão Pacheco

April 17, 2023

1 Intro

The introduction to algorithms and data structures brought me a love for speed and eficiency in programs. To train some of these techniques I present my new project

Cinex Search[1]

Cinex Search is a software to find the shortest path between two actors. It will then show the movies/series/tv appearances as the bridges.

The datasets were not modified to fit anything and so, a bridge can be an odd talkshow where the two guests appeared. It is not designed to only handle movies so please be aware. The following dataset was downloaded at exactly Sat Apr 15 2023 00:39:51 GMT+0100 (Western European Summer Time). Modifications to it may present unexpected results.

2 Installation

The link to the datasets is *here*.
Datasets names: **name.basics.tsv.gz**, **title.basics.tsv.gz**, **title.principals.tsv.gz**
Please extract the files to a folder called data with the following names:

- name.basics.tsv.gz → actors_data.tsv
- title.basics.tsv.gz → titles_data.tsv
- title.principals.tsv.gz → common_data.tsv

Go to Github *here* and follow the rest of the instructions.

3 Concepts in use

- Depth First Search

[1] Thank's to IDSA's professor for showing The Oracle of Bacon

- Hash Tables
- Linked Lists
- Memory Optimizations
- Time Optimizations

4 Data, Pre-Processing and Constraints

There are 3 datasets.

- The **actors_data** which contains a row per actor/director/producer.
- The **titles_data** which contains a row per movie/tv series/television appearances.
- The **common_data** which contains a row per link between a title and a person.

To better understand these datasets, please download them and take a look.

4.1 Limitations

Sometimes there are multiple actors under the same name. To counter this, the program selects all of those actors and outputs a maximum of 4 of the most famous movies they've been in. You can then select a number and it will choose that line.

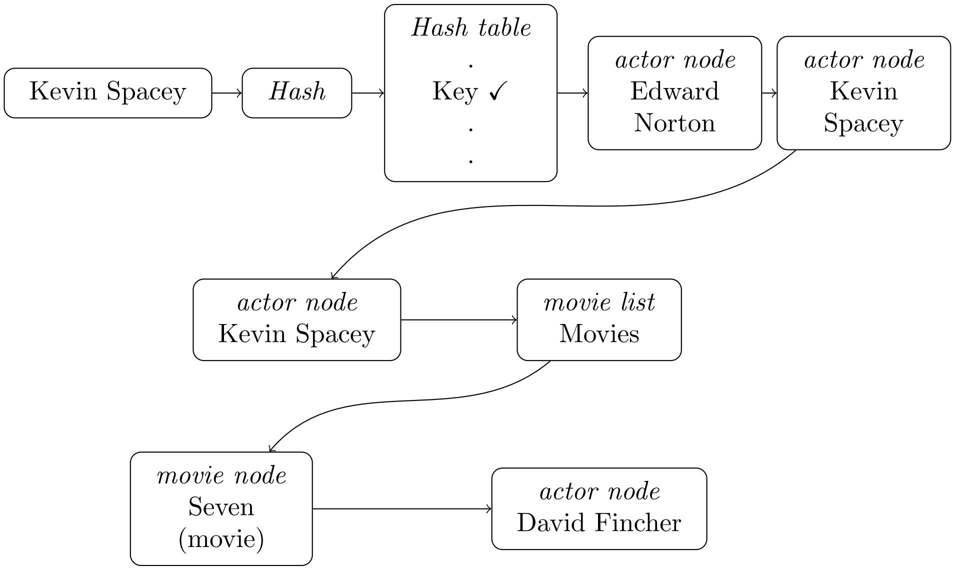
The program was designed to provide instantaneous connections between two actors. However, this comes at the cost of initializing and storing everything in memory. As a result, the program requires approximately 110 seconds to prepare for searching.

4.2 Pre Coding

To make a graph there are a couple of options such as adjacency matrices or adjacency lists. Even before starting coding, we need to discard the adjancency matrices because there would not be enough space. In order to make saving, querying and navigating not only blazingly fast but also memory efficient, we will mostly use pointers to navigate throughout the data set.

So the idea for the schema is the following:

5 Data Schema



6 Thought Process

To better understand the nodes and their connections, please take a look at the code, but to summarize, each actor has a list of movies and each movie has a list of actors.

6.1 Storing Information

Initially, we need to establish every connection between actors and movies. This is achieved by storing both the actor and the movie in a hashmap and then exchanging their pointers. The dataset facilitating this process is the **common_data.tsv**.

There will be three hashmaps: two for the actors and one for the movies. The need for two actor hashmaps arises because, at the beginning, connections are made with the actors' and movies' IDs. Eventually, we aim to perform searches based on the names, not the IDs, which is why we create a second hashmap for user queries.

Afterward, we process the **titles_data.tsv** and use a hash function to access the movie based on the ID and store its name. We repeat this process for the actors' names as well.

6.2 Creating the Hashmap for User Querying

As all the nodes are already available, we avoid recreating them, and instead move each node from the initial actors hashmap to the final one. We achieve this by obtaining the node without breaking the initial hashmap's linked list and then passing the actor's name through the hash function to get the corresponding index. The node is then stored in the new hashmap.

6.3 User Interaction

We now request the user to provide two actor names. To handle cases where two actors share the same name, we pass the user-provided first actor name through the hash function and then traverse every item in the linked list, while storing these actors' pointers.
If the list has more than one entry, duplicates are present, and we display the movies associated with each actor. The user can then identify the intended one. The same process is performed for the second actor. The two actors are then passed to the BFS search.

6.4 BFS Search

Breadth-First Search (BFS) is an algorithm used to identify the shortest path between two nodes, considering that every edge has the same weight.
We enqueue the first actor, search for every movie they are in, and store each related actor in the queue. One note is that we need to store each actor's parent actor, parent movie and mark the visited property to 1, so that we can then reconstruct a path. This process continues until the target actor is found.

7 Optimizations

In this section, we discuss some optimizations implemented with regard to memory and speed.

7.1 Hashmaps

As observed, hashmaps are used exclusively for index access. This ensures near-instant lookup time, as the actors hashmap is approximately $\frac{1}{10}$ of the total size, resulting in an average search time of O(10).

7.2 Linked Lists

Due to the unordered nature of the external chaining in hashmaps, we insert elements in the first position, allowing for O(1) insertion time.

7.3 Pointers

Actors and movies are connected through pointers. This approach not only saves memory space, as there is no need to store each movie ID in the actor and each actor ID in the movie, but also significantly simplifies the process of accessing child nodes (actors in a movie). Each pointer is approximately 8 bytes, as opposed to a variable number of bytes if we were to store strings.

7.4 Bit Reduction

A minor implementation detail involves using 1-bit numbers for booleans. To store the 'visited' property, we only need a 0 or a 1, representing true or false, respectively. With approximately 12 million actors and 10 million movies, instead of storing 22 million integers (704 million bits), we store only about 22 million bits. This results in a memory saving of approximately 85.25 megabytes. In C, this can be achieved by placing a colon between the variable name and the number of bits.

8 Benchmarks

8.1 Memory

When running a massif report (valgrind –tool=massif ./CinexSearch), it shows that there is about 2,978,124,320 bytes allocated. You can view the massif report by goind into the benchmarks folder and running the command (ms_print massif.out.27641).

8.2 Time

Initializing the program takes about 106.454s.
Search times:

- Direct searches take about 0.00114s.
- Double searches take about 0.01486s.
- Triple searches take about 0.06112s.

9 End

Thank you for reading this far. This paper is not intended as an in-depth exploration of the program, but rather as a means to highlight certain concepts and optimization ideas that may spark curiosity about developing a program in the most efficient way possible. One piece of advice is to seek out real-world datasets to work with, as it makes the problem much more interesting to solve. Thank you, and please stay curious.
To get in touch, please feel free to email me at pedrofrazaopacheco@gmail.com.