CPSC 320 2017W2: Quiz 2

January 26, 2018

1 Wall-E Loman

If you find a path with no obstacles, it probably doesn't lead anywhere. — Frank A. Clark

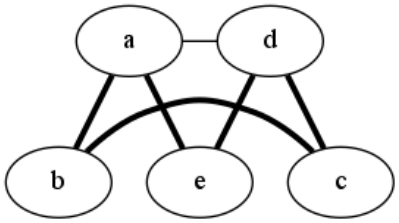
We define TSP just as before: The *Traveling Salesperson Optimization Problem* (TSP) can defined as follows: you are given a set $\{C_1, C_2, \ldots, C_n\}$ of cities. For every two cities C_i, C_j, there is a cost $c_{i,j}$ for traveling from city C_i to city C_j (equal to the cost of travelling from C_j to C_i). Your job is to find the lowest cost path that starts at C_1, travels through every other city, and returns to C_1. You are not allowed to go through a city more than once (except for C_1 which appears both at the start and at the end of your trip).

1. Write in the blank the number of valid solutions to an instance of TSP with n cities: ☐

2. Imagine that you have a brute force algorithm that builds up a partial path step-by-step as a candidate solution to TSP. Each time it has visited all other nodes, it returns to C_1 and measures the cost of the solution. Which of the following is a legitimate heuristic you could add to this algorithm to avoid investigating every valid solution? Fill in the box next to **all** correct answers.
   - ☐ pick a single arbitrary city besides C_1 to be the **only** second city the algorithm considers
   - ☐ always add as the next city in the partial path the cheapest not-yet-visited city to reach
   - ☐ never use the connection between C_1 and its highest-cost neighbor
   - ☐ discard any partial path that is more expensive than the cheapest complete path found so far
   - ☐ discard any partial path of length three like C_1, C_i, C_j if $j > i$

Turn the page for the next problem!

3. **We define HAMCYCLE just as before:** The *Hamiltonian Cycle Problem* (HAMCYCLE) is defined as follows: Given an undirected graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, determine if there is a simple cycle in the graph that visits all nodes, that is, a permutation $v_{i,1}, v_{i,2}, \ldots, v_{i,n}$ of V such that $(v_{i,n}, v_{i,1}) \in E$ and for $j = 1, \ldots, n-1$, $(v_{i,j}, v_{i,j+1}) \in E$. For instance, in the following graph, the bolded edges form a Hamiltonian Cycle, but no path that includes the edge `a--d` is a Hamiltonian Cycle:



Notice that the solution to HAMCYCLE as we defined it is either "Yes" (meaning there is such a path) or "No". Fill in the blanks to make a correct reduction from HAMCYCLE to TSP:

Given an instance of HAMCYCLE, $G = (V, E)$, construct an instance of TSP as follows:

For each vertex v_i, produce [ ].

For each pair of vertices $\{v_i, v_j\}$: If there is an edge $\{v_i, v_j\} \in E$, then let [ ] $= 1$.

Otherwise, let [ ] $= 2$.

Then, given a solution to TSP with total cost k, produce as a solution to HAMCYCLE:

[ ]

2 The Antepenultimate Jedi

It's 30 years in the future. You're working on the prequel to the prequel to The Last Jedi. You want to get the best stars to draw the most people. For each star, you have a positive number indicating their "draw".

You can hire as many stars as you like. Unfortunately, **every actor** considers themselves a "big fish", and big fish won't work with someone between their own draw and half as much (because it steals too much of their own spotlight). Specifically, if you include a big fish with draw x, you may not include any other star with draw k such that $\frac{x}{2} < k \leq x$.

1. Imagine you include a star with draw 11. Can you also hire a star with draw 13? Choose the **best** answer.
   - ◯ Yes
   - ◯ No
   - ◯ It depends on the remainder of the instance/solution.

2. Write in the box below the total draw of the optimal set of stars to hire if the available stars have draw: $[3, 8, 20, 2, 14, 6]$. (Write only the total, not the individual draws.)

   Total draw: ☐

3. Which of these **best** describes the promise of a greedy approach to this problem?

   An optimal greedy approach runs in:
   - ◯ $O(n^2)$
   - ◯ $O(n \lg n)$
   - ◯ $O(n)$
   - ◯ $O(\lg n)$
   - ◯ $O(1)$
   - ◯ none of these because there is no optimal greedy approach

We now alter the problem. In the new version, many actors are not big fish. They'll work with other actors as long as they don't have the exact **same** draw. The big fish will still not work with any other actor with draw k such that $\frac{x}{2} < k \leq x$.

The actor with the highest draw overall is automatically a big fish, whether or not they're in your movie.

Additionally, any *other* actor with draw x is a big fish if there is no actor at all (whether or not they're in your movie) with draw y such that $x < y < 2x$ (**corrected** from the quiz version).[1]

So, for example, if the available stars have draw $[8, 10, 22, 3, 2, 30, 4]$ then 30 (as the highest draw actor) is a "big fish". 22 is not (because 30 is between 22 and 44). 10 is a "big fish" (because no other actor has draw between 10 and 20). 8 is not. 4 is a big fish. 3 and 2 are not.

4. Imagine you hire a star with draw 11. Can you also hire a star with draw 13? Choose the **best** answer.
   - ◯ Yes
   - ◯ No
   - ◯ It depends on the remainder of the instance/solution.

5. Write in the box below the total draw of the optimal set of stars to hire if the available stars have draw: $[3, 8, 20, 2, 14, 6]$. (Write only the total, not the individual draws.)

   Total draw: ☐

6. Below we propose (suboptimal) algorithms to solve this problem. For each algorithm, we provide the draw of a first actor. Fill in the blanks with the draws of the two remaining actors to complete an instance that is a counterexample to the algorithm's optimality. That is, the algorithm will either fail to produce any valid solution or produce a subpotimal solution.

   (a) Iterate through the stars in the order given. For each star, hire them if doing so would not invalidate the so-far-hired set of stars.

   Actor 1: 10. Actor 2: ☐. Actor 3: ☐.

   (b) Sort the stars in decreasing order of draw. Iterate through the sorted stars. For each star, hire them if doing so would not invalidate the so-far-hired set of stars.

   Actor 1: 10. Actor 2: ☐. Actor 3: ☐.

   (c) Sort the stars in decreasing order of draw. Iterate through the sorted stars. Hire the first (biggest) star. Then, for each subsequent star, hire them **unless** they are a big fish (in which case, skip them).

   Actor 1: 10. Actor 2: ☐. Actor 3: ☐.

[1] They're a big fish in a little pond.

3 Blue, Gold, and Green

For its Blue-and-Gold fundraising campaign, UBC has found *anchors*—major donors, each with a *limit* (maximum amount they will donate)—and *causes* to which the anchors will donate.[2] Each cause has a *goal*, the maximum money that will go to the cause. An anchor donates to at most one cause, a cause receives donations from at most one anchor, and the amount of an anchor's donation is the minimum of their limit and the cause's goal. We call this the Blue-and-Gold Problem or BGP. A BGP instance's solution is the maximum dollar amount that can be raised. (Assume limits are distinct, as are goals.)

1. Let the anchors a_1, a_2, a_3, a_4 have limits $10, 100, 30, 40$, respectively. Let the causes c_1, c_2, c_3 have goals $50, 80, 10$, respectively. Fill in the blank below with the maximum amount UBC can raise.

   Maximum amount raised: \$ [ ]

2. Complete the following reduction from BGP to USMP so that it is correct and optimal (if not necessarily efficient). Reminder: USMP is SMP except that there may be fewer women than men.

   If there are [ ], let the women and men be [ ], respectively. Otherwise, let the women and men be [ ], respectively.

   An anchor a_k prefers cause c_i to cause c_j exactly when [ ].

   A cause c_k prefers anchor a_i to anchor a_j exactly when: [ ].

   Given the USMP solution (set of pairs), produce as a solution to the BGP problem:

   [ ]

[2] And which are then named for them, like the Belleville Urban Beautification Project or Wolfman Lunar Research Fund.

For this page, all details of the problem above remain the same (including that each anchor will donate to at most one cause) **except** that we can now assign any number of anchors to a single cause.

3. Let the anchors a_1, a_2, a_3, a_4 have limits $10, 100, 30, 40$, respectively. Let the causes c_1, c_2, c_3 have goals $50, 80, 10$, respectively. Fill in the blank below with the maximum amount UBC can raise.

   Maxmimum amount raised: $ \boxed{}$

4. Below we propose (suboptimal) algorithms to solve this problem. For each algorithm, we have provided a list of anchor limits. Fill in the blanks with two cause goals to complete an instance that is a counterexample to the algorithm's optimality. That is, the algorithm will either fail to produce any valid solution or produce a suboptimal solution.

   Notes: (1) We use "limit" and "anchor" interchangeably. (2) All of these algorithms do the "right" thing when matching an anchor to a goal: They let the anchor donate the minimum of their limit and the money remaining before the goal is met and then update the money remaining for that goal.

   (a) Iterate through the anchors in the order given. For each anchor, assign them to the cause with the most money still left before reaching its goal.

   Anchor 1: 100. Anchor 2: 50. Anchor 3: 200.

   Goal 1: $\boxed{}$. Goal 2: $\boxed{}$.

   (b) Sort the goals in decreasing order. Iterate through the goals, matching each with the largest anchor who has not already been assigned a goal.

   Anchor 1: 100. Anchor 2: 50. Anchor 3: 200.

   Goal 1: $\boxed{}$. Goal 2: $\boxed{}$.

   (c) Sort each of the limits and the goals in decreasing order. Iterate through the anchors, matching each with the first remaining goal. (Remove a goal when its fundraising target has been met.)

   Anchor 1: 100. Anchor 2: 50. Anchor 3: 200.

   Goal 1: $\boxed{}$. Goal 2: $\boxed{}$.

4 Oh Oh Oh

1. Write a good big-O bound on the worst case running time of the following algorithm (as a function of **both** n and m) in the provided box:

```
function intersection1(X, Y)
    Z = { }
    m = length(X)
    n = length(Y)
    for i = 0 to m - 1:
        // assume a typical binary search, which returns
        // -1 if the target element is not found
        if binarysearch(Y, X[i]) != -1:
            add X[i] to Z
```

Worst-case runtime in terms of n and m:

2. `intersection1` is intended to produce a list of the elements that are in both `X` and `Y`. Under what conditions does it work correctly? Fill in the circle next to the **best** choice.

- ◯ when `X` is sorted (but not necessarily `Y`)
- ◯ when `Y` is sorted (but not necessarily `X`)
- ◯ when `X` and `Y` are both sorted
- ◯ for any input lists `X` and `Y`

3. Write a good big-O bound on the worst case running time of the following algorithm (as a function of **both** n and m) in the provided box:

```
function drop(m, n):
    while (m > 0 or n > 0):
        print "(" + m + "," + n + ")"
        if (m > 0):
            m = m - 1
        if (n > 0):
            n = n - 1
```

Worst-case runtime in terms of n and m:

4. Write a good big-O bound on the worst case running time of the following algorithm (as a function of **both** n and m) in the provided box:

```
function intersection2(X, Y)
    Z = { }
    i = j = 0
    m = length(X)
    n = length(Y)
    while i < m and j < n:
        if X[i] < Y[j]:
            i = i + 1
        elsif X[i] > Y[j]:
            j = j + 1
        else:
            add X[i] to Z
            i = i + 1
            j = j + 1
```

Worst-case runtime in terms of n and m: 

5. `intersection2` is **also** intended to produce a list of the elements that are in both `X` and `Y`. Under what conditions does it work correctly? Fill in the circle next to the **best** choice.

   ◯ when `X` is sorted (but not necessarily `Y`)
   ◯ when `Y` is sorted (but not necessarily `X`)
   ◯ when `X` and `Y` are both sorted
   ◯ for any input lists `X` and `Y`

6. Which sentence best characterizes the relative performance of `intersection1` and `intersection2`? Fill in the circle next to the **best** choice.

   ◯ function `intersection1` is almost always slower than function `intersection2`.
   ◯ function `intersection1` is almost always faster than function `intersection2`.
   ◯ each of the two functions can be faster than the other depending on size of the two arrays.

5 O'd to a Pair of Runtimes

The pairs of functions below represent algorithm runtimes on a dense, undirected graph and a simple path in that graph of length k. (A *dense* graph has $m \in \Theta(n^2)$. A *simple path* of length k is a list of $k+1$ vertices where each subsequent pair of vertices is connected by an edge and no vertex appears more than once in the list.) **ASSUME $k > 0$.** For each pair, fill in the circle next to the best choice of:

LEFT: the left function is big-O of the right, i.e., left $\in$ O(right)

RIGHT: the right function is big-O of the left, i.e., right $\in$ O(left)

SAME: the two functions are Θ of each other, i.e., left $\in \Theta$(right)

INCOMPARABLE: none of the previous relationships holds for all allowed values of n and m.

Do not choose **LEFT** or **RIGHT** if **SAME** is true. The first one is filled in for you.

Left Function	Right Function	Answer
n	n^2	**LEFT**
n	k	◯ LEFT ◯ RIGHT ◯ SAME ◯ INCOMPARABLE
k	m	◯ LEFT ◯ RIGHT ◯ SAME ◯ INCOMPARABLE
$n \lg(knm)$	$n \lg(k^4) + \sqrt{m}$	◯ LEFT ◯ RIGHT ◯ SAME ◯ INCOMPARABLE
$2^{\log_{10} m}$	2^n	◯ LEFT ◯ RIGHT ◯ SAME ◯ INCOMPARABLE

Left Function	Right Function	Answer
$(k+m)(n+k)$	m^2	◯ LEFT ◯ RIGHT ◯ SAME ◯ INCOMPARABLE
1	$\frac{m}{\lg k}$	◯ LEFT ◯ RIGHT ◯ SAME ◯ INCOMPARABLE
$n^2 \lg(n^2)$	$k^3 + m \lg m$	◯ LEFT ◯ RIGHT ◯ SAME ◯ INCOMPARABLE
m	k^2	◯ LEFT ◯ RIGHT ◯ SAME ◯ INCOMPARABLE

6 eXtreme True And/Or False

Each of the following problems presents a scenario in a graph and a statement about that scenario. For each one, indicate by filling in the appropriate circle whether:

1. The statement is **ALWAYS** true, i.e., true in *every* graph matching the scenario.
2. The statement is **SOMETIMES** true, i.e., true in some graph matching the scenario but also false in some such instance.
3. The statement is **NEVER** true, i.e., true in *none* of the graphs matching the scenario.

Recall: $|V| = n, |E| = m$. A vertex's *degree* is the number of edges incident on it. For directed graphs, a vertex's *in-degree* is the number of edges ending at the vertex and the *out-degree* is the number starting from it. A *path* of length k is a list of $k+1$ vertices with an edge between each consecutive pair of vertices. A *simple path* repeats no vertex. A *cycle* is a path that starts and ends at the same vertex. A *simple cycle* repeats no vertex other than the starting/end vertex (which only appears twice). Unless stated otherwise, we assume graphs have no self-loops (edges from a vertex to itself).

1. **Scenario:** An undirected graph contains two simple paths that share no edges, each of length k for some integer $k \geq 2$. **Statement:** $n > k + 1$.
   ◯ ALWAYS
   ◯ SOMETIMES
   ◯ NEVER
2. **Scenario:** An undirected graph with $n \geq 2$ and at least $\frac{n^2}{4}$ edges. **Statement:** The graph is connected.
   ◯ ALWAYS
   ◯ SOMETIMES
   ◯ NEVER
3. **Scenario:** A tree (undirected) found by DFS run on a connected, undirected graph with $n \geq 2$. **Statement:** The degree of every node in the tree is two or less.
   ◯ ALWAYS
   ◯ SOMETIMES
   ◯ NEVER

4. **Scenario:** A weakly-connected but **not** strongly-connected, directed graph with $n \geq 2$. **Statement:** A simple cycle of length $n + 1$ exists.
   - ◯ ALWAYS
   - ◯ SOMETIMES
   - ◯ NEVER

5. **Scenario:** A directed graph contains two simple paths that share no edges, each of length k for some integer $k \geq 2$. **Statement:** $n > k$.
   - ◯ ALWAYS
   - ◯ SOMETIMES
   - ◯ NEVER

6. **Scenario:** A connected, undirected graph with $n \geq 3$. **Statement:** The graph includes at least 2^n different simple cycles.
   - ◯ ALWAYS
   - ◯ SOMETIMES
   - ◯ NEVER