Homework-4

CSE-551 Foundation of Algorithms

1218506822

① Let S_{ij} = denote the set of activities, we have to determine the max activities A_{ij} which are compatible in S_{ij}

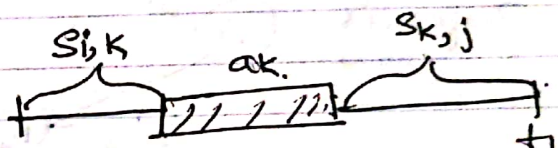
Example

	a_0	a_1	a_2	a_3	a_4	a_5	
S_i	0	2	1	3	6	∞	start time
f_i	0	0	3	4	8	∞	end time
Profit	4	83	6	7	8	10	

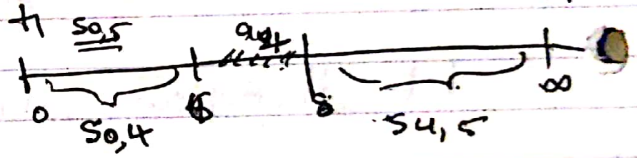
$S_{i,j}$ = set of activities which starts after 'a_i' finishes and finishes before a_j starts.

Select a_k activity i.e., act[i,j] which we consider to put in list of A_{ij}

S_{ij} is divided into three parts



Example :-



Hence.

$A_{ij} = S_{ij}\ \ S_{ik}\ \ S_{k,j}\ \ \{a_k\}$

$A_{ij} = A_{ik} \cup A_{k,j} \cup \{a_k\}$

$$A[i,j] = \max_{i<k<j} \begin{cases} A[i,k] + A[k,j] + 1 & S_{ij} \neq \phi \\ 0 & S_{ij} = \phi \end{cases}$$

* But given that profit is associated with each activity we ~~let p_k is the value associate~~.

Let p_k be the profit associated with each activity. on replacing 1 with p_k we have

$$A[i,j] = \max_{i<k<j} \begin{cases} A[i,k] + A[k,j] + p[k] & S_{ij} \neq \phi \\ 0 & S_{ij} = \phi \end{cases}$$

```
Profit-Maximizer (s, f, n)
{
    let A[0..n+1, 0...n+1] and act[0...n+1, 0...n+1]
        be the tables of
    and P[0...n+1] be the profit table associated
    with activity

    for i = 0 to n
    {       A[i,j] = 0
            A[i, i+1] = 0
    }       A[n+1, n+1]
    A[n+1, n+1] = 0

    for l = 2 to n+1
    {
        for i = 0 to n-l-1
        {   j = i+1
            A[i, j] = 0;
            K = j-1
            while f[i] < f[K]
            {   i-
                if f[i] <= s[K] and P[K] <= s[j]
                and A[i,K] + A[K,j] + P[K] > A[i,j]
                {
                    A[i,j] = A[i,K] + A[K,j] + P[K]

                    act[i,j] = K
                }
                K = K-1
            }
        }
    }
}
```

Print" The ~~Maximum~~ compatible set with Maximum profit" is A[0, n+1]"

Print-Activities (A, act, 0, n+1)

}

Print - Activities (A, act, i, j)
{
    If (A[i, j] > 0)
    {
        k = act[i, j]
        print k
        Print - Activities (A, act, i, k)
        Print - Activities (A, act, k, j)
    }
}

// For. 3 loops, the Time-Complexity of algorithm is $O(n^3)$

We can still optimize the Algorithm to O(nlogn) by following way:

Key: If we assume that we only look at the last job when, ~~nearly~~ sorted the jobs by finish time, then either we pick it or we don't.

Case1: (we pick it) Then we need to remove all overlapping jobs and Compute the solution for the subproblem of the jobs left.

case 2: (we don't pick it). Then we simply compute optimal solution of subproblem with n-1 items left

If OPT(j) is the set of all activities from 1 to j which returns max. weight

$OPT(j) = \max \{ profit[j] + OPT(P[j]), OPT(j-1) \}$

↓

returns set of non overlapping jobs.

P[] → stores index of closest non-overlapping job

```
int findMaxProfit ( Job array[], int n)
{
  Step1 : Sort jobs according to finish times

  Step2 : create an array to store values of
          subproblems profit
          maxProfit[0] = array[0].profit

  for ( i=1 to n)
  {
    //Step3 : Find profit of current job
    int curProfit = array[i].profit.

    Step4 : get nonconflict index by using binarysearch
    int l = binarySearch ( array, i) ;
    If ( l != -1)
    {
      // include curprofit with non-overlapping Job profit
      curProfit += maxProfit[l];
    }
    maxProfit[i] = max { curProfit, maxProfit[i-1] }.
}
```

```
    int result = maxProfit[n-1];
    return result;
}
```

Time Complexity – O(nlogn)

→ Sorting finaltimes – O(nlogn)

→ Finding non-conflicting Index ...O(nlogn) using binary search

→ Finding max weight O(n)

→ Backtracking to find optimal Items O(n).

2) * (a) False.

Sol:- If programs are stored in Increasing value of l_i.

Example :- let $P_1 = 0.2$, $P_2 = 0.8$, $l_1 = 4$, $l_2 = 12$

given $l_1 \to l_2$, $T_{12} = P_1 \times l_1 + P_2 \times (l_1 + l_2)$

$= 0.2 \times 4 + 0.8(16)$

$= 0.8 + 12.8$

$= 13.6$

Consider $l_2 \to l_1$ $T_{21} = P_2 \times l_2 + P_1 \times (l_1 + l_2)$

$= 0.8 \times 12 + 0.2(16)$

$= 9.6 + 3.2$

$= 12.8$

We see that $T_{21} < T_{12}$, Hence our assumption is false

Let's disprove this, Assume

$T_{12} < T_{21}$

$\Rightarrow P_1 \times l_1 + P_2 \times (l_1 + l_2) < P_2 \times l_2 + P_1 \times (l_1 + l_2)$

$\Rightarrow P_1 \times l_1 + P_2 \times l_1 + P_2 \times l_2 < P_2 \times l_2 + P_1 \times l_1 + P_1 \times l_2$

$(P_2) \times l_2 < (P_1) \times l_2$ (False) can be.

This Equation depends on P_1 & P_2, Hence we can't simply declare the $T_{12} < T_{21}$ all the times.

If Programs are stored in Increasing order of l_i, we can't minimize T

(b) False, If the Programs are arranged in decreasing order of P_i

Consider $P_1 = 0.4$, $P_2 = 0.3$, $l_1 = 25$, $l_2 = 5$

$T_{12} = 0.4 \times 25 + 0.3 \times (25+5)$ $(P_1 \rightarrow P_2)$

$= 10 + 9 = 19$

$T_{21} = 0.3 \times 5 + 0.4 \times (25+5)$

$= 1.5 + 12 = 13.5$

$T_{12} \not\geq T_{21}$ (Hence disproved)

Consider by Equation If $P_2 > P_2$: $T_{12} < T_{21}$

$$P_1 \times l_1 + P_2 \times (l_1 + l_2) \not< P_2 \times l_2 + P_1 \times (l_1 + l_2)$$

$$P_1 \times l_1 + P_2 \times l_1 + P_2 \times l_2 \not< P_2 \times l_2 + P_1 \times l_1 + P_1 \times l_2$$

$$P_2 \times (l_1) \not< P_1 \times (l_2)$$

The Equation depends on both l_1 & l_2, there the Time sequence is not guaranteed. and ~~can't be proved in both cases.~~

If the programs are stored in decreasing value of P_i, we can't minimize T

(c) Yes, If the Program stored in decreasing value of P_i/l_i it will minimize T.

Lets prove this Consider programs $P_1, P_2 \cdots P_i, P_{i+1} \cdots P_n$

If P_i is executed before P_{i+1},

$$T_1 = P_1 \times l_1 + P_2 \times l_2 \cdots + P_K(l_1 + l_2 \cdots + l_K) + P_{K+1}(l_1 + l_2 \cdots l_K + l_{K+1}) + \cdots + P_n(l_1 + \cdots l_n) \quad \text{— eq ①}$$

If P_{i+1} is executed before P_i,

$$T_2 = P_1 \times l_1 + P_2 \times l_2 \cdots + P_{K+1}(l_1 + l_2 \cdots + l_{K+1}) + P_K(l_1 + \cdots + l_{K+1} + l_K) + \cdots + P_n(l_1 + \cdots + l_n) \quad \text{— eq ②}$$

$T_2 > T_1 \Rightarrow T_2 - T_1 > 0$, when the time is minized

If considering the decreasing order of $\frac{P_i}{l_i}$

$$T_2 - T_1 \Rightarrow (P_1 \times l_1 + P_2 \times l_2 + \cdots P_K(l_1 + l_2 \cdots + l_K) + P_{K+1}(l_1 + l_2 \cdots + l_K + l_{K+1}) + \cdots + P_n(l_1 + \cdots l_n)$$
$$- (P_1 \times l_1 + P_2 \times l_2 + \cdots P_{K+1}(l_1 + l_2 \cdots + l_{K+1}) + P_K(l_1 \cdots + l_{K+1} + l_K) + \cdots + P_n(l_1 + \cdots + l_n)$$

$$T_2 - T_1 = P_K \times l_{K+1} - P_{K+1} \times l_K$$

$$P_K \times l_{K+1} - P_{K+1} l_K > 0$$

$$\frac{P_K}{l_K} \geq \frac{P_{K+1}}{l_{K+1}} \quad \text{(Hence proved)}$$

Therefore If the programs are stored in decreasing values of P_i/l_i it can guarantee the minimum retrieval time.

CSE-551 FOUNDATION OF ALGORITHMS

12185068 22

3

Sol:- (a) The Brute solution

(1) For given interval, Create & Store all permutations of activities(or) list provided. i.e., multiple sequence orders. we will use these sequences as input to activity interval coloring algorithm without any sorting.

(2) We will allocate resource allocation in the increasing order of c_1, c_2, c_3, c_n for the mutually exclusive compatible activities

(3) We will determine the cost for each permutation ~~By calculating, minimum cost~~. by taking minimum of all possibilities. would be the solution

(4) This may result in Exponential complexity by Brute fore $O(n \cdot n!)$

(b) By dynamic programming, the above solution where we can obtain multiple permutation can be split into subproblems. Hence, polynomial time co be possible of analyzed.

There can be strategies which may not provide optimal solution but feasible solution depending on the Input Instances.

1st strategy can be computing the longest independent set of graph formed by start & finish times. Put the set of jobs in one processor and remove corresponding nodes from graph. Then compute longest Independent set of remaining graph. So, on. until the all the Jobs are assigned to some processor.

Given $C_1 \leq C_2 \leq C_3 \cdots \leq C_n$, J_1, J_2, J_n -- with start & finish times

Algorithm ProcessorAllocation.
begin
    A = { $a_1, a_2 \cdots a_n$ } // Set of Activities
    P be the set of processors. P[n], C be the cost of processor.
    do until (A = ϕ)
    begin
        using Activity Selection Algorithm to set compatible activities or Jobs cA.
        Assign a processor with lowest cost to longest independent set cost = cost + |A| * P[i]
        A := A − cA ; i ← i+1
        no. of activities in cost of independent set processor
    end
    return cost.
end

Algorithm Active Selection (AAS)

```
begin
    n ← |A| where A = {a1, ... an}
    CA ← {a1}
    i ← 1
    for j = 2 to n do
        if sj ≥ fi then
            begin
                CA ← CA ∪ {aj}
                i ← j
            end
    return CA
end
```

Strategy 2 : Can be finding the chromatic number, partition of graph. As discussed in class, partition the node set of graph formed by start & finish times of activities into fewest no. of independent sets.

Begin
Sort jobs by start time, $s_1 \leq s_2 \leq s_3 \cdots \leq s_n$.
let P be the set of processors.

For j=1 to n
    If (job j is compatible with some processor)
        Schedule job j in any such processor.
    else
        Allocate a new processor P_{q+1}
        Schedule job j in processor P+1
        P ← P+1

end. ~~cost ← cost +~~ PT
Return Schedule

the schedule contains minimum number of ~~jobs~~ processors with jobs assigned under it.

Calculate the cost by processor and number of jobs assigned to it respechvely and return the cost

~~Cost = C[i] * P[i]~~ + n = no. of processon.

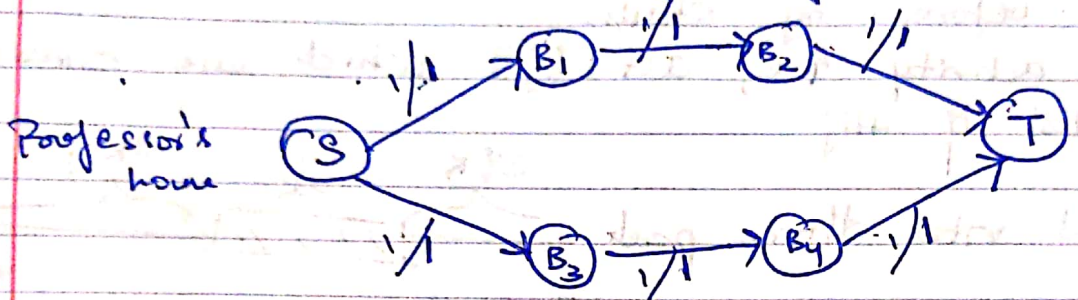
```
for (i=0 ; i<n ; i++)
  {
    cost = c[i] * P[i] + cost;
// c[i] = cost of processor
// P[i] = no. of jobs in processor
  }

return cost.
```

Strategy3: Idle can still find the minimum of sResult returned by Strategy1 & Strategy2. for more feasible results.

(4) Sol:- Given:- the children refuse to walk together and also they don't step on same blocks ~~that~~ th on same day at any time that one has stepped one. They can cross corners. Which is source & destination.

Considering the above requirements, we can have map in the following ways:



Let the Max flow value be 2, for 2 children;

Consider the weight on each edge is 1 by setting the capacity of each edge to 1

→ The Example Network flow has 2 disjoint paths satisfying the conditions and also the max flow '2' making it possible for the two children to reach their destination without any conflict.

Yes, the children can go to same school when capacity of each edge in one & Max flow is 2.