GRAPH BASED MULTI-ATTRIBUTE MOVIE RECOMMENDATION SYSTEM

Date

Using BFS and DFS algorithms

The first idea was to implement a simple network graph of interconnected movie and analyse **Breadth first search (BFS)** as well as **Depth first search (DFS)** algorithm. But during the process of data preprocessing, we found there is a possibility that movie dataset of any size can be easily transformed to a network graph with edges and nodes. This took us one step ahead to think of developing a movie recommendation system based on similarity attributes of each viewer. As a pilot proof of concept, we are trying a simple algorithm. Movie recommendations can be made using the output data from each search. Recommendation will be illustrated using the output as a sample in this project.

Mini Project-1:

Team: Sushanth S | Mangalam Jain | Jinen Mirje

METHODOLOGY:

Method used for finding the shortest from one movie to another using optimized path of BFS and DFS algorithms:

- Raw data set collection
- Extracting the selected data from the raw data set
- Creating a dictionary of dictionary, the core input that is used for traversal as well as prediction.
- Understanding the movie data set
- Converting the data set into GRAPHS with nodes and edges
- Implementation of BFS and DFS respectively:
- Traversing the child nodes
- Traversing process through iterations
- Pseudocode
- Analysis of output (BFS & DFS)
- Getting the **nearest movie to the target movie** using both methods.
- Application
- Conclusion

EXPLAINING THE DATA SET

The master movies data consists of Movie Genre, and Movie Name as keys.

Further the Movie Name contains details such as director, Id, Genre, length etc.

The list contains 146 unique movies with their attributes.

From the above dataset a subset is extracted and a sample set is formulated, which is shown below:

Below is a sample data set, this kind of data can be stitched for any size of actual real-life scenarios.

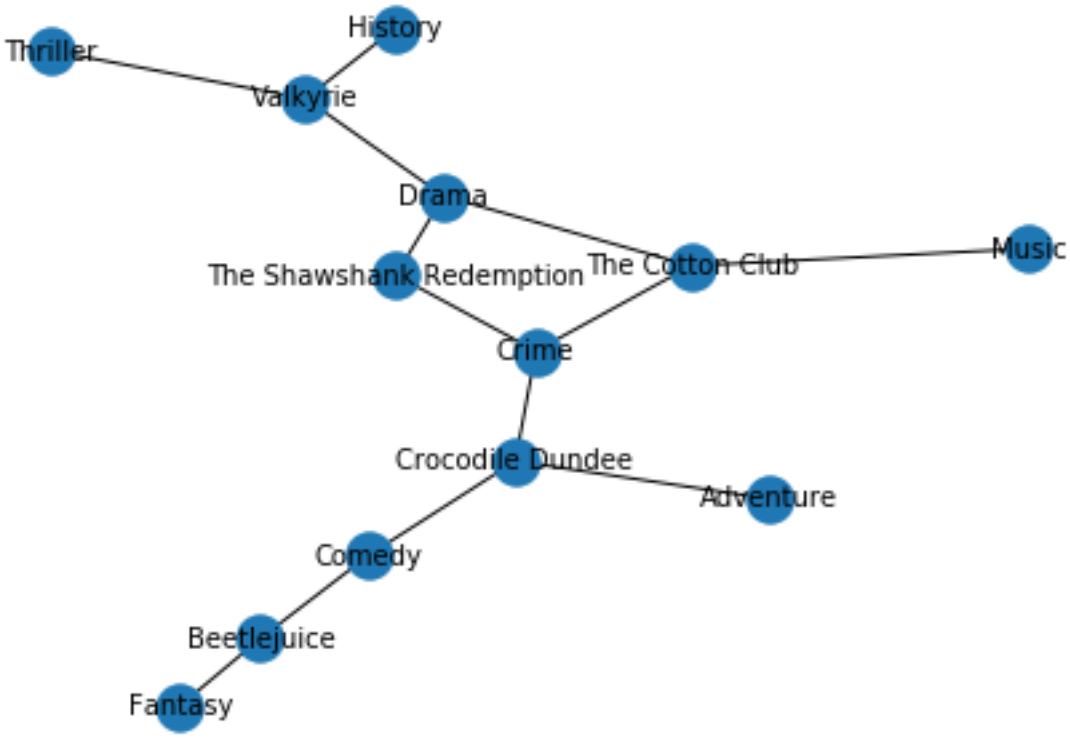
```
d = {'Beetlejuice': {'Comedy': {}, 'Fantasy': {}},
        'The Cotton Club': {'Crime': {}, 'Drama': {}, 'Music': {}},
        'The Shawshank Redemption': {'Crime': {}, 'Drama': {}},
        'Crocodile Dundee': {'Adventure': {}, 'Comedy': {}, 'Crime': {}},
        'Valkyrie': {'Drama': {}, 'History': {}, 'Thriller': {}}, 'Comedy': {'Beetlejuice': {},
        'Crocodile Dundee': {}},  'Fantasy': {'Beetlejuice': {}},  'Crime': {'The Cotton Club': {},
        'The Shawshank Redemption': {}, 'Crocodile Dundee' : {}}, 'Drama': {'The Cotton Club': {},
        'The Shawshank Redemption': {},
        'Valkyrie': {}}, 'Music': {'The Cotton Club': {}},
        'Adventure': {'Crocodile Dundee': {}}, 'History': {'Valkyrie': {}}, 'Thriller': {'Valkyrie': {}}}
```

Converting the Data in Graph

It is useful to convert the data set into a graph format. This can be done through multiple options. Either through an object-oriented programing structure or using a module NetworkX.

In our case we are converting this date to a network of nodes and edges using a network command.

The output of the code resembles as given below:



As it can be observed that all the movies with their respective genres are interconnected to form a network graph.

Implementation of BFS:

Pseudo Code for the implementation:

```
def bfs(dictionary, input_node, target_node):

    # first step is to initialize the empty queue and visited list
    visited = []
    queue = Queue()

    queue.put(input_node)
    visited.append(input_node)

    parent = {}
    parent[input_node] = None

    pathfound = False

    while not queue.empty():
        a = queue.get()
        if a == target_node:
            pathfound = True
            break

        for i in dictionary[a]:
            if i not in visited:
                queue.put(i)
                parent[i] = a
                visited.append(i)
    print(visited)
    # to get the path and represent it as output

    path = []
    if pathfound:
        path.append(target_node)
        #print(parent)
        while parent[target_node] is not None:
            path.append(parent[target_node])
            target_node = parent[target_node]
        path.reverse()
    return path
```

Implementation of DFS:

Pseudo Code for the implementation:

```
visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
```

Analysis the Output from BFS & DFS:

Same input was provided to both the algorithm to understand the path it takes:

Input:

To find the path from movie **"Crocodile Dundee"** and traverse along to reach the last node in the graph.

Output of BFS:

```
Crocodile Dundee, Adventure, Comedy, Crime, Beetlejuice, The Cotton Club, The S
hawshank Redemption, Fantasy, Drama, Music, Valkyrie, History, Thriller
```

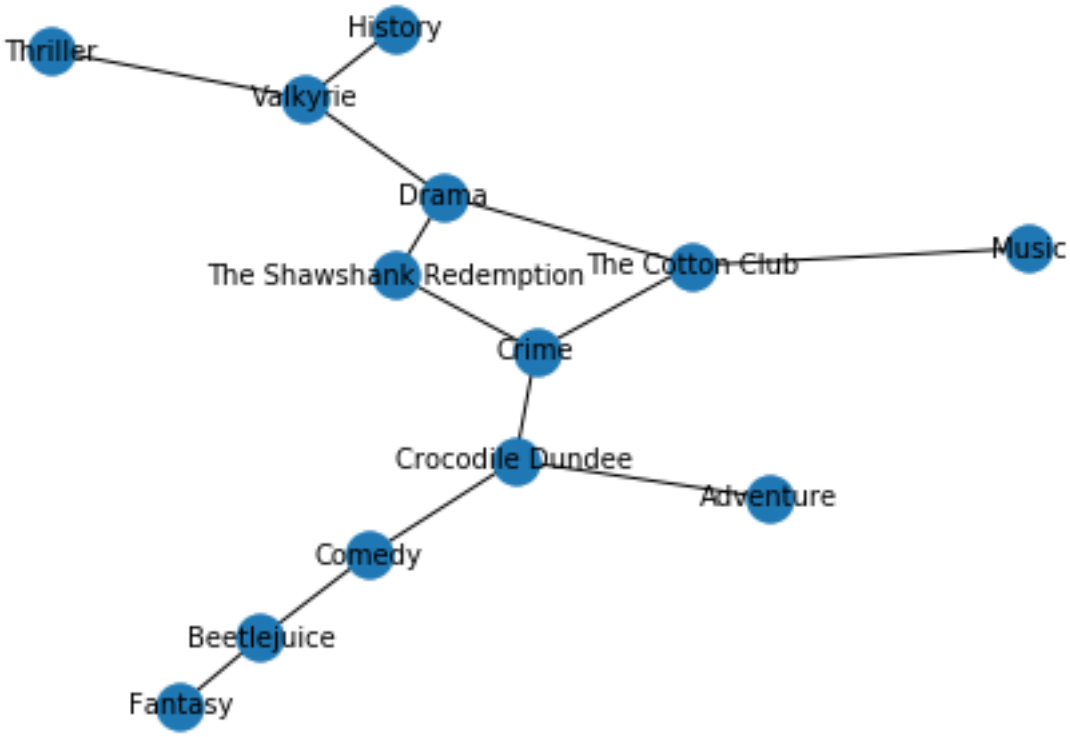
Output of DFS:

```
Crocodile Dundee, Adventure, Comedy, Beetlejuice, Fantasy, Crime, The Cotton Cl
ub, Drama, The Shawshank Redemption, Valkyrie, History, Thriller, Music
```

ANALYSIS AND RECOMMENDATION:

- In our example above we are giving an input **"Crocodile Dundee" as the movie that a customer watched.**
- This is based only on one attribute i.e. genre (**there can be multiple attributes that can be included in the network**)
- In the above network, recommendation can be made using breadth for search algorithm only as it traverses by exploring the nearest neighbors first. Then travels to the second nearest neighbor.
- Depth for search traverses in only one direction can be misleading for the recommendation engine.

Let us understand the functioning of both the algorithms:



BFS: From node (Crocodile Dundee as the central node) it expanded all the adjacent nodes. After every first node is completely expanded the algorithm travels to level-2 nodes that is connecting the first level nodes. So, the farthest point in the node Thriller and History is reached only at the end.

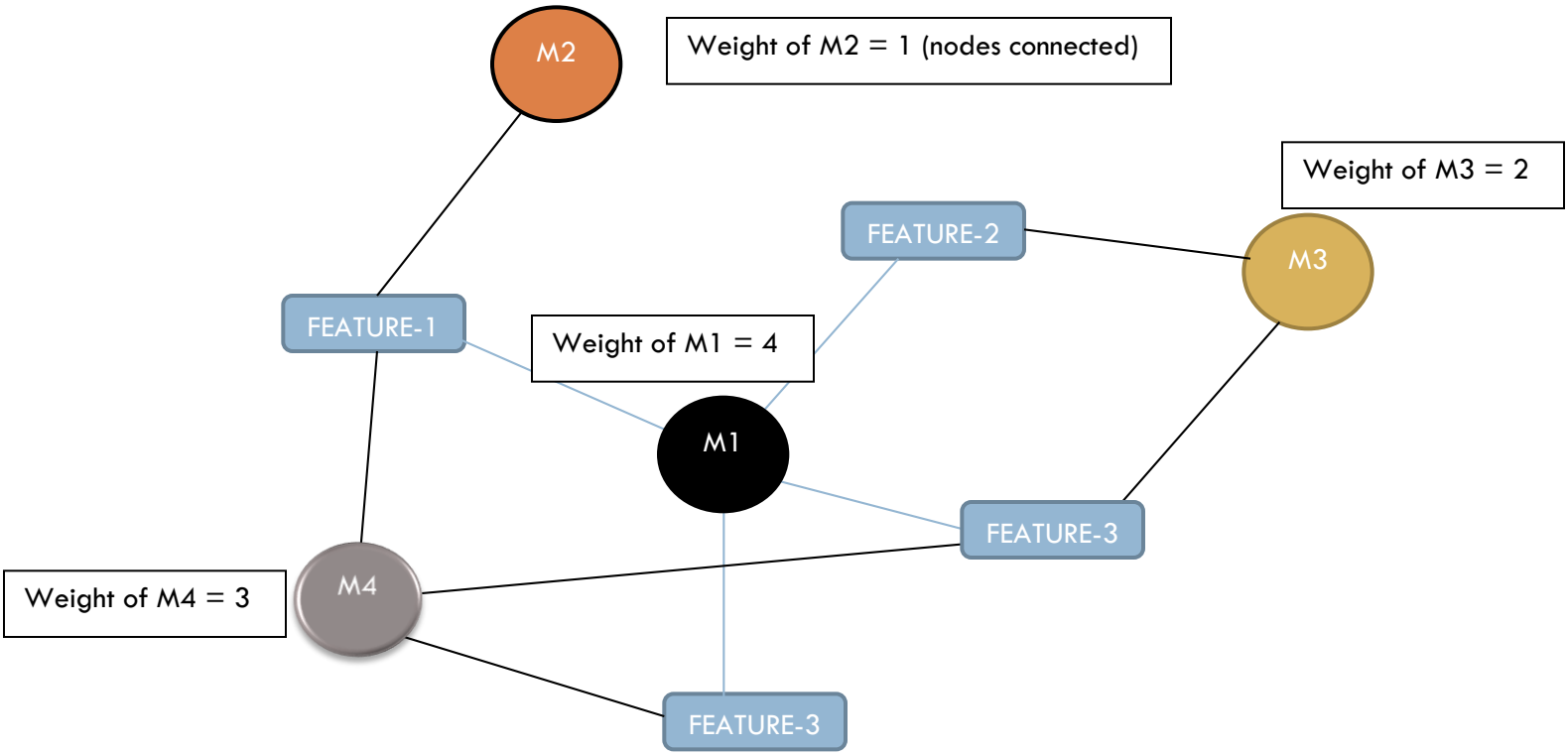
Say:

1. Customer Watched movie: Crocodile Dundee (as the central node)
2. Recommended movie as per BFS algorithm : **Highlighted in red below**

Output of BFS:

Crocodile Dundee, *Adventure*, *Comedy*, *Crime*, **Beetlejuice**, **The Cotton Club**, **The Shawshank Redemption**, *Fantasy*, *Drama*, *Music*, **Valkyrie**, *History*, *Thriller*

3. Based on a complicated network with many nodes and attributes the recommendation can be more precise (see example below)



From the above illustration. Each node can be assigned a weight.

When an individual watched M1, he will be recommended M4 having weight 3, then M3 having weight 2.

Based on a watch history of multiple movies and near future interest, the weights can be adjusted or depreciated. A cumulative recommendation list with weights can be produced. Sorting the final list of a customer can the best recommendation of movies that he/she can watch.

As the graph will update dynamically, the outputs will also get changed automatically.

OTHER APPLICATIONS OF THIS RECOMMENDATION SYSTEM:

All the data that are hierarchical or follow a tree structure. can be converted into a graphical form with nodes and their vector connected with edges. Then these algorithms can be applied to find the optimum travel distance from one node to the other node. To identify the depth of relations.

Some common applications of this algorithms are:

1. Customer product preferences based on their purchase characteristics and recommendation
2. Other general applications of BFS or DFS:
    a. Web page crawling
    b. GPS based traveling
    c. Identifying topology of a tree structure

REFERENCES

Data set : http://github.com/erik-sytnk/movies-list

https://www.hackerearth.com

https://brilliant.org/wiki/depth-first-search-dfs

https://www.tutorialspoint.com/difference-between

End of Report