The task-scheduling problem

June 5, 2018

Student: Tudor Andrei

Computers and Information Technology
(Engligh Language)

Group 1.3 B

1st Year

Introduction

Assume there exist n activities with each of them being represented by a start time si and finish time fi. Two activities i and j are said to be non-conflicting if si fj or sj fi. The activity selection problem consists in finding the maximal solution set (S) of non-conflicting activities, or more precisely there must exist no solution set S' such that —S'— ¿ —S— in the case that multiple maximal solutions have equal sizes.

The activity selection problem is notable in that using a greedy algorithm to find a solution will always result in an optimal solution. A pseudocode sketch of the iterative version of the algorithm and a proof of the optimality of its result are included below.

Problem statement

Schedule several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Given that an activity has a start time and a finish time two activities are compatible if the intervals between theirs start and finish times don not overlap. Implement two different algorithms that resolve this problem.

Pseudocode

```
Greedy-Iterative-Activity-Selector(A, s, f):

    Sort A by finish times stored in f'

    S = {A[1]}

    k = 1

    n = A.length

    for i = 2 to n:

       if s[i]    f[k]:

           S = S U {A[i]}

           k = i

    return S
```

```
Weighted-Activity-Selection(S): // S = list of activities

  sort S by finish time

  opt[0] = 0

  for i = 1 to n:

        t = binary search to find activity with finish time <= start
            time for i

        opt[i] = MAX(opt[i-1], opt[t] + w(i))

        return opt[n]
```

Application design

For Greedy-Iterative-Activity-Selector we have:
Line 1: This algorithm is called Greedy-Iterative-Activity-Selector, because it

is first of all a greedy algorithm, and then it is iterative. There's also a recursive version of this greedy algorithm.
A is an array containing the activities.
s is an array containing the start times of the activities in A.
f is an array containing the finish times of the activities in A.

Line 3: Sorts in increasing order of finish times the array of activities A by using the finish times stored in the array f. This operation can be done, using for example merge sort, heap sort, or quick sort algorithms.

Line 5: Creates a set S to store the selected activities, and initialises it with the activity A [1] that has the earliest finish time.

Line 6: Creates a variable k that keeps track of the index of the last selected activity.

Line 10: Starts iterating from the second element of that array A. up to its last element.
Lines 11,12: If the start time s [i] of the i t h activity A [i] is greater or

equal to the finish time f [k] of the last selected activity A [k] , then A [i] is compatible to the selected activities in the set S, and thus it can be added to S.

Line 13: The index of the last selected activity is updated to the just added activity A [i].

Weighted activity selection problem:

The generalized version of the activity selection problem involves selecting an optimal set of non-overlapping activities such that the total weight is maximized. Unlike the unweighted version, there is no greedy solution to the weighted activity selection problem. However, a dynamic programming solution can readily be formed using the following approach: Consider an optimal solution containing activity k. We now have non-overlapping activities on the left and right of k. We can recursively find solutions for these two sets because of optimal sub-structure. As we don't know k, we can try each of the activities. This approach leads to an O (n 3) solution. This can be optimized further considering that for each set of activities in (i , j) , we can find the optimal solution if we had known the solution for (i , t) , where t is the last non-

overlapping interval with j in (i , j) . This yields an O (n 2) solution. This can be further optimized considering the fact that we do not need to consider all ranges (i , j), but instead just (1 , j).

Source Code

```
//-------------------------------------------------------------
#include<stdio.h>
#include<algorithm>
#include<vector>
using namespace std;

void printMaxActivities(int s[], int f[], int n)
{
    int i, j;
    vector<int> myvector (f, f+sizeof(f)-1);
    sort(myvector.begin(),myvector.end());
    printf ("Following activities are selected n: ");
    i = 0;
    printf("%d ", i);
    for (j = 1; j < n; j++)
    { if (s[j] >= myvector[i])
      {
          printf ("%d ", j);
          i = j;
      }
    }
}

int main()
{
    int startTime[] = {1, 3, 0, 5, 8, 5};
    int finishTime[] = {4, 2, 7, 6, 9, 9};
    int n = sizeof(startTime)/sizeof(startTime[0]);
    printMaxActivities(startTime,finishTime, n);
    return 0;
}
```

```
//-------------------------------------------------------------


  This implementation assumes that the activities are already sorted
      according to their finish time.
  Prints a maximum set of activities that can be done by a single
      person, one at a time.
  n   -  Total number of activities
  s[] -  An array that contains start time of all activities
  f[] -  An array that contains finish time of all activities

  The first activity always gets selected, then we consider rest of the
      activities.
```

If this activity has start time greater than or equal to the finish time of previously selected activity, then select it.

```
//-------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>

struct task   //structuri
{
    int start;
    int finish;
    int duration;
} taskuri[20],taskAux;

void bubbleSort(struct task arr[], int n)
{
    int i, j;                                //orodonare cu metoda
        bulelorr
    for (i = 0; i < n-1; i++)
        for (j = 0; j < n-i-1; j++)
            if (arr[j].duration > arr[j+1].duration)
            {
                taskAux.start = arr[j].start;         //aux=x;
                taskAux.finish=arr[j].finish;
                taskAux.duration=arr[j].duration;

                arr[j].start = arr[j+1].start;          //x=y;
                arr[j].finish = arr[j+1].finish;
                arr[j].duration = arr[j+1].duration;

                arr[j+1].start= taskAux.start;          //y=aux;
                arr[j+1].finish= taskAux.finish;
                arr[j+1].duration= taskAux.duration;
            }
            else
                if(arr[j].duration == arr[j+1].duration)
                {
                    if(arr[j].start > arr[j+1].start)
                    {
                        taskAux.start = arr[j].start;         //aux=x;
                        taskAux.finish=arr[j].finish;
                        taskAux.duration=arr[j].duration;

                        arr[j].start = arr[j+1].start;          //x=y;
                        arr[j].finish = arr[j+1].finish;
                        arr[j].duration = arr[j+1].duration;

                        arr[j+1].start= taskAux.start;          //y=aux;
```

```
                    arr[j+1].finish= taskAux.finish;
                    arr[j+1].duration= taskAux.duration;

                }

            }


}

int main()
{

    FILE *myFile;
    myFile = fopen("in.txt", "r");
    int numberArray[1000];
    int i,j;
    int n;
    int t;
    printf("time=");
    scanf("%d",&t);

    if (myFile == NULL) // daca avem cv de citit
    {
        printf("Error Reading File\n");
        exit (0);
    }

    fscanf(myFile, "%d,", &numberArray[0]);
    n=numberArray[0];
    j=0;
    for (i = 1; i <= n*2; i++)                              // citire din
        fisier
    {
        fscanf(myFile, "%d,", &numberArray[i] );
        if(i%2!=0)
        {
            taskuri[j].start=numberArray[i];
        }
        if(i%2==0)
        {
            taskuri[j].finish=numberArray[i];
            j++;
        }
    }

    for (i = 0; i < n; i++)                 // cat dureaza un task
    {
        taskuri[i].duration=taskuri[i].finish-taskuri[i].start;
    }
```

```
    bubbleSort(taskuri,n);        // apelam functia bublesort ce
        ordoneaza in functie de durata si de start

    for (i = 0; i < n; i++)
    {
        printf("task_%d: start_%d, finish_%d , duration_%d\n\n",
            i,taskuri[i].start,taskuri[i].finish,taskuri[i].duration );
    }

    int k=1;
    int aux;
    aux=taskuri[0].finish;
           for(i=0;i<=n-2;i++){             //ordonare
               for(j=i+1;j<=n-1;j++){
               if(aux<=taskuri[j].start)
               {
                   k=k+taskuri[j].duration;

                   aux=taskuri[j].finish;
                      i++;
               }
           }

}
printf("k=%d",k);
    fclose(myFile);

    return 0;
}
```

Conclusions

Working on this project, I have gained a new appreciation for these relatively simple data structures and their many uses in the information world.

References

1)www.stackoverflow.com

2)http://www.sharelatex.com