Project Book Butler is an application that evaluates desired output based on past data. The application is designed to cater users like students who wish to read books, consumers that visit DVD stores and by new DVDs etc.

We have used graphs as the main technique in our program. Graphs are tools that represent information that are easy to read. It’s an easy tool to understand links between different elements such as examining the relation between users which is the central aim of our project. For example, if two students read the same book, an edge would be created between them. Now the weight of that edge would depend on their taste; whether or not they have the same opinion regarding the books they read. larger the weight, similar the opinion.

This application will mainly recommend users based on their past. For example if a student who has previously read book from the libraries and wants to read new books may use this application. The application evaluates the student’s past data; books read and whether liked or disliked etc. and then recommends new books based on that data. Same is the scenario of a person who visits a DVD store and wants to watch new DVDs

Data structures used are dictionaries, lists, tuples

Dictionary is a data structure that has a wide variety of applications. Unlike lists, its index can be in any form called keys. Corresponding to these keys are values. Dictionaries are on of the best ways to represent graphs which is why we have used this data structure to represent all our graphs

We created two main functions ‘Show\_top\_picks’ and ‘recommend\_genre’ which are the options available to the user. The other functions are created in order for these two main functions to operate.

These are some of the pre-algorithms we made that we’ll need for the main functions:

Function openbookfile(name):

1. Import csv
2. Open csv file
3. X:=read csv file
4. d:=()
5. F:=[]
6. for each x in X
7. d=(x)
8. add d to f
9. d:=()

bookdict:={}

for f in F:

bookdict[f[0]]:=[f[2].split(‘.’), f[1], f[3]]

return bookdict

the above function reads the csv file booklist.csv. as the name suggests that, the file contains the data of the books in the following order, book name, author genre and price. In the first step we import csv and in the second we read the file. d is taken to be an empty file and f an empty list. Then a loop is run over the file and every line is added to the tuple and then appended to the list. After that the tuple is made empty again so for that the next line can be added to it and so on the process goes on. then we take an empty dictionary; bookdict. Then a loop is run over the list, over every tuple. The first element of the tuple is made the key of the dictionary and all the other elements in the tuple are stored in list and then made its value. The order of the values are genre, author and price

at the end of this function, this dictionary is stored in some variable; book\_data

function addNodes(G, nodes):

G=:{}

a)for nodes in NODES:

b) G[nodes]:=[]

this function takes an empty dictionary and creates a graph by putting nodes in the graph

function userdataload(name):

1. import csv
2. x:=read csv file
3. d:=()
4. F:=[]
5. for x in X
6. d:=(x)
7. add d to f
8. d:=()
9. bookdict:={}
10. for f in F
11. if x ()
12. final:={}
13. for y in range(length(f))
14. if f[y]:=’’
15. if y0
16. M:=f[y].split(‘,’)
17. o:=M[1]
18. final[M[0]]:=eval(o)
19. bookdict2[f[0]]:=final
20. return bookdict2

in this function we read the csv file user\_data.csv. as the name suggests, it contains the information of the user; the name of the user followed by the books read alongside the preference of each book. Preference pertains to whether the reader liked the book or not; True or False respectively. So first we import csv and read the file. we take an empty tuple and list. And then run a loop over every line in the file, putting the line into a tuple and then appending it to a list. Then the tuple is set to empty so that the next line can be put into the tuple and appended to the list. After the loop has finished, we take an empty dictionary; bookdict2. A loop is run over the list, accessing each tuple. It first checks whether the tuple is empty or not. If so, creates an empty dictionary; final. Then another loop is run over the tuples, accessing every element of the tuple. If that value is an empty string, we break out of the loop. Now we’ll only deal with all the elements except for the 0th element. The element that we’re dealing is in the form ‘bookname, preference (True/False)’. We split this value and store this value in a variable; M and its preference value in another variable; o. then we add these values in the dictionary, final. The booknames becomes keys and the preference values as items. eval(o) converts string value to Boolean. This entire dictionary is made the value of the key; user in the main dictionary; bookdict.

Now we have a dictionary which has readers as keys and the final dictionary as its item, which contains booknames as keys and preference values/True False as their items.

At the end of the function, bookdict is stored in the variable user\_data.

Function addEdges(G, edges, directed=false):

1. for i in range(length(edges)):
2. for key in G:
3. if key:=edges[i][0]:
4. G[key]:=G[keys]+(edges[i][1:])
5. G[edges[i][1]]:=G[edges[i][1]]+ (edges[i][:1]+edges[i][2:]

return

the function creates edges between the nodes in the graph. The graph is undirected.

Function weightEdge\_Create(datadict):

1. Final\_lst:=[]
2. alrdy\_traversed:=[]
3. For person in datadict:
4. Append person in alrdy\_traversed
5. books\_read:=datadict[person]
6. for neighbor in datadict
7. weight:=0
8. for book read by neighbor:
9. if books also read by person
10. neighbor preference:= datadict[neighbor][book read by neighbor]
11. person preference:=books\_read[book read by neighbor]
12. if person preference=neighbor preference
13. weight:=weight+1
14. else
15. weight:=weight-1
16. if weight
17. append (person, weight, weight) in final list
18. return final list

this function evaluates the taste between two readers by looking at their preferences regarding the books both have read. Initially we take two lists; alrdy\_traversed to keep track of the users we’ve already visited and final\_lst that will contain the relations between readers in tuple form. A loop is then run over the readers, for a reader another loop is run over its neighbors. If for a person and his neighbor they have read the same books, the value of weight is incremented by 1 and decremented by 1 if they have the same preference or different preference respectively, for that particular book. Finally, it checks whether the weight is 0 or not, 0 meaning they haven’t read common books, elsewise a tuple is created and appended to the final\_lst.

At the end of the function the final\_lst is stored in the variable edge\_lst.

Function create\_adjlst(G):

Takes an empty graph(empty dictionary) as input and calls the functions addNodes and addEdges and in this way creates a graph that displays the following information; if there’s an edge between two people it means they read the same book/books and weight represents their taste. A higher weight indicates that they have same preferences for almost every mutual book or all books and vice versa.

……………………………………………………………………………………

MAIN FUNCTIONS

Function recommend\_genre(bookdata, name, c):

Input:

Dictionary for every book in the library

User name

Data/books read of every member of the library.

If name in Members:

Userbooks:=member[username]

rec= [∅]

For all x in Userbooks keys and for all y in Userbooks values:

If y=True:

Genre\_list := bookdata[x][0]

Author:=bookdata[x][1]

Price:=bookdata[x][2]

For all genre in Genre\_list:

For every i in bookdata keys and for every j in bookdata values:

If genre in j[0] and I not in rec and i≠x:

Append i and j[2] (price) in rec

rec2= [∅]

for every tuple in rec:

if x not in rec2:

append x in rec2-------------- helps in omitting the repetition of books

for every book in rec2:

output-------- book[0] (name) and book[1] (price)

this function takes the book\_data , any user and user\_data as inputs. Initially the books read by the user are stored in a variable. If the user hasn’t read any books, the function exits and stores this value in the variable; genre\_output=User record not found. Else the function proceeds. A loop runs over the books read by the user: the genre, author and price of that book is stored in three different variables. A empty list; rec(recommendation ) is created. Then another loop is run over the elements in the genre, and within loop another loop is run over the books in book\_data; for books with that same genre and if the book isn’t already by that reader under consideration, both these values are appended in the rec list in tuple form. Another list is created rec2 and tuples that are repeated in rec are appended once in rec 2. The final output is as follows:

name of book + price + genre.

Def top\_picks (G, name):

links := G[name] #returns neighbors of user from adjacency list  
 toplist := []  
 counter := 0

while counter < 2: #links = [(v1, w1), (v2,w2), .. ,(vn, wn)]  
 max\_index := max(links, 1) #find max value by sorting with respect to second element of each tuple  
 toplist += links.pop(max\_index) #add neighbor with max weight to toplist  
 counter +=1

for connection = 1 to len(toplist): #2 people max in toplist   
 read\_books := user\_data[toplist[connection]]   
 #fetch book dict of connection from master user data dict

booklist := read\_books.keys() #takes only titles of books from read\_books dict

recommendation\_list := []

for book\_index = 1 to len(booklist):  
 if booklist[book\_index] not in user\_data[name].keys() and read\_books[booklist[book\_index]] = True:

recommendation\_list += booklist[book\_index]

**Time-Complexities**

In this section, we will analyze the time complexity of every function in our project, determining the complexity of every step and adding them up.

1. def openbookfile(name):

import csv O(1)

with open(name) as csv\_file: O(1)

x=csv.reader(csv\_file) O(n)

d=tuple() O(1)

F=[] O(1)

for y in x: O(m)

for z in y: O(p)

d+=(z,) O(1)

F.append(d) O(1)

d=tuple() O(1)

bookdict={} O(1)

for x in F: O(m)

bookdict[x[0]]=[x[2].split(','),x[1],x[3]] O(1)

return bookdict O(1)

if we add up the complexities, it would give, . (where n is the length no. of rows in csv file, m is the no. of columns and p is the no. of comma separated elements in one cell.).

1. def userdataload(name):

import csv O(1)

with open(name) as csv\_file: O(1)

x=csv.reader(csv\_file) O(n)

d=tuple() O(1)

F=[] O(1)

for y in x: O(m)

for z in y: O(p)

d+=(z,) O(1)

F.append(d) O(1)

d=tuple() O(1)

bookdict2={} O(1)

for x in F: O(m)

if x!=(): O(1)

final={} O(1)

for y in range(len(x)): O(p)

if y!=0: O(1)

M=x[y].split(',') O(1)

o=M[1] O(1)

final[M[0]]=eval(o) O(1)

bookdict2[x[0]]=final O(1)

return bookdict2 O(1)

Adding up the complexities would give us, .

1. def WeightedEdge\_Create(datadict):

final\_lst=[] O(1)

alrdy\_traversed = [] O(1)

for person in datadict.keys(): O(V)

alrdy\_traversed.append(person) O(1)

books\_read=datadict[person] O(1)

for neighbor in datadict.keys(): O(V)

weight=0 O(1)

if neighbor in alrdy\_traversed: O(1)

continue O(1)

for n\_books in datadict[neighbor].keys(): O(v)

if n\_books in books\_read.keys(): O(v)

currentchoice=books\_read[n\_books] O(1)

neighborchoice=datadict[neighbor][n\_books] O(1)

if currentchoice==neighborchoice: O(1)

weight+=1 O(1)

else: O(1)

weight-=1 O(1)

if weight != 0: O(1)

final\_lst.append((person, neighbor, weight)) O(1)

return final\_lst O(1)

Summing up would give Time complexity as, where V is the number of vertices of main dictionary and v is the no. of vertices of nested dictionary.

1. def GetMeADuo(G,name):

Final=[] O(1)

a=G[name] O(1)

t=0 O(1)

for x in a: O(n)

if x[1]>t: O(1)

Final.append(x[0]) O(1)

t=x[1] O(1)

if Final==[]: O(1)

print('You have no match :c') O(1)

else: O(1)

print('Your highest duo score is with ',end='') O(1)

print(Final[-1]+'.') O(1)

return Final O(1)

the time complexity of this function would be O(n) only.

1. def recommend\_genre(bookdata,name,c):

global genre\_output O(1)

try: O(1)

userbooks=c[name] O(1)

except: O(1)

genre\_output = 'User record not found!' O(1)

return O(1)

for x,y in userbooks.items(): O(V\*E)

if y==True: O(1)

genre=bookdata[x][0] O(1)

author=bookdata[x][1] O(1)

price=bookdata[x][2] O(1)

rec=[] O(1)

for gen in genre: O(g)

for i,j in bookdata.items(): O(v\*e)

if gen in j[0] and i not in rec and i!=x: O(w\*r)

rec.append((i,j[2])) O(1)

for x in rec: O(K)

genre\_output+= x[0] + ' for $' + x[1] + '\n' O(1)

return O(1)

The total time complexity would be, Where V are the vertices of main dictionary, E are the Edges of main dictionary, g is the length of genre of initial book of the user, v and e are the vertices and edges of Book data dictionary and lastly, w and r are the length of genre of each book in book data dictionary, whereas r is the length of rec list.

1. def save\_new\_entries(user\_name, user\_books):

global new\_user\_data O(1)

new\_user\_data=[user\_name] O(1)

for x in user\_books: O(n)

new\_user\_data.append(x) O(1)

for this function, complexity would be simply .

1. def New\_User(datafile,new\_user\_data):

import csv O(1)

with open(datafile, 'a') as csvFile: O(1)

writer = csv.writer(csvFile) O(1)

writer.writerow(new\_user\_data) O(n)

Time complexity would be, O(n).

1. def Add\_to\_records(datafile, user\_name, user\_books):

global new\_user\_data O(1)

save\_new\_entries(user\_name, user\_books) O(n)

New\_User(datafile,new\_user\_data) O(n)

The complexity for this function would be, .

1. def top\_picks(G, name):

global user\_data O(1)

links = G[name] O(1)

counter = 0 O(1)

toplist = [] O(1)

while counter <2: O(E)

maxval = 0 O(1)

max\_index = None O(1)

for person in range(len(links)): O(E)

if links[person][1] > maxval: O(1)

maxval = links[person][1] O(1)

max\_index = person O(1)

if max\_index: O(1)

toplist.append(links.pop(max\_index)[0]) O(1)

counter +=1 O(1)

else: O(1)

break O(1)

recommendation = [] O(1)

for connection in toplist: O(2)

read\_books = user\_data[connection] O(1)

for book in read\_books.keys(): O(V)

if book not in user\_data[name].keys() and read\_books[book]: O(V)

recommendation.append(book) O(1)

return recommendation O(1)

The complexity for this function would be, . Where E would be length of value of user links dictionary and V is the number of users of library.

1. def show\_top\_picks(G, name, book\_data):

global top\_picks\_display O(1)

top\_picks\_display = '' O(1)

try: O(1)

picks = top\_picks(G, name)

for entry in picks: O(1)

top\_picks\_display+= entry+'\t Genres: '+str(', '.join(book\_data[entry][0]))+'\n' O(1)

except: O(1)

top\_picks\_display = 'User record not found!' O(1)

return O(1)

The complexity would be same as the top\_picks function (as it is basically calling that function), hence it would .