Cairo University
Faculty of Computers and Artificial Intelligence

CS361

Artificial intelligence

2nd Semester 2020 Project

Topic (1): Travel agent

Third Year- Computer Science

Spring Semester

PID23868220

Team:

ID	Name	Email	Group
20170071	Aya Mahmoud	ayamahmoud@stud.fci-cu.edu.eg	1-CS
20170411	Moaaz Hasan	mhr411@stud.fci-cu.edu.eg	3-CS
20170419	Yageen Mohamed	ykeen12@stud.fci-cu.edu.eg	3-CS
20180358	Sara Qaid ALomary	Salomary1995@stud.fci-cu.edu.eg	IS-1
20170382	Nadia Idris	nadia98.amer@stud.fci-cu.edu.eg	4-CS

Table of Contents

Introduction (what is problem solving methods (PSMs)):

Problem Solving Methods (PSMs) are field-independent thinking components that define behavior patterns that can be reused again via applications. While the availability of large-scale PSM libraries and the resulting consensus about the languages of the PSM specifications indicates the maturity and presence of the field in the current era, many and many important research issues are still open in this area. In particular, little progress has been made on foundational and methodological issues. Libraries at present in PSMs lack a clear theoretical basis and provide little support for the method development process, usually in the form of informal guidelines.

The ability to solve problems is an essential life skill and is necessary for our daily life at home, school and work. We solve problems every day without really thinking about how to solve them. For example: It's raining and you need to go to the store. What do you do? There are lots of possible solutions. Take your umbrella and walk. If you don't want to get wet, you can drive, or take the bus. You might decide to call a friend for a ride, or you might decide to go to the store another day. There is no right way to solve this problem and different people will solve it differently.

Problem solving is the process of identifying a problem, developing possible solution paths, and taking the appropriate course of action.

Problem Solving Methods (PSM) describes independent and domain-specific heuristics, which define patterns of behavior that can be reused again via different applications. For example, a suggestion and review (Marcus et al., 1988; Zdrahal and Motta, 1995) provides a general mode of thinking, characterized by an iterative sequence of an 'extension' and 'review' model, which can be reused very easily for a scheduling solution (Stout et al, 1988) and design problems (Marcus et al., 1988).

PSMs provide an important technology to support structured and structured development approaches to knowledge engineering: they can be used

i) To provide robust model-based frameworks for implementing knowledge acquisition (Marcus, 1988; van Heijst et al., 1992)

ii) To support the rapid development of robust and maintainable applications through a re-use of the component (Runkel et al., 1996; Motta, 1997; Motta and Zdrahal, 1997). In general, the study of PSMs can be seen as a way to go beyond the idea of knowledge engineering as an "art" (Feigenbaum, 1977), to formulate a field-oriented methodology, which would make it possible to produce rigorous manuals similar to those available in other engineering fields. From a philosophical perspective, such organization can be used as a source for the experiment of "functional theories of intelligence" (Chandrasekaran, 1987).

Why is problem solving important?

Good problem-solving skills enable you not only in your daily personal life, but also important in your professional and work life. In today's rapidly changing global economy, employers often define daily problem solving as critical to the success of their organization. For employees, problem solving can be used to develop practical and creative solutions, and to show independence and initiative to employers.

What Does Problem Solving Look Like?

The ability to solve problems is a skill, and like any other skill, the more and more you practice and repeat, the better results you get. How exactly do you practice solving your problems? Knowing the different problem-solving strategies and when to use them will give you a good start in learning and mastering problem solving that are practical. Most strategies provide you with a set of steps to help you identify the problem and choose the best solution. There are two main types of strategies: an algorithm and a guide.

Background (different PSMs techniques)

Some problem-solving methods techniques:

Inductive reasoning is concluding general results from a specific detail which needs to use observation and what is learned from other experiences to reach the general truth. And it is different from Deductive Reasoning which is applying a concluded general or theory results on a specific instance. Example on Inductive reasoning a market owner notices that putting some unknown products next to other regular product make the buyer buy more of it and notices it. Deductive Reasoning as saying all fruits does have seeds an apple is a fruit so apples have seeds. Generate-and-Test a good generating for the possible solutions must be completed until a suitable solution is found. Example as in playing the chess with the computer it will generating all the possible moves and chose the one that may lead to the win. Means-ends analysis it works by solving the obstacles between goal state and current state as if it sub goals it is used in the long range goals example if thier is points A till Z and the goal to reach it first sub goal is to reach point B and then C and so on till reaching the goal Z. Problem reduction it is as the divide and conquer strategy to solve the problem reduce it to another problem that you know how to solve then recombined to get a solution as a whole ,example to find the Closest Pair of Points in a set of points divide the pairs and find the closest in each group and then find the closest from the last results.

Travel agent as a search problem

Travel agent is a search problem as most travelers prefer to search for the shortest duration between two cites (source and destination). So instead of planning their trip, a travel agent software system helps achieving this task for them taking into consideration the longitude and latitude of each city, arrival, and departure time between them. A* search finds the shortest path between two nodes according to their weights and with respect to another factor H. this is exactly what is done here in the travel agent problem the weights 'g' is the duration between the two cities (nodes) and the H is the distance between two points "longitude and latitude". We add the 'H' and 'g' to get the value of 'f 'searching for the minimum path between the two cities.

System component

First our system needs to read excel info with mentioned directory path. It's stored in lists as known as system facts in prolog. User is required to enter the source and destination cites, his suitable flight days (allowed days). **Input example: please enter source: Alexandria, please enter destination Cairo, allowed days: sun tue (with enter between them)** Then our system search for a flight path using a star search algorithm.

The **search function** takes the source and destination strings as parameters and returns path found or none (null) if not found **printing "No path available".**

Our system consists of nodes (single city) each node holds all information of the city name, parent (source city which is connected before it), arrival time, departure time, flight number, available flight days, city latitude and city longitude, g, h, f which is calculated in the constructor. Whenever a node is needed in function whether we convert string taken from the user into a node by the constructor or read the specific index from the lists (system facts read from the excel sheet). Within the Constructor we overwrote sort function to get the shortest duration between the self 'this' node and other one.

as soon as A* search work we have open list (contain all nodes) then sort them to get the shortest one, and visited list (as marker to make sure that there is no repeating), then we have two cases FIRST one when the current node is the GOAL we move to **getPath Function** in it we receive current node and destination node and get the full path and return it.

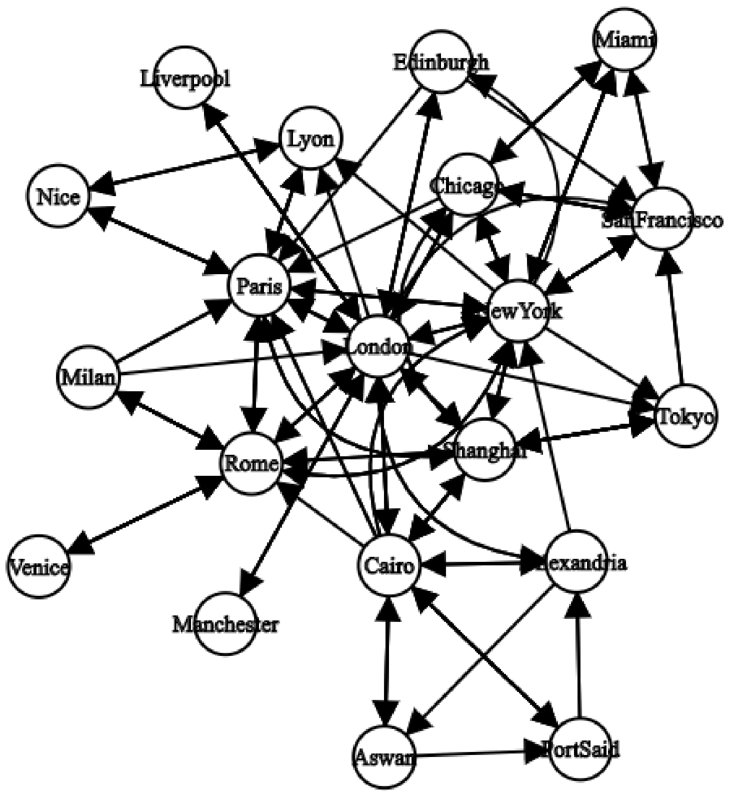
The **SECOND** one is to move to getChildren Function in it we return all possible nodes that can we go from this node and then loop across children and get its index check if it is visited and append it in open list.

We handle the overall of travel agent in print path function **there are three main scenarios**.it takes the path that was returned in A* search function and allowed days that the user entered first we create the rang days by passing allowed days to range days function already we have fact consist of all days we get the indexes and make them the new rang of our days Like if user enter(Sunday Tuesday) we get the index of Sunday =1 and Tuesday 4 and return array consist those days and days between them. Then check if the path days in rang or not if in range we **print its path information** this is the **first scenario**. The **second one** is if it's not in range we check another path which is longer than the last one (as we didn't fulfil the allowed days condition). we **print path info** if a path was found, however, if it fails and there's no path within the same range that the user entered, we move on to **the last scenario** where we are supposed to minus one and add one to the days range and then compare according to the new rang if we fail again in the new range we will **print No path allowed in this range**.

Testcases:

Test case 1

No path found which the case is where no path leads to the destination city. This maps to our code where the open list was empty so there is no path to check. This case happens when user choses a destination city where no flights are supported or there is no intermediate city that links source city to the destination. **Output**: no path found. Note: that this test case doesn't work on the available data all cities are connected together. The following graph represents the cities connection and that there is no case that user chooses two cities that aren't connected.



Printing function test cases which takes the path and checks the allowed days user input. It contains three test cases. Whether it print the result immediately, checks a longer path or subtract and adds 1 to the allowed days range.

Test case 2

Shortest path found. in this case we have to print the path info in user friendly way showing the flight number and the transit flights with the source and destination city names. **Output**: path info with the steps.

Expected Input: Source city: Alexandria, Destination city: Edinburgh
Range for allowed two days: sun, mon.

Expected output: allowed path source range days are ['mon', 'tue', 'wed']
first flight: flight MS003 Form Alexandria to Cairo

second flight: flight MS021 Form Cairo to Rome

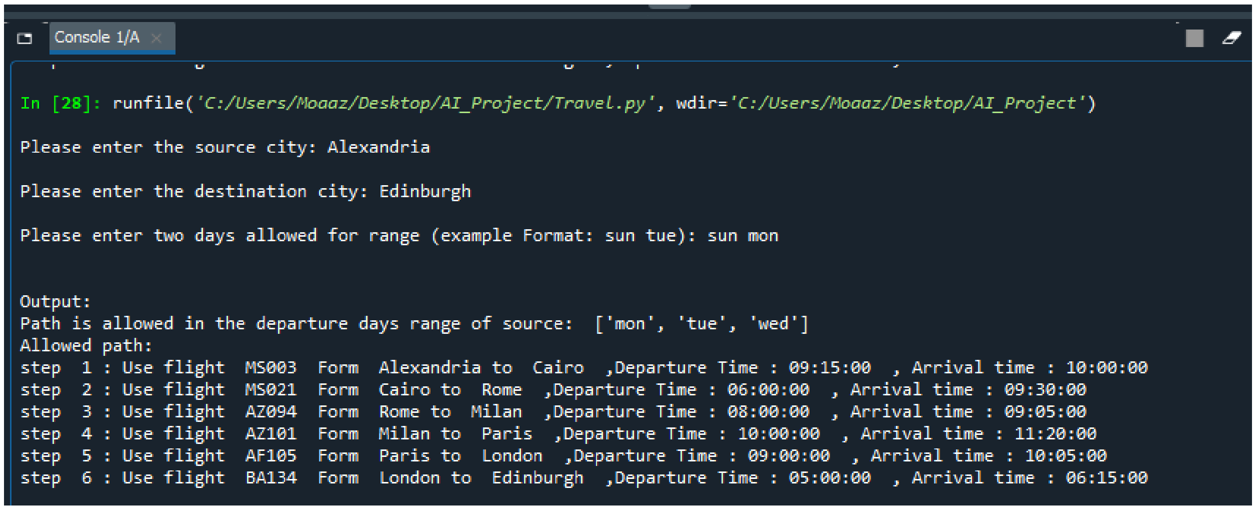
third flight: flight AZ094 Form Rome to Milan

fourth flight: flight AZ101 Form Milan to Paris

fifth flight: flight AF105 Form Paris to London

sixth flight: flight BA134 Form London to Edinburgh

the following screenshot represents this testcase input and output



```
In [28]: runfile('C:/Users/Moaaz/Desktop/AI_Project/Travel.py', wdir='C:/Users/Moaaz/Desktop/AI_Project')

Please enter the source city: Alexandria

Please enter the destination city: Edinburgh

Please enter two days allowed for range (example Format: sun tue): sun mon


Output:
Path is allowed in the departure days range of source:  ['mon', 'tue', 'wed']
Allowed path:
step  1 : Use flight  MS003  Form  Alexandria to  Cairo  ,Departure Time : 09:15:00  , Arrival time : 10:00:00
step  2 : Use flight  MS021  Form  Cairo to  Rome  ,Departure Time : 06:00:00  , Arrival time : 09:30:00
step  3 : Use flight  AZ094  Form  Rome to  Milan  ,Departure Time : 08:00:00  , Arrival time : 09:05:00
step  4 : Use flight  AZ101  Form  Milan to  Paris  ,Departure Time : 10:00:00  , Arrival time : 11:20:00
step  5 : Use flight  AF105  Form  Paris to  London  ,Departure Time : 09:00:00  , Arrival time : 10:05:00
step  6 : Use flight  BA134  Form  London to  Edinburgh  ,Departure Time : 05:00:00  , Arrival time : 06:15:00
```

Test case 3

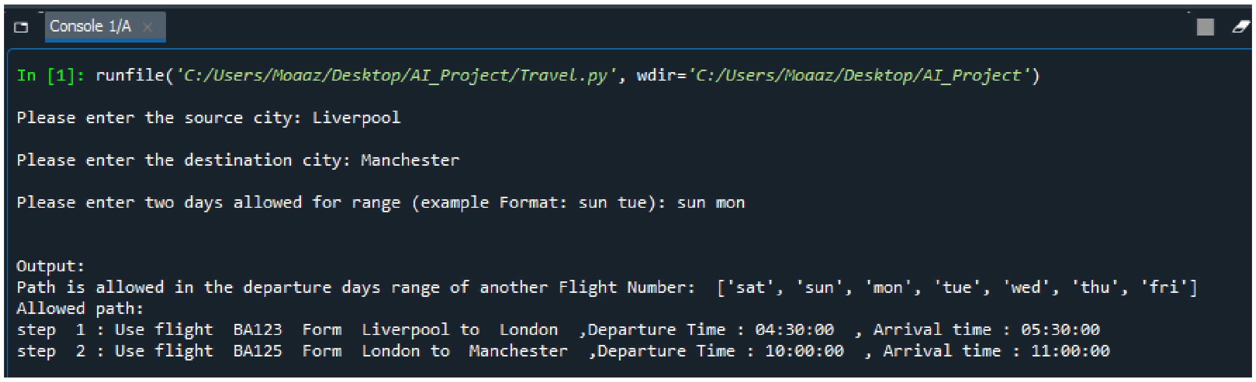
The first flight step is checked as the latter was not within the allowed(required) user fight(departure) days. we search with the same first flight city names (source and destination) only changing the flight number as the its flight days may suits the user demand. **output**: path info with the steps or moves to the other case adding and subtracting to the allowed days range.

Expected Input: Source city: Liverpool, Destination city: Manchester
Range for allowed two days: sun, mon.

Expected output: path is allowed in the range days of source 2['sat', 'sun', 'mon', 'tue', 'wed', 'thu', 'fri']
first flight: flight BA123 Form Liverpool to London

second flight: flight BA125 Form London to Manchester

the following screenshot represents this testcase input and output



```
In [1]: runfile('C:/Users/Moaaz/Desktop/AI_Project/Travel.py', wdir='C:/Users/Moaaz/Desktop/AI_Project')

Please enter the source city: Liverpool

Please enter the destination city: Manchester

Please enter two days allowed for range (example Format: sun tue): sun mon


Output:
Path is allowed in the departure days range of another Flight Number:  ['sat', 'sun', 'mon', 'tue', 'wed', 'thu', 'fri']
Allowed path:
step  1 : Use flight  BA123  Form  Liverpool to  London  ,Departure Time : 04:30:00  , Arrival time : 05:30:00
step  2 : Use flight  BA125  Form  London to  Manchester  ,Departure Time : 10:00:00  , Arrival time : 11:00:00
```

Test case 4

In this case, we make a new allowed days range by adding and subtracting to the range. As the user's allowed days range didn't match the first flight timetable.

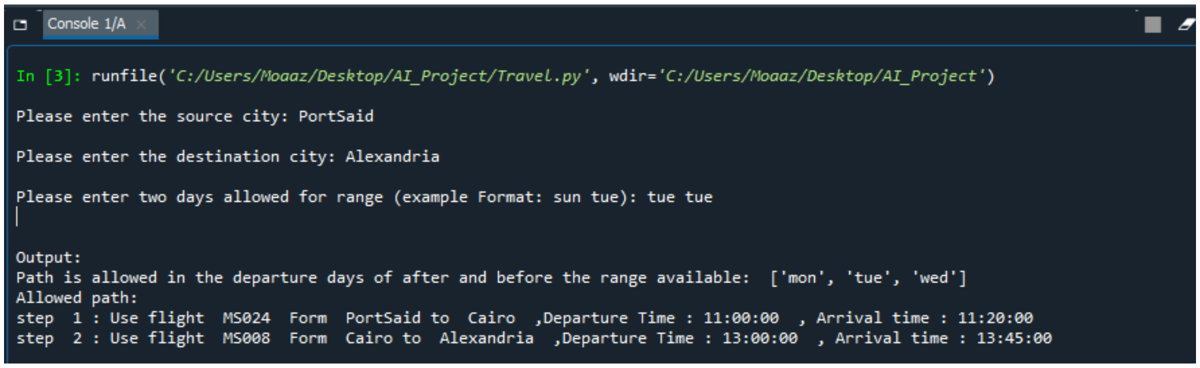
4.1 so either it prints the path as follows. **output**: printing path info with the steps

Expected Input: Source city: PortSaid, Destination city: Alexandria
Range for allowed two days: tue, tue.

Expected output: path is allowed in the days after and before the range available ['mon', 'tue', 'wed']
first flight: flight MS024 Form PortSaid to Cairo

second flight: flight MS008 Form Cairo to Alexandria

the following screenshot represents this testcase input and output



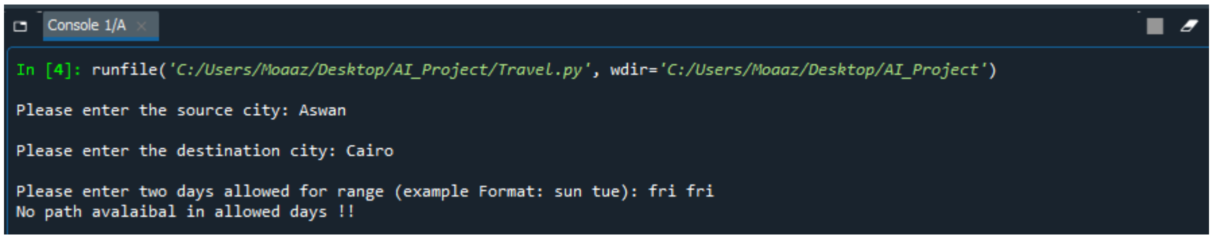
```
In [3]: runfile('C:/Users/Moaaz/Desktop/AI_Project/Travel.py', wdir='C:/Users/Moaaz/Desktop/AI_Project')

Please enter the source city: PortSaid

Please enter the destination city: Alexandria

Please enter two days allowed for range (example Format: sun tue): tue tue


Output:
Path is allowed in the departure days of after and before the range available:  ['mon', 'tue', 'wed']
Allowed path:
step  1 : Use flight  MS024  Form  PortSaid to  Cairo  ,Departure Time : 11:00:00  , Arrival time : 11:20:00
step  2 : Use flight  MS008  Form  Cairo to  Alexandria  ,Departure Time : 13:00:00  , Arrival time : 13:45:00
```

4.2 or nothing could be done so we have to inform the user with the available days so maybe he could change his mind and pickup another day for the flight. **output**: no path available just prints the allowed flight days as there was no matches.

Expected Input: Source city: Aswan, Destination city: Cairo
Range for allowed two days: fri, fri.

Expected output: No path available in allowed days!

the following screenshot represents this testcase input and output



```
In [4]: runfile('C:/Users/Moaaz/Desktop/AI_Project/Travel.py', wdir='C:/Users/Moaaz/Desktop/AI_Project')

Please enter the source city: Aswan

Please enter the destination city: Cairo

Please enter two days allowed for range (example Format: sun tue): fri fri
No path avalaibal in allowed days !!
```

References

http://ksi.cpsc.ucalgary.ca/KAW/KAW98/fensel2/

https://ccmit.mit.edu/problem-solving/

Appendix

```python
import pandas as pd
import math

excel_file_name = 'C:\\Users\Moaaz\Desktop\AI_Project\Travel Agent2.xlsx'
travelAgent = pd.read_excel(excel_file_name)
travelAgent2 = pd.read_excel(excel_file_name , sheet_name='Cities')
Source = travelAgent['Source'].values.tolist()
Destination = travelAgent['Destination'].values.tolist()
DepartureTime = travelAgent['Departure Time'].values.tolist()
ArrivalTime = travelAgent['Arrival Time'].values.tolist()
FlightNumber = travelAgent['Flight Number'].values.tolist()
ListofDays = travelAgent['List of Days']
Cities = travelAgent2['City'].values.tolist()
Latitude = travelAgent2['Latitude'].values.tolist()
Longitude = travelAgent2['Longitude'].values.tolist()
days = ['sat','sun', 'mon','tue','wed','thu','fri']

# to remove all spaces in Source and Destination
for s in range(len(Source)):
    Source[s] = Source[s].replace(" ","")
for d in range(len(Destination)):
    Destination[d] = Destination[d].replace(" ","")


# to claculate the G
def claculate_g(D,A):
    H = abs(D.hour - A.hour)
    M = abs(D.minute - A.minute)
    return H + (M/60)

# to claculate the H
def calculate_h(x,y,parent):
    if parent == None:
        return math.sqrt((x - 0)**2 + (y-0)**2)
    else:
        return math.sqrt((x - parent.x)**2 + (y - parent.y)**2)

# Node for take info about the city
class Node:
    def __init__(self, name , parent , index):
        self.name = name
        self.parent = parent
        self.Arrival = ArrivalTime[index]
        self.Departure = DepartureTime[index]
        self.FN = FlightNumber[index]
        self.Days = ListofDays[index][1:-1].split(',')
        index2 = Cities.index(self.name.replace(" ",""))
        self.x = Latitude[index2]
        self.y = Longitude[index2]
        if parent is None:
            self.g = 0
        else:
            self.g = parent.g + claculate_g(self.Departure,self.Arrival)
        self.h = calculate_h(self.x ,self.y,parent)
        self.f = self.g + self.h
    # To print node object with name and f value
    def __repr__(self):
        return str(self.name) + ' ,' + str(self.f)
    # Sort nodes
    def __lt__(self, other):
         return self.f < other.f

# to get the index of the city to get info for it take the source  and
destination
def get_Index(s,d):
    SIndex = -1
    DIndex = -1

    for x in range(len(Source)):
        if s == Source[x]:
            SIndex = x
            break
    for y in range(SIndex , len(Destination)):
        if d == Destination[y]:
            DIndex = y
            break
    return DIndex , DIndex

# to get all cities that connect with source city
def getChaildern(name):
    child = []
    for x in range(len(Source)):
        if Source[x] == name:
            child.append(Destination[x])
    return child

# if found the gole city so it's get all path from gole to the start node by
parent node
def getPath(current_node , start_node):
    path = []
    while current_node != start_node:
        path.append(current_node)
        current_node = current_node.parent
    path.append(start_node)
    # take the reverse path from start to gole
    NewPath = path[::-1]
    s, d = get_Index(NewPath[0].name,NewPath[1].name)
    New_node = Node(NewPath[0].name, None , s)
    NewPath[0] = New_node
    return NewPath
```

```
# check if node in open
def check_open(open, child):
    for node in open:
        if (child.name == node.name and child.f >= node.f):
            return False
        if (child.name == node.name):
            open.remove(node)
    return True

# check if node is visted
def check_visted(visted, child):
    for node in visted:
        if (child.name == node.name):
            return False
    return True

# the A star algorithem that get the shortest path from start node to gole
def A_search(start,gole):

    open = []
    visted = []

    # get the index of the start node to creat all info of it
    i = Source.index(start)
    start_node = Node(start, None , i)

    # add start node to open list
    open.append(start_node)

    # loop until the open list is empty thats tell us there is no path
    while len(open) > 0:

        # sort the open list depend on cost of F to take the lowest cost
        open.sort()
        current_node = open.pop(0)

        # add the current node to visted list
        visted.append(current_node)

        # here if found the gole path will return it
        if current_node.name == gole:
            return getPath(current_node , start_node)

        # get all childerns of the current node
        childs = getChaildern(current_node.name)

        # loop to all childs to check if in open list or is visted
        for child in childs:
            i1 , i2 = get_Index(current_node.name , child )
            Child  = Node(child, current_node ,i2)

            if(not check_visted(visted , Child)):
                continue

            if(check_open(open, Child) == True):
                open.append(Child)

    return None
```

```
# check if there is one day of the source city in the avalibale range of days
def Check_Days(list1,list2):
    for n in range(len(list1)):
        list1[n] = list1[n].replace(" ","")
    for s in range(len(list2)):
        list2[s] = list2[s].replace(" ","")

    for i in list1:
        if i in list2:
            return True
    return False

# to create the range of days
def rangeDays(listDays):
    i = days.index(listDays[0])
    j = days.index(listDays[1])
    return days[i:j+1]

# check if the path that founded is allowed in range of days or no
def printPath(path, allowedDays):
    i ,j = get_Index(path[0].name,path[1].name)
    notFound = True
    NewDays = []
    isDayExist = False
    allowedDays = rangeDays(allowedDays)
    # check days of the source path is allowed
    if Check_Days(allowedDays , path[0].Days):
        print('\n\nOutput:')
        print('Path is allowed in the departure days range of source: ',
path[0].Days)
        print('Allowed path:')
        for p in range(len(path)):
            if p > 0:
                print('step ',p ,':' ,'Use flight ',path[p].FN, ' Form ',
path[p].parent.name , 'to ', path[p].name , ' ,Departure Time :',
path[p].Departure , ' , Arrival time :' , path[p].Arrival  )
    # if not allowed in source path I will check if there is a soucned flight
number is allowed with it's days
    else:
        while True:
            isDaysInNewFN = True
            isSourceNotEqualDestination = True

            if Source[i] == path[0].name and Destination[j] == path[1].name:
                isSourceNotEqualDestination = False
                if path[0].FN == FlightNumber[j]:
                    isDaysInNewFN = False
                    isSourceNotEqualDestination = True
                    i = i+1
                    j = j+1
            if isDaysInNewFN:
                NewFN = FlightNumber[i]
                NewDays = ListofDays[i][1:-1].split(',')
                isDayExist = Check_Days( allowedDays , NewDays )
            if isDayExist:
                path[0].FN = NewFN
                path[0].Days = NewDays
                print('\n\nOutput:')
                print('Path is allowed in the departure days range of another 
Flight Number: ', path[0].Days)
                print('Allowed path:')
                for p in range(len(path)):
                    if p > 0:
                        print('step ',p ,':' ,'Use flight ',path[p-1].FN, ' 
Form ', path[p].parent.name , 'to ', path[p].name , ' ,Departure Time :', 
path[p].Departure , ' , Arrival time :' , path[p].Arrival  )
                break
            if isSourceNotEqualDestination:
                notFound = False
                break

        # if there is no flight number is allowed I will check day after and 
before the range
        if not notFound:
            indexS = days.index(allowedDays[0])
            indexE = days.index(allowedDays[-1])
            if indexS > 0 and indexS < 6:
                indexS = indexS - 1
            if indexE > 0 and indexE < 6:
                indexE = indexE + 1
            allowedDays = days[indexS:indexE+1]
            isDayExist = Check_Days( allowedDays , path[0].Days  )
            if isDayExist:
                print('\n\nOutput:')
                print('Path is allowed in the departure days of after and 
before the range available: ',allowedDays)
                print('Allowed path:')
                for p in range(len(path)):
                    if p > 0:
                        print('step ',p ,':' ,'Use flight ',path[p].FN, ' 
Form ', path[p].parent.name , 'to ', path[p].name , ' ,Departure Time :', 
path[p].Departure , ' , Arrival time :' , path[p].Arrival  )
            # if is not allowed after and before ranges I will print the path 
with available days
            else:
                print('No path avalaibal in allowed days !!')


def main():
    source = input("Please enter the source city: ")
    destination = input("Please enter the destination city: ")
    allowedDays = input("Please enter two days allowed for range (example 
Format: sun tue): ").split(' ')

    path = A_search(source, destination)
    if path != None:
        printPath(path , allowedDays)
    else:
        print('No Path avalaibe !!')

if __name__ == "__main__": main()
```