**Project Algorithms & Data Structures**

Group 35

Ward De Muer & Garben Tanghe

**Content Based Filtering**

**Scenario 0:**

HashMaps:

Advantages of using HashMaps are:

* Provides key-value access to data.
* Time complexity of Search and Insert is O(1) when hashing with chaining or open addressing is used.

Disadvantages of using HashMaps are:

* If a wrong hashing algorithm is used, the performance will decrease dramatically and the HashMap might turn into a linked list.
* When open addressing is used, deleting elements is not a good idea.

Stacks:

Stacks work with the principle of ‘Last In, First Out’ (LIFO).

If we need to search for an element in the stack, we are supposed to remove each element at the top until the required one has been found. But we are also obliged to keep track of all the removed elements, so the use of another stack to temporarily save all other elements until we find the one we’re searching for is necessary. Doing this, we need extra memory and a dramatic increase in time complexity might occur if all the elements must be stored in another stack, and then need to be put back in the original stack.

Queues:

A queue is a “First In, First Out” (FIFO) system, which means that it keeps its order.

A priority queue is always sorted by priority. The element with the highest priority will be handled first.

A fixed size priority queue is almost the same as a normal priority queue, but only a fixed capacity of elements is stored.

Like stacks, searching for elements will cost us more space and time. We start from the head of the queue and remove the elements and keep track of them in another queue until the one we are searching for has been found.

Dictionaries:

Stores a dynamic set of elements, each with an associated key.

Advantages of using dictionaries are:

* Each key is unique.
* When implemented as a linked list or with direct addressing, Insert and Delete have a time complexity of O(1).

Disadvantages of using dictionaries are:

* When implemented as a linked list, Search has a time complexity of O(n).
* When implemented with direct addressing, the used space is O(M).
* When implemented as a sorted array, Insert and Delete have a time complexity of O(n).

Applications of dictionaries are:

* IP addresses
* Telephone numbers

**Scenario 1:** - The complexity of Dynamic1 is O(n5) in worst case.

- A worst case scenario is that all the letters in String s are the same. There will be many (equal) SquareSubsequences. The longest SquareSubsequence will have a length of n when n is even or n-1 if n is uneven, with n = s.length();

- The complexity of Dynamic2: not implemented correctly.

**Collaborative Filtering**

**Scenario 0:** Idem as scenario 0 in ContentBasedFiltering.

**Scenario 2:** The cosine distance is totally wrong when, 0 is a possible rating.

Let’s consider these 2 cases:

Case 1:

|  |  |  |
| --- | --- | --- |
|  | Movie A | Movie B |
| User A | 0 | 5 |
| User B | 0 | 5 |
| User C | 0 | 5 |
| User D | 0 | 5 |
| d(A, B) = 1 - = 1 - = 1 | | |

Case 2:

|  |  |  |
| --- | --- | --- |
|  | Movie A | Movie B |
| User A | 0 | 0 |
| User B | 1 | 0 |
| User C | 0 | 0 |
| User D | 0 | 1 |
| d(A, B) = 1 - = 1 - = 1 | | |

Attention: at least 1 rating should be non-zero!

In case 1, the distance couldn’t be much bigger and we do get the biggest distance possible.

In case 2, the distance is not big at all, but we get an equally big distance as in case 1.

Another case in which the cosine distance is completely wrong, without the possibility of a zero, is this one:

|  |  |  |
| --- | --- | --- |
|  | Movie A | Movie B |
| User A | 1 | 5 |
| User B | 1 | 5 |
| User C | 1 | 5 |
| User D | 1 | 5 |
| d(A, B) = 1 - = 1 - = 0 | | |

The distance should be as big as possible, but we get zero. This is caused by the fact that the 2 vectors are pointing in the same direction.

**Scenario 3:** When a movie is not rated, there is a problem. We can solve it by giving that movie a rating of 2.5. If a user hast just watched one of the greatest movies he’s ever seen or one of a kind he never wants to witness again, he’s much more likely to give this movie a rating. Whereas if the user thinks the movie is rather mediocre, he might not have the needs to express his / her feelings about the movie. Giving this movie an average rating of 2.5 might be a good solution to get rid of movies which don’t have a rating. Another solution is to give the movie the average rating of all the ratings the user gave before or an average of all the ratings other users gave that movie.

**Scenario 4:** A solution for giving a movie a rating for each user is to give it a weighted average, based on other users who have seen the movie. The smaller the distance between 2 users, the bigger the weight and the other way around.

Another method is to use content based filtering. When a user didn’t rate a movie, find the closest movie that the user did rate and give it the same rating. When 2 movies are very similar, the same user should give it almost the same rating.

A similar method to this one, without using content based filtering, is to just look at some of the similar movies to a specific movie, which the user did rate and calculate an average rating for it.

Working further from the last method, we could recommend only the movies with the highest rating by saving the newly created ratings in a fixed size priority queue.