CS 317, Assignment 4

Answers

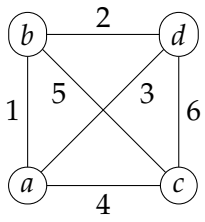
1. Let $G = (V, E)$ be an undirected, connected graph with weight function $w : E \to R$. Suppose that $|E| \geq |V|$ and that all the edge weights are distinct.

   Let $\mathcal{T}$ be the set of all spanning trees of G, and let T be the minimum spanning tree of G. Then a *second-best minimum spanning tree* is a spanning tree T' such that $w(T') = \min\{w(T'') : T'' \in \mathcal{T} \setminus \{T\}\}$.

   (a) Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree is not necessarily unique.

   Answer:

   The following graph that has a minimum spanning tree of cost 7 and two second-best spanning trees (each of cost 8).

   (b) Let T be the minimum spanning tree of G. Prove that G contains some edge $(u, v) \in T$ and some edge $(x, y) \notin T$ such that $T \setminus \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.

   Answer:

   We need to show that there is a single edge swap that can demote our minimum spanning tree to second best. A simple greedy-replace argument will work as always.

   Recall the meta-algorithm for minimum spanning trees discussed in class. When we obtain a second best minimum spanning tree, there must be some edge (x, y) that is added to the tree and is not safe (but not useless either) instead of the safe edge (u, v), for otherwise we would obtain the minimum spanning tree rather than the second best minimum spanning tree. Let now (x, y) be the smallest weight such a replacement for (u, v).

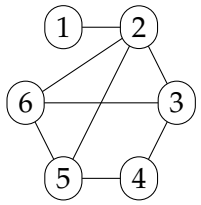
Then $T \setminus \{(u,v)\} \cup \{(x,y)\}$ is a second-best minimum spanning tree of G: Its cost is higher than T (since we replace an edge with one of higher weight), and also smaller than the cost of any other spanning tree (since the weight of (x,y) is smaller than the weight of any other non-useless edge that could replace it).

---

2. Let $G = (V, E)$ be an undirected graph. A vertex cover of G is a subset U of V such that every edge in E is incident to at least one vertex in U. A minimum vertex cover is one with the lowest number of vertices. For example $\{1,3,5,6\}$ and $\{2,3,5\}$ are both vertex covers for the graph below, with the latter being a minimum vertex cover.

   Consider the following greedy algorithm for the minimum vertex cover problem:

   algorithm VertexCover($G = (V, E)$)**:**
   - $U \leftarrow \emptyset$
   - **while** $E \neq \emptyset$ **do**
     - Find a vertex v of maximum degree
     - $U \leftarrow U \cup \{v\}$
     - $V \leftarrow V \setminus \{v\}$
     - $E \leftarrow E \setminus \{(x,y) \in E : x = v \vee y = v\}$
   - **return** U

   (a) Refine the algorithm above by providing detailed descriptions of all the steps using suitable data structures. Justify your choice of data structure and determine the overall running time.

   Answer:

   Let as usual $n = |V|$ and $m = |E|$. Any respectable greedy algorithm is going to use a priority queue. Since we always choose the vertex with maximum degree, we will use this queue to store the remaining vertices using their degree as key. This is what V is going to become. We also notice that each time we remove an edge one vertex in the queue will change its key, which pretty much rules out anything but a heap as implementation for our queue.

   The output set U can be any data structure that allows adding elements to it in constant time. We could use an array but the easiest way out (for me) is to make it a singly linked list with the operator Cons(x, l) adding x in front of l (in obvious constant time).

   I cannot think of any realistic data structure that permits filtering out values from a set in less than linear time. If we accept linear time (because we must) for this operation then we can thing of E again as a singly linked list. Every time we delete an edge we also have to update the degree of the end point that is not the vertex being considered. Here is the refined algorithm:

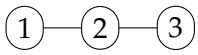
```
algorithm VertexCover(G = (V, E)):
    U ← ⟨⟩
    Q ← MakeQueue(V)
    while E ≠ ⟨⟩ do
        v ← DeQueue(Q)
        U ← Cons(v, U)
        foreach (x, y) ∈ E do
            if (x, y) = (v, u) ∨ (x, y) = (u, v)
            then
                E ← E \ {(x, y)}
                DecreaseKey(u, Q)
    return U
```

For the running time we note that in the worst case the foreach loop executes m times. This happens when the update always remove a single edge, which happens e.g. when the graph is a linear structure with each vertex connected to a "next" vertex. That is, the worst case running time is $O(t_m(n) + m(t_-(n) + mt_x(n))$, with t_m the time complexity of making the queue,$t_-(n)$ the complexity of DeQueue, and t_x the complexity of DecreaseKey. With the queue implemented as a heap we end up with a $O(n+m^2 \log n)$.

(b) Give an example (other than the graph above) where this algorithm returns a minimum vertex cover.

Answer:

The simplest example I can think of goes like this:

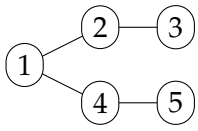


The maximum degree vertex is 2, which is the vertex cover and is also picked out first (and as it turns out also last) by the algorithm.

(c) Give an example (other than the graph above) where this algorithm does not return a minimum vertex cover.

Answer:

Here is the simplest example I can think of:



If the algorithm is lucky enough to never choose vertex 1 then it will reach the optimal vertex cover $\{2, 4\}$. However, the moment the algorithm is unlucky and chooses vertex 1 (which is available as the maximum degree vertex together with 2 and 4) things go

sideways. When this happens the vertex cover returned by the algorithm is either $\{1, 2, 5\}$ or $\{1, 4, 3\}$, which are both sub-optimal.

Another, more complex example is shown in my answer to Question 2d below.

(d) How far from the minimum vertex cover do you think the vertex cover produced by this algorithm is in the worst case? You do not have to provide a proof or tight argument, but you need to offer some justification.

Answer:

It may seem counterintuitive and is not easy to see, but the ratio between the sizes of the minimum vertex cover and the vertex cover produced by this algorithm is actually arbitrarily large (meaning that it is not bound by any constant).

To see this, consider the graph $G = (L \uplus R, E)$. The vertex set L consists of r vertices. The vertex set R is further sub-divided into r sets $R_1, \ldots, R_r$. Each vertex in R_i has an edge to i vertices in L and no two vertices in R_i have a common neighbour in L; thus, $|R_i| = r/i$. It follows that each vertex in L has degree at most r and each vertex in R_i has degree i. The total number of vertices is $n = \Theta(r \log r)$. Note that the graph is bipartite, and so both L and R are vertex covers. It is easy to see that the optimal vertex cover is L (of size r).

While the algorithm can choose vertices from L at any stage thus producing the optimal vertex cover, that would be just a matter of luck since at every step there are two maximum degree vertices, one in L and another in R. It is therefore quite possible that the vertex cover produced by the algorithm will be R, which means that the ratio between the sizes of the produced solution to the optimal solution is $O(\log r)$.

---

3. You are an advertising company that wants to advertise something to all n people in the city. You know that each of these n people will come downtown on Sunday for some interval of time. You have acquired these time intervals for all people through some unethical means. You do not afford to put ads downtown, but you can pay people to carry your ad on their t-shirts. Assume that if X is carrying the ad, then anyone whose time interval intersects with the time interval of X will see the ad (of course, X will also see the ad). You want to choose the minimum number of people you should pay so that everyone sees the ad. Design an algorithm for this problem. Establish the correctness of the algorithm and analyze its running time.

Answer:

The only information that matters for every person i, $1 \leq i \leq n$ is the time interval $[s_i, e_i]$ when that person is downtown. We thus need to choose the minimum amount of time intervals that overlaps all the time intervals. The problem becomes: Given a set of intervals $I = \{I_i = [s_i, e_i] : 1 \leq i \leq n, s_i, e_i \in \mathbb{R}\}$, choose a minimal subset $S \subseteq I$ such that for all $[s_i, e_i]$ there exists an overlapping interval $[s, e] \in S$.

However tempting it is to produce a greedy algorithm for this problem, we cannot. Indeed, the problem does *not* have the greedy choice property.

To see this, consider the following instance: $I = \{B_1, B_2, B_3\}$ with $B_1 = [0, 4]$, $B_2 = [1, 5]$, and $B_3 = [2, 6]$, with the optimal solution $S = \{B_2\}$ (which overlaps all the intervals). We must choose an interval according to some greedy choice rule g. We do not know what g is, except that it must choose B_2. Note incidentally that g cannot choose the earliest-starting interval (which is B_1) or the latest-ending interval (B_3), and in general cannot be based on solely the start or the end of the intervals.

Consider now the slightly larger instance $I' = \{A, B_1, B_2, B_3, C\}$, where the additional intervals are $A = [0, 3]$ and $C = [5.5, 7]$. The optimal solution for this instance is $S' = \{B_3\}$. The greedy choice rule g cannot choose either A or C, so it must choose one of B_1, B_2, or B_3 and thus solve the instance I. The choice g is greedy (that is, solely based on the current instance) and so g will choose B_2 (as seen above). This will result in the overall solution $\{B_2, C\}$, which is not optimal. Thus no greedy choice rule can return the optimal solution for all instances.

Since we cannot use a greedy algorithm for the problem we will proceed with the next best thing namely, dynamic programming. Such a solution is pretty straightforward, as follows: First we sort the instance based on the end time of the intervals, thus obtaining $I = \langle I_1, I_2, \ldots, I_n \rangle$. For each I_i define $l_i = \min\{j < i : I_j \cap I_i \neq \emptyset\}$ (the earliest interval that overlaps I_i) and $r_i = \max\{j < i : I_j \cap I_i = \emptyset\}$ (the last interval that does not overlap I_i).

Let now D_i be the minimum size of the set that covers all the intervals I_1 to I_i. We have:

$$D_i = \begin{cases} 0 & \text{if } i = 0 \text{ (no intervals to cover)} \\ \min\{1 + D_{r(i)}, D_{l(i)}\} & \text{otherwise} \end{cases}$$

Indeed, we can either pick I_i and then the rest of the intervals to cover are from 1 to $r(i)$ (covered with cost $D_{r(i)}$), or we do not pick I_i, case in which all the intervals must be covered by the earlier intervals at cost $D_{l(i)}$.

The values D_i can be clearly computed left to right, resulting in the obvious dynamic programming solution. The answer to the overall problem is a cover for all the intervals that is, D_n. To find out the actual intervals in the solution we can mark I_i as included iff the minimum is realized by $1 + D_{r(i)}$.

The implementation is straightforward and so I am not going to include it here.

Regarding the running time, the initial sorting takes $\Theta(n \log n)$ time. Computing l_i and r_i can be accomplished by an obvious sweep of I maintaining the earliest interval that still overlap I_i and also the last interval whose end is less than the start of I_i. The dynamic programming algorithm runs in linear time. Overall the sorting dominates the rest of the computation for an overall running time of $\Theta(n \log n)$.

And yes, this should teach me to not give problems before I try them out myself.

---

4. Consider the problem of making change for n cents using the smallest number of coins. Assume that the value of each coin is an integer.

(a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Analyze the running time and prove that your algorithm yields an optimal solution.

Answer:

Consider the following greedy solution:

i. Give $q = \lfloor n/25 \rfloor$ quarters, which leaves $n_q = n \bmod 25$ cents to make change.
ii. Then give $d = \lfloor n_q = 10 \rfloor$ dimes, which leaves $n_d = n_q \bmod 10$ cents to make change.
iii. Then give $k = \lfloor n_d/5 \rfloor$ nickels, which leaves $n_k = n_d \bmod 5$ cents to make change.
iv. Give $p = n_k$ pennies.

Assuming a reasonably small original amount (so that arithmetic operations can be done in constant time) there is one step for each coin denomination and so the algorithm clearly runs in $O(d)$ time where d is the number of coin denomination (that is, $d = 4$).

To prove correctness we need to show that the problem is correctly solved by a greedy algorithm. We therefore show that the change for some amount n includes at least one coin of value c where c is the maximum coin denomination such that $c \leq n$. This shows that the greedy choice (of choosing the largest denomination available first) is the right choice.

If $n < 5$ then this property clearly holds ($c = 1$ and so only pennies are usable). For $5 \leq n < 10$ (so that $c = 5$) suppose we have a solution that does not use nickels (and so only use pennies); we then take off 5 pennies and replace them with a nickel, thus obtaining a better solution. For $10 \leq n < 25$ (that is, $c = 10$), we replace some subset of nickels and pennies that add up to 10 with a nickel for a better solution, and so on.

(b) Suppose that the available coins are in denominations that are powers of c that is, the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

Answer:

The proof is a generalization of the corresponding proof from the answer to the previous question. Let the amount be n and i be the largest integer such that $c^i \leq n$. Assume that the optimal change does not include any coin of value c_i. Then, that change must make up for the c^i component of the change using more coins, and so it is not optimal (since that component can be replaced by a single coin of value c^i for a better solution).

We still need to show that we must have a distinct component of value c^i in the optimal change. Let an optimal solution to the problem be $n = \sum_{k=0}^{i} a_k c^k$. If this is the case then it must be the case that $a_k < c$ for all $0 \leq k \leq i$. Indeed, whenever $a_k \geq c$ we can make better change by replacing c coins of value c^k with a single coin of denomination c^{k+1}. We then have $\sum_{k=0}^{i-1} a_k c^k < \sum_{k=0}^{i-1} c c^k = c^i$ and so a_i cannot be zero.

(c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

Answer:

Suppose that back in 2013 the Royal Canadian Mint decided to stop minting the nickel instead of the penny, so that we can use only 1, 10, and 25 valued coins to make change. The greedy change for 30 cents would consist of 6 coins (one quarter and five pennies), whereas the optimal change consists of 3 dimes.

In this situation the above argument of greedy optimality breaks down because we no longer necessarily have a component in the optimal change that add up to exactly the largest denomination. In the change for 30 cents for example no component add up to 25.

(d) Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations using the smallest number of coins, assuming that one of the coins is a penny.

Hint. Such an algorithm is obviously not going to be a greedy algorithm, so you may want to look at the next algorithm design technique.

Answer:

Let n be the amount to make change for, the coin denominations be $d_1, d_2, \ldots, d_k$, and c_j be the minimum number of coins needed to make change for j cents. Obviously $c_j = 0$ for $j = 0$ (and as it turns out to be convenient for $j \leq 0$). Now consider that c_j contains a coin of denomination d_i. If this is the case, then $c_j = 1 + c_{j-d_i}$. Since we are looking to minimize c_j we will need to choose d_i so that c_{j-d_i} is minimized. In all we reach the following recursive definition for c_j:

$$c_j = \begin{cases} 0 & \text{if } j \leq 0 \\ c_j = 1 + \min_{1 \leq i \leq k} c_{j-d_i} & \text{if } j > 1 \end{cases}$$

This can be trivially converted into an instance of dynamic programming using an array for memoizing c_j and filling in the array in increasing order of indices up to index n. The desired running time follows immediately.

Make sure you review the submission guidelines posted on the course's Web site before submitting.