NPS2001C: Matrix Unplugged: Using Computer for Real-World Problems [2320]

GROUP PROJECT MILESTONE 4

Lecturer: Dr Jonathan Kang

Group Members:

A0253501U Ethan Lim Jun Xian

A0254490A Chan Junlin, Dylan

A0253153M Isaac Lim Shile

A0252179Y Ang Rui Jie, Thaddeus

Dijkstra's Algorithm ("get to your chosen toilet quickly")

Dijkstra's algorithm is used for solving shortest path problems for different paths. In *FlushFinder*, this would allow us to connect user's live location to their desired toilet, providing the shortest path possible with the given map data.

The inputs consist of a weighted graph with several nodes of which one is selected to be the point of origin and edges, where each edge is a non-negative weight representing the cost of travelling between said nodes. The outputs are the shortest path lengths from the original node to all other nodes within the graph as well as predecessor points that allow us to reconstruct the shortest route possible. A map of the four toilets from milestone 3 (see map.jpg) and nodes weighted with estimated values are the basis for our executable code. Ideally, the app would take the user's chosen toilet location and current location, feed it into Dijkstra, and calculate the shortest path from their current location to the chosen toilet. An input function with a pre-allocated current location in our executable code simulates this process.

First, the distance array (D) and predecessor array (P) are initialised. Starting from our origin node D is 0 and P is null. A priority queue is also established, storing nodes based on their distance from the origin. The main loop of the algorithm consists of dequeuing node X with the smallest value from the priority queue. Then comparing the distance of the X's neighbouring nodes to the current distance travelled, its predecessor and covered distance is updated if the current distance travelled is smaller. The next updated node is then enqueued into the priority queue. This process repeats until the priority queue is emptied. Once empty, the algorithm will output the shortest path by tracing back along the parent nodes to the starting node.

Key Limitations:

Time complexity

The equation O((V+E)logV), where V is the number of nodes and E is the number of edges, gives us the time complexity of the algorithm. Though efficient if nodes are sparse, it can become increasingly inefficient if graphs become denser. Fortunately, within FlushFinder we plan to work only with the user's immediate surroundings, keeping the number of toilet locations (nodes) to a minimum.

Requires Full Graph Knowledge

The algorithm assumes that the graph with its nodes and distances are known beforehand. However in real life, this graph is constantly in flux and incomplete with new toilets and pathways constantly under construction. To mitigate this, we aim to constantly update toilet and map data.

Alternatives:

Bellman-Ford Algorithm

Also useful for pathfinding, the Bellman-Ford algorithm also takes into consideration negative edge weights. However since our nodes and their weights represent physical distances and locations, the graphs would contain no such negative edge weights, hence there being no need to use this specific algorithm in combination with or over Dijkstra.

IBCF Algorithm ("here are some toilets you might like")

Item-Based Collaborative Filtering in *FlushFinder* recommends new, unused toilets that share similarities with toilets a user had previously used and given high ratings to.

1) Identify two 2 "nearest neighbours" for each toilet in *FlushFinder's* database. These neighbours are the most similar toilets, determined using cosine similarity based on their ratings provided by all users. Cosine similarity calculates how similar two vectors are by measuring the angle between them. In context, if we were to consider toilet 1, its nearest neighbours might be toilets 5 and 4 because they have the highest similarity in ratings.
2) Generate a matrix containing the indices of each toilet's nearest neighbours, which helps to identify most similar toilets to a particular toilet.
3) Find the predicted rating of all unused toilets within a particular radius of the user's location. For each unused toilet "t", its predicted rating by user "u" is determined by *averaging the ratings for similar toilets* (like toilet $i \in Q$ that represents the set of similar toilets in reference to the matrix) *weighted by their similarity to toilet "t"*. This ensures that toilets with similar rating patterns contribute more to the prediction. The formula is:

$$R(t, u) = \{\sum_{i \in Q} S(t, i) * R(i, u)\} / \sum_{i \in Q} S(t, i)$$

R(t, u): predicted rating for toilet t by user u
S(t, i): similarity between toilet t and another toilet i
R(i, u): known rating for toilet i by user u
Q is the set of similar toilets to toilet t

Ultimately, the two toilets with the highest predicted ratings will be recommended to the user.

Key Limitations:

Cold Start Problem for New Items

The cold start problem arises when new items are introduced into the system without any prior interactions or ratings. IBCF relies on historical interaction data to identify similarities between items. Without this data, the system cannot accurately calculate similarities for new items, making it difficult to recommend them to users. New toilets added to *FlushFinder* might remain unrecommended to users until they accumulate enough interactions, delaying the discovery of potentially ideal options.

Less Sensitive to Evolving User Preferences

User preferences can change over time due to various factors, including new experiences, changing needs, or shifts in external conditions. IBCF by design, recommends items based on past interactions, potentially overlooking recent changes in user preferences. *FlushFinder* may continue to recommend toilets based on outdated preferences, missing the opportunity to adapt to the user's current needs or to suggest new options that might better match their evolved tastes.

Alternatives:

User-Based Collaborative Filtering: This method recommends items by finding similar users. The assumption is that users with similar tastes in the past will have similar tastes in the future. However, UBCF may not be as effective in environments where user preferences are highly specific or where user data is sparse.