CS430 Homework 5

Chapter 6

Problem-1:

a)

Give an example to show that the following algorithm does not always find an independent set of maximum total weight.

The "heaviest-first" greedy algorithm

Start with S equal to the empty set

While some node remains in G

Pick a node vi of maximum weight

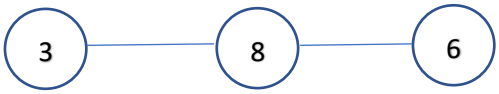
Add vi to S

Delete vi and its neighbours from G

End while

Return S

Answer:



The heaviest-first" greedy algorithm chooses the node with value 8 which is not maximum total weight.

The better solution is to select the node first with value 3 and last with value 6 which has total value 9.

b)

Give an example to show that the following algorithm also does not always find an independent set of maximum total weight.

Let S1 be the set of all vi where i is an odd number

Let S2 be the set of all vi where i is an even number

(Note that S1 and S2 are both independent sets)

Determine which of S1 or S2 has greater total weight,

and return this one

Answer:

The above given algorithm choose 2nd node with value 4 and 4th node with value 8, which have combined total weight 12 greater compared to 11(1st and 3rd node) .

Total weight 12 return by above algorithm is not maximum weight. The best solution is to select 1st and last node whose combine weight is 14 which is maximum as well.

c)

Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

Answer:

We'll define subproblem i to be the problem of choosing the highest-valued set of nodes from among the first i nodes (labelled from left to right).

For i >2,

We can choose either to include node i in solution, or not.

CASE-I :

If we include node i then we cannot include node i − 1, so we can get the best value when considering just the first i − 2 nodes.

CASE-II:

If we don't include node i then the best we can do is whatever our best solution was for the first i − 1 nodes.

We end up with the recurrence:

$$V(i) = \max \{V(i-1), wi + V(i-2)\}$$

We can compute value of V(i) in increasing order from i=1 to n.

Algorithm:

Def function independent_set_max (G)//G = (V, E)

1. Let W list of weight for the nodes
2. set W = [0] + W

   // a new array to store best values
3. n = len(W)
4. set V = [0] * (n)

```
// a new array to store best values
5. set V [0] = 0; v[1] = w[1];
6. for i = 2 to n
        if wi + V [ i - 2]  >  V [ i – 1 ]
            set V[i] = w[i] + V [ i – 2]
    else
        set V[i] = V[i-1]
    // The set of nodes to include

Return V[n-1]
```

Running Time:

Above algorithm involves a for loop runs for n iterations and in each iteration we have constant time work, so loop run in O(n) time.

Reconstruction of nodes also take O(n) time.

So overall Running Time is O(n).

Problem-2

Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

a)

```
For iterations i = 1 to n
If hi+1 > Li + Li+1 then
    Output "Choose no job in week i"
    Output "Choose a high-stress job in week i + 1"
    Continue with iteration i + 2
Else
    Output "Choose a low-stress job in week i"
    Continue with iteration i + 1
Endif
End
```

Answer:

WEEK	Week - 1	Week - 2	Week - 3	Week - 4
Low Stress	10	1	10	10
High Stress	5	20	50	10

As per the given Algorithm maximum value would be to choose “none” in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value return by algorithm would be 0 + 20 + 10 + 10 = 40.

Unique optimal solution would choose low stress job in week1, none in week-2 follow by high stress job in week-3 and low stress job in week 4. Then value return by optimal solution would be 10 + 0 + 50 + 10 = 70.

b)

For given a sequence of n weeks, a plan is specified by a choice of “low-stress,” “high-stress,” or “none” for each of the n weeks.

Each week we are allowed to pick either a low stress job or a high stress job

However, if “high-stress” is chosen for week i > 1, then “none” has to be chosen for week i − 1.

Our goal is to maximize your total income.

The value of the plan is determined in the natural way: for each i, you add li to the value if you choose “low-stress” in week i, and you add hi to the value if you choose “high-stress” in week i.

Give an efficient algorithm that takes values for l1, l2, . . . , ln and h1, h2, . . . , hn and returns the value of an optimal plan.

Answer:

OPT(i) : Maximum value we get by working for week 1,2,….i only.

If It choose low stress Job in week i, it can behave optimally up to week i-1, followed by this job

If it selects the high-stress job in week i, It has to take week i − 1 off, but it can behave optimally up to week i-2, followed by this job

So, we have recurrence for OPT(i):

OPT (0) = 0

OPT (1) = max (L1 , H1)

OPT(i) = max (Li+OPT(i - 1), Hi + OPT (i – 2))

Algorithm:

1. Function Max_income (low, high)
   //Low and high are list to store weekly revenue for high stress and low stress Job
2. n = Length(high)
   //get length of low list or high list
3. low = [0] + low // append element 0 at the beginning of the array
4. high = [0] + high // append element 0 at the beginning of the array
5. optimal = [0] * (n+1)
   // Initialize optimal [0..n] with all zeros
6. optimal [1] = max (low [1], high [1])
7. for i =2 to n+1
   Low-total = low[i] + opt[i-1]
   High-total = high[i] + opt[i-2]
   optimal[i] = max (low-total, high-total)
8. return optimal[n]

Running Time:

Loop run for n iteration and each iteration take constant time so total time is O(n).

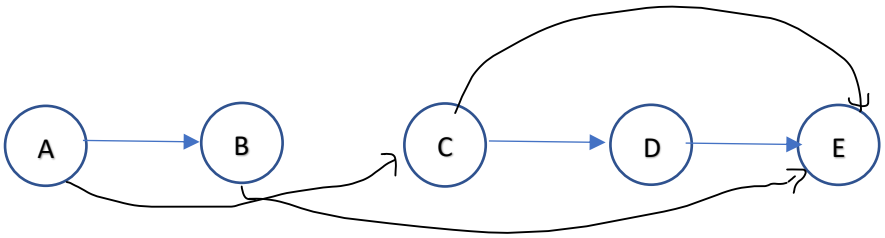
Sequence of job can be reconstructed by tracking back through the optimal value.

Problem-3

a)

Answer:

For Directed Graph G(V, E) ,V={A,B,C,D,E} and E={ (A,B) ,(A,C),(C,D),(C,E),(D,E),(B,E) }



As per the Given Algorithm will find longest path from source node A to node E whose value is 2 and use the edges (A, B) and (B, E).

However, path return by given algorithm is not longest path, optimal solution is 3 using the path (A, C), (C, D) and (D,E).

b)

Let G = (V, E) be a directed graph with nodes v1, . . ., vn

OPT(i): Length of the longest path from V1 to Vi

OPT (i) = -1 if there is no path from V1 to Vi

OPT (1) = 0 for longest path from node V1 to V1 itself

Algorithm:

1. Function Longest Path(G)
2. for i= 1 to n //**n** no of vertices

    opt[v] = -1;

3. opt [1] = 0;
4. for i=1 to n //**n** no of vertices

    if (opt[i] >= 0)

        For each vertices j reachable from vertex i

            opt[j] = max(opt[i]+1, opt[j])

5. return opt[V]

Running Time: O(n^2)

For n number of nodes, each node needs to explore all the corresponding edges related to it. Thus, there are n*E number of iterations, this complexity can be approximated to O(n^2).

Problem-4

Given a long string of letters y = y1y2 . . . yn, a segmentation of y is a partition of its letters into contiguous blocks of letters. Each block corresponds to a word in the segmentation.

The total quality of a segmentation is determined by adding up the qualities of each of its blocks.

We need to compute the optimal segmentation of the input string into words such that the sum of these words' qualities is the highest possible.

Segmentation y1, y2……, yn is optimal for string Y then segmentation y1, y2,……. yn-1 also an optimal segmentation for prefix of string Y.

Input sequence y1y2...yn of length n,

function quality that given any sequence of letters returns the quality of the sequence

Optimal segmentation function:

OPT(i): optimal segmentation of prefix containing first i character of String Y.

For 0<= i <= n

If i=0 OPT (0) = 0

If i>0,

$OPT(i) = \max_{1<=k<=j} \{OPT(k-1) + Quality\ (Y_k \ldots\ldots Y_n)\}$

Function segment the string into 2 parts, first part consist one word and 2nd part is renaming string which is segmented recursively.

Algorithm:

Input string S [y1y2……..yn] of length n

Quality Function: returns the int value shows quality of a given string S.

1. Function compute (S)
2. OPT [0]: = 0
3. n = len(S) //(length of String)
4. for i: = 1 to n

        temp_opt=0

        for k = 1 to i

            if OPT[k-1] + quality (yk...yi) > temp_opt

              //check optimal quality of substring

                temp_opt: = OPT[k-1] + quality (yk...yi))

      OPT[i]: = temp_opt
5. End For
6. Return OPT[n-1]

Running Time: o(n*n)

We are using 2 for loops first loop iterate for n times and 2nd loop iteration depend on 1st loop (1 to I times)All statements inside loop takes constant time for execution. So, Overall Time complexity for Algorithm is $O(N^2)$

Problem-5

Consider a firm that trades shares in n different companies. For each pair i not equal j, they maintain a trade ratio rij, meaning that one share of i trades for rij shares of j.

A trading cycle for a sequence of shares i1, i2, . . ., ik consists of successively trading shares in company i1 for shares in company i2, then shares in company i2 for shares i3, and so on, finally trading shares in ik back to shares in company i1.

Answer:

Graph G (V, E)

Trading Cycle **C** is an opportunity Cycle if trading along the cycle increases the number of shares.

This happens exactly if the product of the ratios along the cycle is above 1.

Implied $\pi\, r_{ij} > 1 \rightarrow \pi\, (1/ r_{ij}) < 1$----------(I)

So by taking logarithm in above Equation **(I)**

$$\sum (1/ r_{ij}) < 0 \rightarrow \sum -\log(r_{ij}) < 0$$

Implied Trading cycle C is an opportunity cycle if and only if it is negative Cycle.

Bellman-Ford algorithm allows you to check whether there exists a cycle of negative weight in the graph, and if it does, find one of these negative cycles.

Algorithm:

1. For all vertices v of Graph G (V, E)
2. d[v]= ∞ //set initial distance
3. d[s]=0 //distance to start node
4. For k=1 to n-1
5. For each edge (u, v) with weight w in set E

   d[v]= min {d[v], d[u]+w (u, v)}

   //update the cost of v

   //Trading Cycle C -Negative cycle detection

6. **For each edge (u, v) in set E**

   d[v] > d[u] + w (u, v)

   print "Negative Cycle/Opportunity Cycle "

7. **Return d[t] for all t in set V**

Algorithms detect Negative Cycle if there is a negative cycle reachable from the source vertex S, then for some edge (u, v), $dn-1(v) > dn-1(u) + w(u, v)$.

If Graph has no Negative Cycle, distance calculated from last iteration are equal to true shortest distance.

Running Time: O(n*n)

The first for loop relaxes each of the edges in the graph n − 1 times. So Overall the algorithm takes O(mn) time[O(VE)]