CS110 LBA - Armin Hamp

April 4, 2019

1 Activity Scheduler

Given the wide range of things that I can and I should do on a given day, it is really difficult to come up with an optimal schedule. For my scheduler, I will consider two types of activities: daily recurring chores, activities that I can freely choose.

1.1 Daily chores

1. Going to class. Taking three courses this semester, this averages out to a little less than 1 class per day on a given week.
2. Eating. I usually skip breakfast, and thus only eat lunch and dinner, but I must have these two meals.
3. Preparation for class.
4. Excercise. I never miss a day without yoga or going to the gym.
5. Talking to friends and family at home. Every day I call my mother, and often my friends.

1.2 Freely chosen activities

1. Exploring a new cafe.
2. Sightseeing. Going to see the Charminar, Golconda Fort, Royal tombs etc.
3. Grabbing a drink at a local pub.

1.3 Assumptions

1. Activities cannot overlap, i.e. I can only do one activity at a given time.
2. I did not include sleeping in the daily chores, but I do require 8 ours of sleep every day. So I will assume that the available time during the day will be 16 hours.
3. Every activity or chore has an associated category, starting and ending time, and an associated benefit with it.
4. The chores and activities above are the only type of activities I engage in, the only thing that can change about them is their length, timing and the benefit I receive from doing them.
5. I need to finish preparation for class before going to class.

Given these tasks and assumptions, I will write an algorithm that can maximise the benefit for my day and puts together a schedule.

2 Implementation

In [125]:
```
'''
Activity Scheduler.

I created a class called 'scheduling' with methods that can
carry out the optimisation method.

.optimal() is a strictly RECURSIVE and puts out maximum benefit but cannot store cho
.memoi_optimal() is a dynamic programming solution using MEMOIZATION and can store c
.find_schedule() is a tracaback of the dynamic programming solutions in O(n) time to

'''
from operator import itemgetter
#library to help sort input list of tasks by finish time

class scheduling(object):
        def __init__(self,tasks):
            self.data = [] #store name,start,finish and benefit of each task
            self.inorder(tasks) #orders tasks by finish time
            self.nonconflict = [None]*len(self.data) #storage vector for latest non-
            self.memoi = [None]*len(self.data) #storage vector for memoization

        def inorder(self,tasks):
            #read in input list and first sort it by finish time
            self.data=sorted(tasks, key=itemgetter(2))

        def non_conflict(self):
            #find latest non conflicting task for given task
            for i in range(len(self.data)):
                if i != 0:
                    for j in range(i-1, -1, -1):
                        if self.data[j][2] <= self.data[i][1]:
                            self.nonconflict[i] = j
                            break

        def optimal(self, j):
            #recursive solution to find max schedule
            if j is None or j < 0:
                return 0 #base case if no previous nonconflict
            else:
                #subproblem is: MAX(J+OPT(NONCONFLICT(J)),OPT(J-1))
                return max(self.data[j][3] + self.optimal(self.nonconflict[j]), self

        def memoi_optimal(self, j):
            #memoization solution same as recursive, but stores solutions to subprob
            if j is None or j < 0:
                return 0 #base case if no previous nonconflict
```

```
            elif self.memoi[j] is not None:
                return self.memoi[j] #MEMOIZATION STORAGE accessed if subproblem alr
            else:
                #store subproblems in self.memoi
                self.memoi[j] = max(self.data[j][3] + self.optimal(self.nonconflict[j
                return self.memoi[j]

        def find_schedule(self,j):
            #backtracking the memoization solution
            if j is None or j<0:
                return None
            #check which way does the subproblem J+OPT(NONCONFLICT(J)),OPT(J-1) go
            elif (self.data[j][3] + self.optimal(self.nonconflict[j])) > self.optimal
                #print name of j accordingly
                print(self.data[j][0], 'from',self.data[j][1], 'to', self.data[j][2])
                self.find_schedule(self.nonconflict[j])
            else:
                self.find_schedule(self.nonconflict[j-1])
```

2.1 Flexibility and Constraints

In [136]:
```
'''
CONSTRAINTS/FLEXIBILITIES

There are two features I want to incorporate in my scheduler:
1. given a time window, a chore could be carried in it at any tiem
2. don't allow certain activities to take place before others (e.g. class before stu
3. don't allow more than one of some activities to take place

1. I will achieve this by manipulating the input lists to my scheduler, so that
the input has an additional element, referring to duration.
2. Remove instances of class before prep
3. Randomly remove instances that do not make sense to have more than a certain
number of times (e.g it would be stupid to spend the whole day in a cafe)
'''
import random

def flexible(chores):
    #achieving flexibility by duplicating tasks as per duration and timeframe for sc
    input_list=[]
    for element in chores:
        j=0
        while (element[1]+j+element[3])<=(element[2]):
            input_element=[]
            input_element.append(element[0])
            input_element.append(element[1]+j)
            input_element.append(element[1]+j+element[3])
            input_element.append(element[4])
```

```
            input_list.append(input_element)
            j+=element[3]
    return input_list

def cleaner(tasks,string,n):
    index=[]
    for i in range(len(tasks)):
        if tasks[i][0]==string:
            index.append(i)
    to_delete=sorted(random.sample(index,len(index)-n))
    c=0
    for i in to_delete:
        del tasks[i-c]
        c+=1
    return tasks


def clean_up(tasks):
    chores=flexible(tasks)
    #find flexible tasks and remove elements according constraints in a random manne
    #1. no more than eating three times
    cleaner(chores,'eat',3)
    #2. no more than exercising two times
    cleaner(chores,'exercise',2)
    #3. no more talking than once
    cleaner(chores,'talk',1)
    #4. only one class
    cleaner(chores,'class',2)
    #5. no class before prep
    for i in range(len(chores)):
        if chores[i][0]=='class':
            index_class=i
    index_prep=[]
    for i in range(len(chores)):
        if chores[i][0]=='prep':
            index_prep.append(i)
    c=0
    for i in index_prep:
        if i>index_class:
            del chores[i-c]
            c+=1
    #6. only two cafes
    cleaner(chores,'cafe',2)
    return chores
```

2.2 Let's make a schedule

In [139]:
```
'''
 SCHEDULER IN ACTION

 '''
# tasks now have args:[name,start,finish,duration,benefit]
tasks=[['sleep',0,8,8,10.0],['prep',9,17,2,10.0],['class',11,17,2,11.0],['eat',8,24,
       ['talk',18,22,1,4.0],['cafe',10,16,2,7.0],['sight',12,20,4,8.0],['drink',21,24

s=scheduling(clean_up(tasks))
s.non_conflict()
print('The maximum benefit you can get for this day is:',s.memoi_optimal(len(s.noncon
print('Your schedule to follow is:')
s.find_schedule(len(s.nonconflict)-1)
```

```
The maximum benefit you can get for this day is: 69.5
Your schedule to follow is:
drink from 21 to 24
talk from 20 to 21
sight from 16 to 20
exercise from 13 to 14
class from 11 to 13
prep from 9 to 11
sleep from 0 to 8
```

3 Living a day according to the Scheduler

As there is some randomisation introduced to make the Scheduler flexible, every time it is run the input list to the scheduler changes. Accordingly, the schedule changes. I chose to live my day by first working run of the Scheduler. So my day looked like:

1. Sleep from 0 to 8
2. Eat from 8 to 9
3. Prep from 9 to 11
4. Class from 11 to 13
5. Excercise from 13 to 14
6. Cafe from 14 to 16
7. Go sightseeing from 16 to 20
8. Talk to my mother from 20 to 21
9. Have a drink from 21 to 24

In [140]:
```
'''
I did not take a picture for every activity, but most of them.
'''
from IPython.display import Image
#my breakfast
Image(filename='eat1.jpg')
```

Out[140]:

In [141]:
```
#my coffee
Image(filename='drink.jpg')
```

Out[141]:

```
In [142]: #my lunch
          Image(filename='eat2.jpg')
```

Out[142]:

```
In [143]: #my sightseeing
          Image(filename='sight.jpg')
```

Out[143]:

```
In [145]: #my friends and a beer
          Image(filename='cafe.jpg')
```

Out[145]:

4 Analysis of Code

4.1 Dynamic Programming

As seen in my code there are actually two solutions to this problem of interval scheduling with different weights or benefits associated with each task. After ordering tasks by their finish time, each of them find an OPTIMAL solution by recursively finding the maximum of two subproblems:

1. The benefit incurred from the j^{th} task plus the OPTIMAL solution to the $j's$ latest non-conflicting task.
2. The OPTIMAL SOLUTION to the task of index $j-1$.

We can see that a recursive solution to the problem is inefficient as the same subproblems will be solved multiple times. Thus a MEMOIZATION problem is straightforward, whereby we store all the answers to the subproblem of OPTIMAL(J), and access it any time that it is required again. While this takes a little bit of extra memory, it reduces the time complexity of the problem from O(2ˆn) to O(nlogn).

4.2 Things to improve

While currently the Scheduler works perfectly for the given task, improvements could be implemented, such as: 1. If tasks runs over time, it could reschedule from the input list of tasks that

start from the time I am free again. 2. The scheduler could automatically assess the priority (or benefit) of tasks by looking at what I have and have not done in previous days. 3. The scheduler could also offer multiple similar total benefit schedules, and leave it up to the user to decide what schedule seems most intuitive to their day and circumstances.

5 Additional HC Applications

#creativeheuristics: As I explained in my analysis of the code, the Dynamic Solution to the problem came to my mind after seeing how inefficient the recursive algorithm was. Then I used the heuristic of MEMOIZATION to provide a more efficient solution. I also explained the improvement in the algorithm from recursive to dynamic in terms of time complexity.