

ASSIGNMENT 3

ALGORITHMS & COMPLEXITY (CIS 522-01)

Javier Arechalde

February 21, 2018

1. Time-series data mining

1.1 Problem description

In this problem, we will be given two sequences of events, and we want to find out if the first sequence of events is a subsequence of the second given sequence. The events of the first sequence must appear in the same order in the second sequence too, but they don't necessarily need to be consecutive.

For example, we will be given the following sequence of events.

buy Yahoo, buy eBay, buy Yahoo, buy Oracle

We will also be given this other sequence, which may, or may not contain the first given sequence.

buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle

These two sequences of events will be named S' and S . Given these two sequences of events, S' of length m and S of length n each containing an event possibly more than once, we want to find in time $O(m+n)$ if S' is a subsequence of S , in the fastest way possible.

1.2 Proposed solution

Our proposed solution is to iterate over the two sequences S' and S simultaneously, to find if the first sequence is a subsequence of the second one. We will start by taking the first element in S' , then we will iterate over the S sequence to see if we can find that element. If we find that element in S , we move to the next element in S' , and start iterating to search for this second element in the sequence S , starting from the S position right after the position on which we found the first element of S' . If we can't find that element in S , that means we iterated over all S and reached the final position in S without finding it. In that case we will return that S' is not a subset of S . If we find all the elements of S' in S , then we will return that S' is a subset of S .

1.3 Pseudo code

Algorithm 1 Checking if S' subset of S

```
1: We initialize  $i_{pos} = j_{pos} = 0$ 
2: while We didnt reach the end of  $S$  or  $S'$  do
3:   Take  $S(i_{pos})$ 
4:   for  $j$  in range  $j_{pos} \rightarrow \text{length}(S)$  do
5:     if  $S'(i_{pos}) == S(j_{pos})$  then
6:       if  $i_{pos} == \text{length}(S')$  then
7:          $S'$  is a subsequence of  $S$ 
8:       end if
9:       if  $i_{pos} \neq \text{length}(S')$  then
10:         $i_{pos}++$ 
11:         $j_{pos} = j$ 
12:        Break the loop
13:      end if
14:    end if
15:    if  $S'(i_{pos}) \neq S(j_{pos})$  then
16:       $j++$ 
17:    end if
18:  end for
19: end while
```

1.4 Example

We implemented the algorithm in *Python*, if we run the script, the algorithm will iterate over the two sequences to find if the first sequence is a subsequence of the second sequence. If it is it will print that S' is a subsequence of S , and the opposite otherwise.

```
#These are the two sets that we want to compare
s1 = [ 'buy_Yahoo', 'buy_eBay', 'buy_Yahoo', 'buy_Oracle' ]
s2 = [ 'buy_Amazon', 'buy_Yahoo', 'buy_eBay', 'buy_Yahoo', 'buy_Yahoo', 'buy_Oracle' ]
```

```
def comp(s1,s2):
```

```
    #Getting the length of the two sets
    l1 = len(s1)
    l2 = len(s2)
```

```
    #We initialize the operators
    ipos = 0
    jpos = 0
```

```

while(jpos<l2-1):
    el1 = s1[ipos]
    for j in range(jpos,l2):
        el2 = s2[j]

        #If the element is in that position of the list...
        if el1==el2:
            if(ipos==l1-1):
                ipos = ipos+1
                print("S1 is a subset of S2")
                return
            elif(ipos!=l1):
                ipos = ipos+1
                jpos = j
                break #We exit the for loop

        #If we cant find the element in that position of the list
        #We move to the next element
        elif(el1!=el2):
            jpos = j
            continue

    #If the loop above didn't return that S1 is a subset of S2
    # we print the opposite
    print('S1 is not a subset of S2')
    return

comp(s1,s2)

```

1.5 Time complexity

In worst case scenario, the last element of S' will be in the last position of S , therefore, we would have iterated over both lists to check if S' is a subset of S . As the length of S' is m and the length of S is n , the time complexity of our implementation will be $O(m+n)$.

2. Competition scheduling

1.1 Problem description

In this problem, we want to host a competition. In this competition, we planned to do a mini-thriathlon, in which we will have to swim 20 laps of a pools, then bike for 10 miles, and then run 3 miles.

The competition must follow this rule: no more than one person can be swimming in the pool at a time.

Each one of the contestants has a projected *swimming time*, a projected *biking time*, and a projected *running time*. These are the times that will take each one of the contestants to complete each one of the different sections of the thriathlon.

Our goal is to design an efficient algorithm that produces an schedule whose competition's completion time is as small as possible.

We will name each one of the contestants c_i , and each one of this contestants will have a $\{st_i, bt_i, rt_i\}$ that will be the projected time for each of the sections of the thriathlon.

1.2 Proposed solution

This problem is a classic problem of minimizing lateness.

In this case, we will have one single resource, the pool, and n requests to use this resource for an interval of time.

Our implementation will consist in a *greedy* algorithm, that follows the *Earliest Deadline First* rule, that is sort the jobs in increasing deadline order, and schedule first the jobs that end earlier.

For this problem, the deadline will be the sum of the three projected times for all each contestant's thriathlon.

We will return a set of scheduled intervals for each contestant $sched_i = [s(i), f_1(i), f_2(i), f_3(i)]$ for each one of the contestants.

1.3 Pseudocode

Algorithm 2 Competition scheduling implementation

```
1: First we sort the students by completion time using QuickSort
2: function QUICKSORT(array)
3:   if Array length is 1 then
4:     return array
5:   end if
6:   if Array length > 1 then
7:      $pivot == array[0]$ 
8:     for Element in array do
9:       if  $element < pivot$  then
10:        We add the element to the lowerlist
11:       end if
12:       if  $element > pivot$  then
13:        We add element to the upperlist
14:       end if
15:       if  $d$  then
16:        Append element to the pivotlist
17:       end if
18:     end for
19:     upperlist = QUICKSORT(upperlist)
20:     lowerlist = QUICKSORT(lowerlist)
21:   end if
22:   Return lowerlist + pivotlist + upperlist
23: end function
24: Now we will start scheduling the students by increasing deadline
25: We initialize start time of swimming:  $s_{swim} = 0$ 
26: while We didn't schedule all students do
27:   Get the timings for the corresponding participant
28:   Time swimming:  $t_{swim}$ 
29:   Time biking:  $t_{biking}$ 
30:   Time running  $t_{running}$ 
31:
32:   Swimming start and finish
33:    $f_{swim} = s_{swim} + t_{swim}$ 
34:
35:   Biking start and finish
36:    $s_{bike} = f_{swim}$ 
37:    $f_{bike} = s_{bike} + t_{bike}$ 
38:
39:   Running start and finish
40:    $s_{run} = f_{bike}$ 
41:    $f_{run} = s_{run} + t_{run}$ 
42:
43:   We return the start and finish time for each one of the sections
44: end while
```

1.4 Example (Implementation)

Here we should prove that our algorithm is correct too.

```
#We will first create a list that contains the participants in the competition
#and their swimming time, biking time, and running time
contestants = [
    (1,2,3), #Contestant #1
    (2,3,10), #...
    (4,5,3) #Contestant #N
]

#Deadlines list
deadline = [0]*len(contestants)

#We calculate the total deadline for swimming for each one of the contestants
for i in range(0,len(contestants)):
    contestant = contestants[i]
    deadline[i] = sum(contestant)

print(deadline)

def quicksort(array ,array0):
    upperlist = []
    lowerlist = []
    pivotlist = []

    upper0 = []
    lower0 = []
    pivot0 = []

    #If the length of the array is 1 it doesnt need sorting of any kind
    if len(array)<=1:
        return array ,list(array0)

    #Else, we will separate the list into the upper elements and lower
    else:
        pivot = array[0]

        for i in range(0,len(array)):
            element = array[i]
            element0 = list(array0[i])

            if element<pivot:
                lowerlist.append(element)
                lower0.append(element0)
```

```

        elif element > pivot:
            upperlist.append(element)
            upper0.append(element0)
        else:
            pivotlist.append(element)
            pivot0.append(element0)

    upperlist, upper0 = quicksort(upperlist, upper0)
    lowerlist, lower0 = quicksort(lowerlist, lower0)

    return lowerlist + pivotlist + upperlist, list(lower0 + pivot0 + upper0)

#We will return an array with the indexes in the order that they need to be accessed
schedule, sortcont = quicksort(deadline, contestants)

#We initialize the variables
s_swim = 0

#Now we will start scheduling the events
for contestant in contestants:

    print("Contestant: ", contestant)

    #The time that will take the contestant to finish each one of the sections
    t_swim = contestant[0]
    t_bike = contestant[1]
    t_run = contestant[2]

    #Swimming time
    f_swim = s_swim + t_swim

    #Biking time
    s_bike = f_swim
    f_bike = s_bike + t_bike

    #Running time
    s_run = f_bike
    f_run = s_run + t_run

    print("Start Swimming: ", s_swim, "End Swimming: ", f_swim)
    print("Start Biking: ", s_bike, "End Biking: ", f_bike)
    print("Start Running: ", s_run, "End Running: ", f_run)
    print("\n")

#The next contestant will start swimming when the previous one is done
s_swim = f_swim

```


1.5 Time complexity