CMSC 451:Fall 2017 Dave Mount

Solutions to the Midterm Practice Problems

Solution 1:

(a) $\Theta(n)$: The innermost loop is executed i times, and each time the value of i is halved. So the overall running time is

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \;=\; n\left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) \;\leq\; 2n \;=\; \Theta(n).$$

(b) (i) True: There is a graph that has a cut vertex but no cut edges. Consider a "bowtie-shaped" graph consisting of two triangles that share a common vertex.

(ii) True: There is a graph that has a cut edge but no cut vertices. Consider a graph consisting of a single edge. (By the way, this is essentially the only way to do this. If the graph is connected and has three or more vertices, then one of the two vertices incident to a cut edge is a cut vertex.)

(c) $kn/2$: The sum of degrees of the vertices of such a degree-k graph is at most kn. The sum of degrees in a graph is twice the number of edges (since each edge is counted twice, one for each endpoint). Therefore, the number of edges is at most $kn/2$ (or more precisely $\lfloor kn/2 \rfloor$ since it is an integer quantity).

(d) (i) Yes, this is a prefix code. No code word is the prefix of any other.

(ii) No, this would never be generated by Huffman's algorithm. There would be no reason for generating the codeword 101, since the shorter code 10 would suffice.

(e) (iii): Dijkstra's algorithm (as given in class) always runs in the same time, but when negative cost edges are given, it may incorrectly label a vertex's distance.

(f) (i): Bellman-Ford works correctly even if there are negative edge costs, assuming that there are no negative-cost cycles.

(g) (i): Floyd-Warshall works correctly even if there are negative edge costs, assuming that there are no negative-cost cycles.

Solution 2:

(a) EAF can be arbitrarily bad. Consider an instance with $n-1$ non-overlapping intervals and one interval that covers all of them. This covering interval starts first, and so it will be selected. The optimum algorithm would select the other $n-1$ intervals, so the performance ratio is $(n-1)/1$, which is unbounded as n tends to infinity.

(b) SAF is not optimal. Consider an instance consisting of three activities consisting of two non-overlapping intervals and a shorter interval that overlaps both. It is possible to schedule the two non-overlapping intervals, but greedy will select the shorter overlapping interval.

We can prove that SAF has a performance ratio of at most 2. Here is a sketch. Each time greedy selects an interval there are two possibilities: (1) Opt also selects this interval, or (2) Opt does not choose this interval. In case (2), Opt can select at most two intervals that overlap the interval chosen by greedy. This is because if there were three such intervals in Opt, the middle one would need to be even shorter than the greedy choice, contradicting the fact that greedy picks the smallest interval. Since each greedy choice is balanced against at most two optimum choices, it follows that $|G| \geq |O|/2$.

Solution 3: Define $P[u]$ to be the number of maximal paths that start at u. We apply DFS to G and compute $P[u]$ as we visit each vertex u. If u has no outgoing edges then $P[u] = 1$, which counts the trivial path $\langle u \rangle$. Otherwise, observe that we can form every maximal path from u by taking a tail maximal path from each of its neighbors v and prepending u to this path. Therefore, the total number of maximal paths is the sum of $P[v]$ for all neighbors v of u. For each neighbor v of u, we invoke DFSvisit on v if necessary, and then we add $P[v]$ to $P[u]$. Clearly, the running time is $O(n + m)$.

Solution to Problem 5

```
DFS(G=(V,E)) {
    for each (u in V) { mark[u] = undiscovered }
    for each (u in V) {
        if (mark[u] == undiscovered) { DFSvisit(u) }
    }
    return P
}

DFSvisit(u) {
     mark[u] = discovered
     if (Adj[u] == empty) P[u] = 1  // outdegree = 0 - count trivial path
     else {                         // u is a non-terminal
        P[u] = 0
        for each (v in Adj[u]) {
            if (mark[v] == undiscovered) { DFSvisit(v) }
            P[u] += P[v]            // count paths u,v, ...
        }
     }
 }
```

Solution 4: We assume no gap exceeds 100 miles. We greedily go as far as possible before stopping to refuel. The variable `lastStop` indicates the location that he last got fuel. We find the farthest station from this that is within 100 miles and get fuel there. The code block below provides a sketch of the algorithm. To avoid subscripting out of bounds, let us assume that $x[n+1] = x[n]$.

Clearly the running time is $O(n)$. Observe that this produces a feasible sequence, since we never go more than 100 miles before stopping. To establish optimality, let $F = \langle f_1, f_2, \ldots, f_k \rangle$ be the indices of an optimal sequence of refueling stops, and let $G = \langle g_1, g_2, \ldots, g_{k'} \rangle$ be the greedy sequence. If the two sequences are the same, then we are done. If not, let i be the smallest index such that $g_i \neq f_i$. Because greedy algorithm selects the *last* possible gas station, we know that $g_i > f_i$. Consider an alternative solution F' which comes by replacing f_i with g_i. We claim that F'

Greedy Algorithm for Refueling

```
fuel(x, n) {
    lastStop = 0
    for (i = 1 to n) {
        if (x[i+1] > lastStop + 100) {  // will run out of gas before next station?
            refuel at x[i]              // refuel here
            lastStop = x[i]             // this was our last gas
        }
    }
}
```

is a also a feasible solution. To see this observe that sequence up to g_i is the same as G (which we know is feasible) and because we have delayed fueling, for the rest of the trip we have at least as much gas as we had with F (which we know is feasible). The sequence F' has the same number of stops as F and so is also optimal, and it has one more segment in common with G. By repeating this, eventually we will have an optimal solution that is identical to G.

Solution 5:

(a) Let v_i be the cost of bottle i, and let b_i denote the number of pills it holds. In order to place W pills as inexpensively as possible, sort the bottles in increasing order of cost per pill, that is, v_i/b_i. Then fill the bottles in this order, until all the pills are gone.

To show that this is optimal, define the *incremental cost* for a pill to be v_i/b_i, where i denotes the bottle into which this pill was placed. Because we only pay for the portion of the bottle that we use, the total cost of bottling all the pills is equal to the sum of the incremental costs over all W pills. (This is important. For example, we can put a single pill into a last bottle, without paying for the entire bottle.)

For a given input, let G denote the sorted order of incremental costs for the greedy solution, and let O denote the sorted order of incremental costs for any optimal solution. We will use the usual exchange argument to show that G achieves the same cost as O.

If $G = O$, we are done. Otherwise, consider the first pill (in sorted order of incremental cost) where O differs from G. Let i denote the bottle into which G puts this pill, and let i' denote the bottle used by O. Since both sequences have been sorted, we know that O does not put any more pills into bottle i, even though there is still space remaining there (since G put this pill there). Since G places pills in increasing order of incremental cost, it must be that $v_i/b_i \leq v_{i'}/b_{i'}$. Let us create a new bottling plan by moving this one pill from bottle i' to bottle i. The incremental change in cost by doing this is $v_i/b_i - v_{i'}/b_{i'} \leq 0$. Therefore, the total cost cannot increase as a result of the change. (Since O is optimal, it should not decrease.) By repeating this process, we will eventually convert O into G, while never increasing the bottling cost. Therefore, G is optimal.

(b) For $0 \leq i \leq n$, define $P(i, w)$ to be the minimum amount paid assuming that we place w pills using some subset of the first i bottles. For the basis case, observe that if $w = 0$ and $i = 0$, we can trivially put the 0 pills in 0 bottles for a cost of 0, and thus $P(0, 0) = 0$. If $W > 0$, then there is no solution using 0 bottles, and so we have $P(0, W) = \infty$.

For the induction, let us assume that $i \geq 1$. There are two cases. Either we do not use the ith bottle or we do. If not, we put all w pills in the first $i-1$ bottles, for a cost of $P(i-1, w)$. Otherwise, we put $\min(b_i, w)$ pills in this bottle, and put the remaining pills in the previous $i-1$ bottles. The total cost is $v_i + P(i-1, w - \min(b_i, w))$. We prefer the lower of the two choices, which implies the following recursive rule:

$$\begin{aligned} P(0,w) &= \begin{cases} 0 & \text{if } w = 0 \\ \infty & \text{otherwise} \end{cases} \\ P(i,w) &= \min(P(i-1,w), v_i + P(i-1, w - \min(b_i, w)) \qquad \text{for } i \geq 1. \end{aligned}$$

Solution 6:

(a) Suppose in general that there are $n+1$ coin denominations of the form $2^0, 2^1, \ldots, 2^n$. Here is a greedy algorithm for making change from these denominations. Let A denote the amount to make change for. For $0 \leq i \leq n$, let g_i denote the number of coins of value 2^i. The algorithm generates the maximum number of coins for the highest denomination and then decreases A by the amount generated. Let `2**i` denote 2^i. Here is the algorithm.

```
for (i = n downto 0) {
    g[i] = floor(A / 2**i)  // maximum number of coins of value 2**i
    A -= 2**i * g[i]        // decrease A by the amount generated
}
return g[0..n]
```

We will show by induction on n that this greedy algorithm is optimal. For the basis case $n = 0$, we only have coins worth $2^0 = 1$ cent, and the algorithm generates $g_0 = \lfloor A/2^0 \rfloor = A$ 1-cent coins, which is clearly optimal.

Otherwise, let us assume the induction hypothesis that the algorithm is optimal for coins of values $2^0, \ldots, 2^{n-1}$. We will show that it is optimal when the largest coin value is 2^n. Let g_n and o_n be the number of coins of value 2^n generated by the greedy and optimal algorithms, respectively. The algorithm generates $g_n = \lfloor A/2^n \rfloor$ coins of this value. It is not possible to generate more such coins without exceeding A, and thus $o_n \leq g_n$. We assert that $o_n = g_n$. To see why, suppose to the contrary that $o_n = g_n - c$ for $c \geq 1$. The greedy algorithm leaves a remainder of $A' = A - g_n 2^n$, and the optimum leaves a remainder of $A'' = A' + c2^n$. By induction, the optimum way to make change for A'' is to generate $\lfloor A''/2^{n-1} \rfloor = \lfloor A'/2^{n-1} + c2^n/2^{n-1} \rfloor \geq 2c$ of value 2^{n-1}. We can replace these extra $2c$ coins with c coins of value 2^n, which results in the same remainder but uses c fewer coins. This contradicts the hypothesis that $o_n < g_n$. Therefore, $o_n = g_n$. By the induction hypothesis, greedy is optimal for the remaining coins.

Challenge solution: Here is the solution for the US coin system. Let A be the input amount in cents. The greedy algorithm repeatedly extracts the maximum number of quarters ($\lfloor A/25 \rfloor$) and computes the remainder ($A \bmod 25$). It then repeats the process on the smaller denominations.

```
Q = A / 25   A = A mod 25
D = A / 10   A = A mod 10
```

```
N = A / 5    A = A mod 5
P = A
return (Q,D,N,P)
```

Let (Q, D, N, P) be the output of this program. It is easy to see that this yields the correct amount, but the question is whether the number of coins is as small as possible. To prove this we use induction on the number of different coins. For the basis case, when there is only one coin (the penny) it is trivial, since there is only one way to make change.

It turns out that the hardest case in the induction is the case of quarters. And we will skip the nickel and dime cases, and focus solely on the case of quarters.

Suppose we are given an initial amount A. The greedy algorithm extracts the maximum number $Q = \lfloor A/25 \rfloor$ quarters and leaves a remainder of $A' = A \bmod 25$ cents to be accounted for from the smaller coins. Any alternative algorithm could not take more quarters, so suppose that it took $q < Q$ quarters instead. Then it would use $q' = Q - q$ fewer coins at this stage, but it would have to make change for the larger remaining amount of $A' + 25q'$.

At this point, we may apply the induction hypothesis, which states that greedy is optimal for all smaller denominations. In the first case, the greedy algorithm would extract $\lfloor A'/10 \rfloor$ dimes and continue with $(A' \bmod 10)$. In the case of the alternative algorithm, the number of dimes that the greedy algorithm would extract is

$$\left\lfloor \frac{A' + 25q'}{10} \right\rfloor = \left\lfloor \frac{A'}{10} + q' \frac{25}{10} \right\rfloor \geq \left\lfloor \frac{A'}{10} + 2q' \right\rfloor = \left\lfloor \frac{A'}{10} \right\rfloor + 2q'.$$

Thus the alternative algorithm uses at least $2q'$ more dimes. Then it would continue with $(A' + 25q') \bmod 10 \geq (A' \bmod 10) - 5$. The -5 means that it might save a nickel over greedy (although a more careful analysis shows that it really doesn't). So this alternative algorithm saves q' quarters and possibly one nickel, but uses at least $2q'$ additional dimes. Since $2q' \geq q' + 1$ for all $q' \geq 1$, the savings is never greater than the loss, so the alternative algorithm can be no better.

(b) (This is not the smallest counterexample, but it really happened in history.) The old British system had coins for 1 penny (1d), 3-pence (3d), 6-pence (6d), 10-pence (10d) and 1 shilling which equals 12 pence (1s = 12d). To make 20d the greedy algorithm would use four coins (a shilling, a 6-pence, and two pennies) whereas the optimum algorithm would use two coins (two 10-pence).

Solution 7: We present a dynamic programming solution. An obvious first attempt at a DP solution is to define an array $P[i]$, which is the maximum profit attainable for weeks 1 through i. The problem is that in order to update this array, we need to know where we were during the previous week since we need to know whether we need to charge the \$1000 plane fare.

We will encode this extra conditional information by adding an additional parameter to control for the location where the businessman spent the last week. Let $DC = 0$ and $LA = 1$. For $0 \leq i \leq n$:

$$\begin{aligned} P[i, DC] &= \text{the max profit for weeks 1 through } i\text{, assuming week } i \text{ is spent in DC} \\ P[i, LA] &= \text{the max profit for weeks 1 through } i\text{, assuming week } i \text{ is spent in LA.} \end{aligned}$$

Let's see how to compute each of these arrays. For the basis case, because we start in DC, we pay nothing in travel costs and so we have $P[0, DC] = 0$. On the other hand, if we want to start in LA, we need to pay to get there, and thus, $P[0, LA] = -1000$. (The problem states that you must start in DC, and a valid way to interpret this constraint is to forbid starting in LA, which could be done by setting $P[0, LA] = -\infty$.)

In general, for $i > 0$, to compute $P[i, DC]$, we consider two possibilities, depending on where we spent our last week. If we spent it in DC, we don't need to travel, and we obtain a profit of $\text{DC}[i]$ on top of whatever profit we accrued up to week $i-1$. Thus, we have $P[i, DC] = \text{DC}[i] + P[i-1, DC]$. On the other hand, if we were in LA last week, we need to pay the transportation costs, but we still obtain the weekly DC profit and the accrued profit from the first $i-1$ weeks. In this case we have $P[i, DC] = \text{DC}[i] + P[i-1, LA] - 1000$. To maximize our profit we take the better of the two options. This yields the following recursive rule for P:

$$P[i, DC] \;=\; \text{DC}[i] + \max(P[i-1, DC], P[i-1, LA] - 1000).$$

Symmetrically, to compute $P[i, LA]$, we have the rule for P:

$$P[i, LA] \;=\; \text{LA}[i] + \max(P[i-1, LA], P[i-1, DC] - 1000).$$

Once we succeed in computing the values $P[i, DC]$ and $P[i, LA]$, for $0 \le i \le n$, we return the value $P[n, DC]$ as the final result (because we want to end in DC.) An example is shown below. The final result is $P[5, DC] = \$2000$.

Week	0	1	2	3	4	5
DC		400	100	200	50	1100
LA		210	900	100	1500	20
$P[i, DC]$	0	400	500	700	750	2000
$P[i, LA]$	-1000	-790	300	400	1900	1920

Solution 8: This problem is related to the ancient stone-drawing game called *Nim*.

(a) For $0 \le a \le A$, $0 \le b \le B$, and $0 \le c \le C$, define $W[a, b, c]$ to be true if the first player can force a win with the three piles containing these many stones and false otherwise. Imagine that we are the current player. We win if and only if we can force our opponent into a losing position. To compute $W[a, b, c]$, we assume that all smaller configurations have already been computed. We check all configurations that are reachable from (a, b, c). If we find a reachable configuration (a', b', c') such that $W[a', b', c'] = \text{false}$, then we know that if we make this move we will force our opponent into a configuration where we can force them to lose, thus we win. In this case, we set $W[a, b, c] = \text{true}$. In contrast, if *all* reachable configurations are winners for our opponent ($W[a', b', c'] = \text{true}$), then no matter what move we make, our opponent (if he/she plays perfectly) can force us to lose, and so we set $W[a, b, c] = \text{false}$.

For example, an application of Rule (1) maps the configuration (a, b, c) to $(a-1, b, c)$. If $W[a-1, b, c]$ is false, then this means that we force our opponent to lose, and hence we win, so we set $W[a, b, c] = \text{true}$. Observe that an illegal move results in a negative subscript. To avoid the need to check for negative subscripts, let us assume that if any subscript is out of bounds $W(a,b,c) = \textsf{true}$. (Intuitively, this means that if we attempt to make an illegal move, then our opponent automatically wins.)

The DP formulation is given below. (The notation "$\neg$" denotes boolean negation.)

$$W(a,b,c) \;=\; \begin{cases} \textsf{true} & \text{if } ((\min(a,b,c) < 0) \text{ or} & \text{(neg index)} \\ & \quad \neg W(a-1,b,c) \text{ or} & \text{(Rule 1)} \\ & \quad (\neg W(a,b-1,c) \text{ or } \neg W(a,b-2,c)) \text{ or} & \text{(Rule 2)} \\ & \quad (\neg W(a,b,c-2) \text{ or } \neg W(a,b,c-3)) & \text{(Rule 3)} \\ \textsf{false} & \text{otherwise.} \end{cases}$$

To determine the winner we check $W[A,B,C]$. If it is true, then the first player, Jen, has a winning strategy, and otherwise Ben has a winning strategy.

Observe that the principal of optimality holds here, since each player moves independently from the other. Note that we do not need to make special cases for the losing configurations $(0,0,0)$ and $(0,0,1)$ (but it wouldn't hurt to do so). The reason is that all moves starting at these configurations lead to illegal configurations, which are marked as true. If all reachable configurations are true then this entry is marked false.

(b) The memoized version is straightforward. I will present a bottom-up solution that runs in $O(ABC)$ time. If n denotes the total number of stones, the running time is $O(n^3)$.

```
findWinner(int A, int B, int C) {
      // Assume that W[a, b, c] = true if any of a, b, or c is negative
      for (a = 0 to A) {
        for (b = 0 to B) {
            for (c = 0 to C) {
                if (!W[a-1, b, c] || !W[a, b-1, c] || !W[a, b-2, c] ||
                    !W[a, b, c-2] || !W[a, b, c-3]) W[a, b, c] = true
                  else W[a, b, c] = false
              }
          }
      }
      if (W[A, B, C]) print("First player wins")
      else print("Second player wins")
  }
```

Solution 9:

(a) A counterexample is shown in Fig. 1. The left side shows that there exists a dominating set of size two (the black vertices). The greedy algorithm will select one of the central four vertices, since each dominates five vertices. Suppose that it first selects vertex a, covering itself and all the shaded vertices. After this, it is doomed to picking at least two other vertices. In this case it will select b and then c. Thus, the greedy dominating set is of size three.

(b) The dominating-set problem can be reduced to the set-cover problem. Recall that in set cover we are given a universe $X = \{x_1, \ldots, x_m\}$ and a collection of sets $S = \{s_1, \ldots, s_n\}$ and are asked to find a minimum-sized collection of S whose union covers all the elements of X.

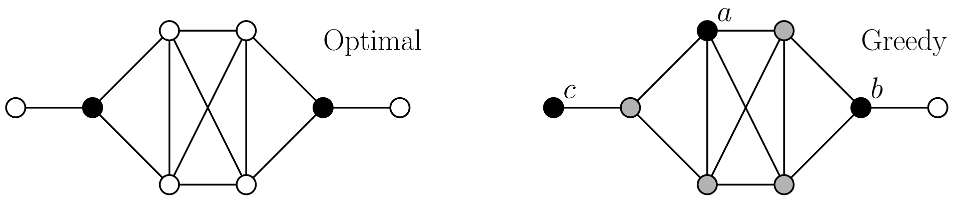

Figure 1: Counterexample for the greedy dominating set algorithm.

Given the graph $G = (V, E)$, we set $X = V$, and for each vertex u, we define the set s_u to consist of u together with all of u's neighbors in G. To see why this is correct, observe that if G has a dominating set V' of size k, then every vertex of V is either in or is adjacent to a vertex of V', which implies that the sets associated with the vertices of this dominating set define a set cover for S of size k. Conversely, if S has a set cover of size k, then the associated vertices of G must have the property that ever vertex of G is either in this set or is adjacent to a vertex of this set. Therefore, the associated vertices defines a dominating set of size k within G'.

Finally, because the greedy set cover algorithm is guaranteed to produce a collection whose size is larger than the optimum by a factor if at most $\ln |X| = \ln |V| = \ln n$.