Assignment 3 Solutions

November 15, 2017

Fiber Connectivity

We can visualize a graph G=(V,E) for our problem, where V=C (the set of vertices is our set of cities) and E=R (the set of edges is our set of roads). We have an edge $e : (u, v)$ between two nodes u and v in the graph G only if we have a road between the corresponding cities u and v in our original problem. We now rephrase the question as: given this graph G, is there a subset of edges $T \subseteq E$ such that $|T| \leq |V| - 1$ and $1 \leq d_T(u) \leq k$ for all nodes $u \in V$, and all nodes in V are connected together. Recall $d_T(u)$ is defined as the number of edges that node u is connected to using only edges in T.

First thing to note about our problem, *Fiber Connectivity*, is that the only way to choose a set of edges that satisfy the requirements that 1) at most $|V| - 1$ edges can be used, and 2) all $|V|$ nodes need to be connected by the chosen edges, is to pick a set of edges that form a tree, i.e. a Spanning Tree of the graph G. In fact, our problem is a disguised well-known problem, called *degree-constrained spanning tree*, asking whether a graph has a spanning tree where every vertex has degree at most k.

NP membership: First, we will show that *Fiber Connectivity* belongs to the class NP. Suppose we are given a candidate solution, say T. You have to confirm in polynomial time if

- T is a spanning tree: (1) $|V| - 1$ edges are used, (2) all nodes are connected, and
- All vertices have degree less than or equal to k.

To check if it is a tree you can run any graph traversal algorithm (DFS, BFS, etc) and see if you have a loop or not. This takes polynomial time in $|V| + |E|$.

Then you should check if T connects all vertices. This is simple too. You can check whether the number of edges of T is equal to $|V| - 1$ or not. Any tree with $|V| - 1$ edges must have V nodes.

Then, you can check that all vertices in T have degrees $\leq k$ by examining each vertex one by one. All in all, we can check if the candidate is a correct solution to the decision problem or not in polynomial time in $|V| + |E|$.

Reduction: Next, we show *Fiber Connectivity* problem is NP-Complete by reducing a known NPC problem, the *Hamiltonian Path* problem (HamPath), to it. Given an instance of HamPath, we will convert it to an equivalent instance of *Fiber Connectivity* in polynomial time. In HamPath, we are given a connected graph G, and are asked whether it contains a simple path that visits all vertices of the graph. The key point inspiring this reduction is that a Hamiltonian path is a spanning tree with all degrees of vertices smaller than or equal to 2. So the transformation of a HampPath instance to a *Fiber Connectivity* instance is trivial: use the same graph G as the one given in the original HamPath and set the degree bound $k = 2$, i.e. is there a spanning tree in G where each vertex has degree at most $k = 2$.

Poly-time of reduction: Trivially so, since we just use the same graph.

Correctness of reduction: HamPath is a YES-instance if and only if the corresponding Fiber Connectivity is a YES-instance.

($\Rightarrow$) If the HamPath instance is a YES instance, then it has a path P that visits all vertices. Every path is also a tree (no loops). Hence P is a tree that visits all vertices, i.e. P is a spanning tree. In addition, since P is a path, the start and end vertices have degree 1 and all others have degree 2, hence all vertices have degree at most 2. Therefore, P is also a solution for our *Fiber Connectivity* instance.

($\Leftarrow$) If the *Fiber Connectivity* instance has a solution T, then this solution has vertex degrees at most 2 and no loops. Therefore T must be a path. T also must visit all vertices. Hence T is a path that visits all vertices, and therefore is also a solution to the orginal HamPath instance.

If we had a poly-time algorithm to solve *Fiber Connectivity*, then we would also have a poly-time algorithm to solve HamPath using our reduction. But HamPath is an NP-Complete problem, hence *Fiber Connectivity* is also NP-Complete.

In Hartford, Hereford, and Hampshire...

Let's first restate the `PYGMALION` decision problem:

> Given an undirected graph $G = (V, E)$, where nodes represent towns and edges represent roads, and given a number k, is there a way to build k bunkers at k different towns so that every town either has its own bunker, or is connected by a (direct) road to a town that does have a bunker?

As always, we start by showing that `PYGMALION` is in NP:

A candidate solution will be given as a subset of the vertices, $C \subset V$. To verify that C is indeed a solution, we first check that the number of vertices $|C|$ does not exceed k. Then, for each vertex in the graph, $v \in V$, we check if it is either in the given set $v \in C$ (i.e. $O(|V|)$, or one of its edges has an endpoint in that set (this can be done in $O(|V| + |E|)$ by hashing the vertex set and then looping over the set of edges) . Doing that will take at most $O(|V| + |E|)$ comparisons, which means that the solution can be verified in poly-time. Therefore, `PYGMALION` is in NP.

Now, we need to show that `PYGMALION` is NP-complete.

To show that `PYGMALION` is NP-Complete, we will follow the standard steps for showing NP-Completeness by reducing from Vertex Cover.

1. Show that `VERTEX-COVER`(G, k) can be **reduced** to `PYGMALION`(G', k'), and that the reduction can be done **in polynomial time**:
   Remember: A vertex cover is a set of vertices such that for every edge in the graph, at least one of its endpoints is in the vertex cover. In `VERTEX-COVER`(G, k), we are given a graph $G = (V, E)$ and a target number k, and we want to find whether there exists is vertex cover of size at most k. Using that, we can construct the input for `PYGMALION`(G', k') as follows: Let $k' = k$. We now want to construct a new graph $G' = (V', E')$. First, G' has all the edges and vertices of G. In addition, for every edge $\{u, v\} \in E$, we create a new vertex w and new edges $\{u, w\}$ and $\{w, v\}$, and place them in G'.

   This reduction requires us to copy the initial graph G (i.e. $O(|V| + |E|)$), and for every edge, we need to add a new node (i.e. $O(|E|)$) and create two new edges (i.e. $O(|E|)$). All in all, this takes poly-time: $O(|V| + |E|)$.

2. Show that `VERTEX-COVER`(G, k) has a solution **if and only if** the corresponding `PYGMALION`(G', k') has a solution:

   - Show `VERTEX-COVER`$(G, k) \Rightarrow$ `PYGMALION`(G', k'):
     Let $S \subseteq V$ be a vertex cover in G. We will show that S is a valid solution for G'. First notice that if S is a vertex cover, then this implies that for every edge in G, at least one of its endpoints is in S. Now consider some vertex x in G', i.e. $x \in V'$.

  - If x is an original vertex in G, then either $x \in S$ or there must exist some edge in G which connects x to some vertex $u \in S$. Therefore, in `PYGMALION`, x either has a bunker (i.e. $x \in S$), or x is a connected to some vertex u which has a bunker (i.e. $u \in S$).
  - If x is not in G, then notice that it forms a "triangle" with two adjacent vertices u, v in G. The triangle consists of three edges: (u, x), (x, v), and (u, v). Note that edge (u, v) is in G. Therefore, it must be covered by S (i.e. at least one of u or v is in S). This means that vertex x is connected to by an edge to a vertex belonging to S. In other words, x is connected to some vertex which has a bunker.

  Therefore, if G has a vertex cover S, then S is also a solution to `PYGMALION`(G', k'), where $k' = k = |S|$.

- Show `PYGMALION`$(G', k') \Rightarrow$ `VERTEX-COVER`(G, k):
  Let D be a solution for G' (of size $|D| = k'$). Consider first the vertices $w \in D$ that **do not** correspond to an original vertex in G. Notice that w must be connected to **exactly two** vertices u, v in G. Therefore, we can replace w by either u or v, while still maintaining the size and validity of the solution (since u, v, w form a triangle, and w connects only to u and v). Doing this for all vertices $w \in D$ that don't belong to the graph G, we obtain a solution S (of size $|S| = |D| = k'$) that is still a valid solution over G', but that only consists of vertices in the original graph G. Because every edge (u, v) generates a vertex x that must be covered in G' by either u or v, this guarantees that the edge (u, v) is covered in G by either u or v. Thus, all the additional vertices in G' are "covered" by the solution S, which implies that all edges in G are covered by S. So, if G' has a solution D, this can be converted so a valid vertex cover for G of size at most k.

3. Hence, because `PYGMALION` is in NP and has been shown to be at least as hard as `VERTEX-COVER`, then it is in NP-complete.

Programming

Divide-and-conquer
This algorithm works as follows:

1. Divide the given array of interest rates into two halves
2. Return the total interest rate, start and end days of the interval with the maximum total interest rate among:
   - the first half (Make a recursive call)
   - the second half (Make a recursive call)
   - the intervals that cross the midpoint between the halves. This can be obtained in linear time by starting at the midpoint and extending the interval in both directions while maximizing the sums achieved at both sides of the midpoint.

Time Complexity: The following recurrence characterizes the running time complexity of this algorithm:

$$T(n) = \begin{cases} O(1) & n = 1 \\ 2T(n/2) + O(n) & n > 1 \end{cases}$$

Using the master method, the recurrence above has a solution $T(n) = O(n \log n)$. Since $a = 2, b = 2, d = 1$, case 2: $a = b^d$ applies and the time is $O(n^d \log n)$ (see CLRS Page 74).

Space Complexity: We only require $O(1)$ space for one call of the subroutine, but taking into account all the recursive calls, the overall space complexity is $logn$.

Algorithm 1: DivideAndConquer

```
Input: Array of interest rates $A = a_1, ..., a_n$, low, high
Output: Total interest maxsum, start and end days for optimal interval between low, high
if $low = high$ then
    return $maxsum = \max(0, a_{low})$, $start = end = low$
end
mid = (low + high)/2
[maxsum1, start1, end1] = DivideAndConquer(A, low, mid)
[maxsum2, start2, end2] = DivideAndConquer(A, mid+1, high)
[maxsum3, start3, end3] = CrossingSum(A, low, high, mid)
if $maxsum1 \geq maxsum2$ and $maxsum1 \geq maxsum3$ then
    return maxsum1, start1, end1
end
else if $maxsum2 \geq maxsum1$ and $maxsum2 \geq maxsum3$ then
    return maxsum2, start2, end2
end
else
    return maxsum3, start3, end3
end
```

Algorithm 2: CrossingSum

```
Input: Array of interest rates A, low, high, mid
Output: Total interest maxsum, start and end days for optimal interval between low, high that
        includes mid
maxleft = 0
maxsofar = $a_{mid}$
for $(mid - 1) \geq i \geq low$ do
    maxsofar = maxsofar + $a_i$
    if $maxsofar > maxleft$ then
        maxleft = maxsofar
        start = $i$
    end
end
maxright = 0
maxsofar = $a_{mid+1}$
for $(mid + 2) \leq i \leq high$ do
    maxsofar = maxsofar + $a_i$
    if $maxsofar > maxright$ then
        maxright = maxsofar
        end = $i$
    end
end
return $maxsum = maxleft + maxright$, start, end
```

Dynamic Programming
We are given an array of interest rates $A = a_1, ..., a_n$. Let $B(j)$ denote the maximum sum of interest rates that can be obtained if j is the last day of investment. Then, it is easy to see that $B(j)$ can be computed using the following recurrence relation:

$$B(j) = \begin{cases} 0 & j = 0 \\ \max\{B(j-1) + a_j, 0\} & \text{otherwise} \end{cases}$$

In words, if j is Emily's last day of investment, the maximum interest rate he can obtain up to j is either: the maximum interest rate he can obtain up to $j-1$, plus that of the j-th day (if $B(j-1) + a_j \geq 0$); or zero (if $B(j-1) + a_j < 0$, i.e. there is no investment interval ending on day j that is profitable).
You can easily (and efficiently) compute $B(j), \forall j = 1...n$ using a bottom-up approach. Once you have computed all the necessary $B(j)$ values, we first find the maximum value among all $B(j)$, then we find a j which achieves this maximum (there could be multiple such). Then starting from j, we scan backward in time adding up the interests until we achieve a running sum of exactly $B(j)$, at this point we stop and use that day as our starting point i.

Time Complexity: We have $O(n)$ subproblems and each takes constant time, hence the total running time of computing the $B(j)$'s is $O(n)$. Finding the max and reconstructing the interval also takes linear time.

Space Complexity: We only need $O(n)$ space in order to keep track of $B(i)$, and the maximum total interest rate encountered up to day i.

Algorithm 3: DP

```
Input: Array of interest rates $A = a_1, ..., a_n$
Output: Total interest max, start and end days for optimal interval
maxsofar = 0
$B(0) = 0$
for $1 \leq j \leq n$ do
    $B(j) = \max(a_i, a_i + B(j-1))$
    maxsofar = $\max($maxsofar$, B(j))$
    maxindex = $j$
end
max = 0, end = start = maxindex
while max < maxsofar do
    max = max + $a_{\texttt{start}}$
    start = start − 1
end
return max, start, end
```