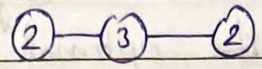
Aaryan Ajay Sharma 202212001

Algorithm Analysis & Design

04/11/2022

Problem set 3

1. (a) Example: 

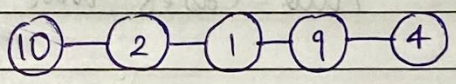
G_1

In the above example,

The greedy "heaviest" first algorithm finds ③ to be the max-weight independent set but ②,② = 4 is actually the max-weight independent set.

∴ The ~~heavy~~ heavy-first greedy approach does not always give the optimal answer.

(b) Example:



Our algorithm clubs all odd & even vertices with each other & chooses the set with greater weight as MIS.

In the above example, the weight of even & odd sets are

even → {10, 1, 4} → 15

odd → {2, 9} → 11

so our algorithm say MIS has weight 15.

But we can see that by choosing 10 & 9, we get MIS as 19.

∴ This algorithm too, does not yield the optimal solution everytime.

(c)

Let $Opt(i)$ be the ~~wig~~ weight of the MIS formed with vertices $V_1, V_2, \ldots, V_i$. whose weights are $w_1, w_2, \ldots, w_i$ respectively.

Now, let S be the set of vertices in MIS of $G_i = \{V_1, V_2, \ldots, V_i\}$.

Two cases are possible:

Case 1: $V_i \in S$

If $V_i \in S$, the $V_{i-1} \notin S$ since two adjacent vertices are in S.

$$\therefore \quad Opt(i) = opt(i-2) + w_i$$

Case 2: $V_i \notin S$

If $V_i \notin S$, then $Opt(i)$ will simply be equal to

$$Opt(i) = Opt(i-1)$$

Since we do not know whether $v_i \in S$ or not, we consider both of the cases and take the max of the two.

Doing so, we are considering all possible Independent sets and ~~re~~picking the one with max value.

$$\therefore \quad Opt(i) = \max \begin{cases} opt(i-2) + w_i \\ opt(i-1) \end{cases}$$

with $opt(0) = 0$ & $opt(1) = w_1$ as the base condition.

Algorithm:

We'll walk through the path from left to right, using the memoized of minimum weight so far to decide ~~which~~ which property must ~~be~~ apply (either the last vertex is a the part of the set or not)

Initialize Array A of size $n+1$

$A[0] = 0$, $A[1] = w_1$.

for i in $n-1$

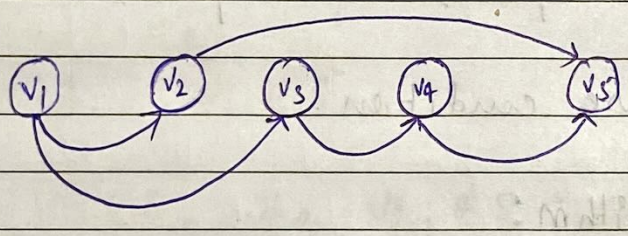
$A[i+1] = \max(A[i], w_i + A[i-1])$

```
set S ← ϕ.

i = n

While i >=1 :
    if A[i] = A[i-1]   # least vertex not in set
        i--
    else                # least vertex in set
        S.insert(i)
        i -= 2
    end
end
return S;
```

2. (a)



The algorithm will give the length of the longest path as 2, as it will choose the path $v_1 \to v_2 \to v_5$, but there is a longer path, $v_1 \to v_3 \to v_4 \to v_5$ of length 3 which is the longest.

∴ The specified algorithm doesnot correctly solve the problem.

__/__/___

(b)

(b) Let opt(i) be the length of the longest path from v_1 to v_i.

and let opt(1) = 0

If there is no path from v_1 to v_i, set opt(i) = $-\infty$

Now, opt(i) can be written as follows:

$$opt(i) = 1 + \max_{1 \le j < i} (opt(j)) \text{ if } \exists \text{ path from } v_j \text{ to } v_i.$$

i.e.

we choose the ~~vertic~~ vertex which has the longest path from v_1 just preceding i which also has path ~~from v_j~~ to v_i.

∴ we now need to find opt(n).

We can do this iteratively by creating an array & building the solution from 1 to n

Algorithm / Pseudo code:

```
Longest-path(n):
  A[1,...n] ← -∞
  A[0] = 0
```

~~for i in 2 . . n~~
~~if M[i] ≠ -∞ && A[i] < A[j]+1~~

for i in 2 . . . n

for all edges (j, i):
if M[j] ≠ -∞ & M[i] < M[j]+1:
M[i] = M[j]+1

Return M[n].

The run-time is $O(n^2)$ as the max. number of edges can be $1+2+..+n-1 = \frac{n(n-1)}{2} = O(n^2)$

and assuming all of them can/will be traversed

3. Let opt(i) denote the min. cost of a solution till week i.

For week i either company A or B will be picked.

Case 1: Company A is picked.

then opt(i) = rc_i + opt(i-1)

~~Case~~ Case 2: company B is picked

then opt[i] = 4c + opt(i-4)

we do not know which of the two is minimum, therefore we choose the minimum of the two.

$\therefore$ $opt(i) = \min(rs_i + opt(i-1), 4c + opt(i-4))$.

for $i \leq 0$, let $opt(i) = 0$.

$\therefore$ $opt(n)$ can be found be found by building up the opt values in increasing order of i, assuming each iteration takes constant time.

We can obtain the optimal schedule by ~~tracing~~ tracing back through the arrays of opt values.

4. Let $opt(i)$ be the total cost of the optimal ~~conf~~ configuration from server 1 to i, given there is a copy of the file at server j.

Let server $\hat{j}$ be the place where the copy of the file is located, which is an immediate predecessor of server j.

$\therefore$ The total cost of the optimal configuration, ~~cost~~ $opt(i)$, can be written as follows:

$opt(i) =$ placement cost at j + access time cost from j to i.

placement cost at $i = c_i$

and access time cost from $j+1$ to i.

$$= \underbrace{0}_{\text{for } i} + \underbrace{1}_{\text{for } i-1} + \underbrace{2}_{\text{for } i-2} + \cdots + \underbrace{i-j}_{\text{for } j+1} = \frac{(i-j)(i-j-1)}{2}$$

We do not know the j might be,

$$\therefore \quad opt(i) = c_i + \min_{0 \le j < i} \left(opt(j) + \frac{(i-j)(i-j-1)}{2} \right).$$

with $opt(0) = 0$.

The values of opt can be built in a bottom up manner in $O(i)$ for i^{th} iteration leading to a total run time of $O(n^2)$.

We can also find the optimal configuration by tracing through the values of opt in the array that stores it.

5. The following algorithm checks the validity of the given $d(v)$'s in $O(m)$ time:

If for any edge $e = (v, w)$, we have

$$d(v) > d(w) + c_e$$

we can immediately reject.

This follows since if $d(w)$ is correct, then there is a path from v to t via w of cost $d(w) + c_e$. The minimum distance from v to t is at most this value, so $d(v)$ must be incorrect.

Now, consider the graph G' formed by taking G and removing all edges except $e = (v, w)$ for which $d(v) = d(w) + c_e$.

We will now check that every node has a path to t in this new graph.
This can be easily checked in $O(m)$ by reversing all edges and doing a DFS or BFS.

If any nodes fails to have such a path, reject it and accept otherwise.

Observe that if $d(v)$ were correct for all nodes v, then if we consider those edges on the shortest path from any node v to t, these edges will all be in G'. Therefore we can safely reject if any node can not reach t in G'.

So far, we have argued that when our algorithm rejects a set of distances, those distances must be incorrect.

We now need to prove that if our algorithm accepts, then the distances are correct. Let $d'(v)$ be the actual distance in G from v to t. We need to show $d'(v) = d(v)$ $\forall v$.

Consider the path found by our algorithm in G' from v to t.

The real cost of this path is exactly $d(v)$. There may be shorter path, but we know that $d'(v) \leq d(v)$, and this holds $\forall v$.

Now suppose that there is some v for which $d'(v) < d(v)$.

Consider the actual shorter path P from v to t. Let x be the last on P for which $d'(x) < d(x)$. Let y be the node after x on P. By our choice of x,

$$d'(y) = d(y)$$

Since P is the real shortest path to t from v, it is also the real shortest path from all nodes on P.

So $d'(x) = d'(y) = c_e$ where $e = (x, y)$

This implies that $d(x) > d'(x) + c_e$. !

But then we have a contradiction, since our algorithm would have rejected upon ~~reject~~ inspection of e.

$\therefore \quad d'(v) = d(v) \quad \forall \quad v.$

The same solution can be phrased in terms of the modified cost function that will be given in part (b) of the solution.

There are two common mistakes:

1) To simply use the algorithm that just checks for $d(v) > d(w) + c_e$.
This can be broken if the graph has a cycle of cost 0, as the nodes on such a cycle may be self consistent, but may have a much longer distance to the root than reported.

Consider two nodes x & y connected to each other with 0 cost edges, and connected to t by an edge of cost 5.
If we report that both of these are at distance 5 from the root ~~...~~, the algorithm will ~~fail~~ wrongly accept the distances.

2) To fail to prove the algorithm output is correct on both yes & no instances of the problem.

b) Given distances to a sink t, we can efficiently calculate distance to another sink t' in $O(m \log n)$ time.

To do so, note that if only edge costs were non-negative, then we could just run ~~Dji~~ Dijkstra's algorithm.

We will modify costs to ensure this, without changing the min cost paths.

Consider modifying the cost of each edge $e = (v, w)$ as follows:

$$c_e' = c_e - d(v) + d(w)$$

First observe that the modified costs are non-negative, since if $c_e' < 0$ then $d(v) < d(w) + c_e$ which is not possible if d reflect true distances to t.

Now, consider any path P from some node x to t'. The real cost of P is

$$c(P) = \sum_{e \in P} c_e.$$

The ~~cost~~ modified cost of P is ~~c'(P)~~

$$c'(P) = \sum_{e \in P} c'_e = \sum_{e \in P} c_e - d(v) + d(w)$$

Note that all but the first and last node in P occur once positively & once negatively in this sum, so we get that $c'(P) = c(P) - d(s) + d(t')$.

Hence the modified cost of any path from s to t' differs from the real cost of that same path an additive

$d(t') - d(s)$, a constant. So the set of minimum cost paths from s to t' under c' is the same as the set under c.

Furthermore, given the min. distance from s to t' under c, we can calculate the minimum distance under c by simply adding $d(s) - d(t')$.

Our algorithm is exactly that:

1) Calculate the modified costs
2) Run Dijkstra from t'
3) Adjust the distances as described.

The edges need to be reversed for standard implementation of Dijkstras

__/__/___

The time complexity of the algorithm is

$O(m + m \log n + n) = O(m \log n)$ time.