CPSC 320 2017W1: Midterm 1 Sample Solution

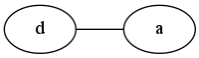
January 26, 2018

Problem reminders:

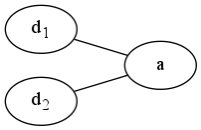
EMERGENCY DISTRIBUTION PROBLEM (EDP)

EDP's input is an undirected, unweighted graph $G = (V, E)$ plus a set of distribution points $D = \{d_1, d_2, \ldots, d_k\}$ each a vertex in V and a single aid location $a \in V$ that is not in D. The output is the number of non-overlapping (edge-wise distinct) paths leading from some d_i to a. (Multiple paths may lead from a single distribution point, and paths may lead from different distribution points.)

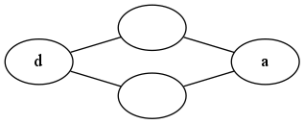
Here are some small sample instances with their solutions, where d, d_1, d_2 are distribution points and a is the aid vertex:



Solution: 1.



Solution: 2.



Solution: 2.

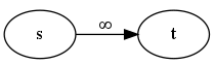
NETWORK BANDWIDTH PROBLEM (NBP)

You have an efficient algorithm to solve the "network bandwidth problem" (NBP). NBP's input is a **weighted**, **directed** graph $G = (V, E)$ (where the tuple $(u, v, w) \in E$ represents a **directed** edge from u to v with **integer weight** w) and designated source and target vertices s and t. A node in the graph is a server and an edge is a network link between servers, weighted by its bandwidth—the maximum number of bytes per second the link can carry. (A weight of ∞ is also allowed, indicating unlimited bandwidth.)

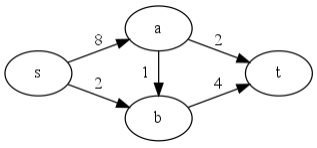
NBP's output is the maximum bandwidth that can be carried from s to t.

Notes: The bandwidth on any link cannot exceed that link's weight. The bandwidth coming **out of** s is unlimited but none can go in, while unlimited bandwidth can go **into** t but none can come out. Otherwise, for any node v, the bandwidth coming into the node must equal the bandwidth coming out. **Assume only integral** (or infinite for links with weight ∞) **amounts of bandwidth can be used on each edge.**

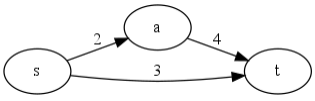
Here are some small sample instances with their solutions and a brief description of how to send the solution bandwidth from s to t. (Note: solving small instances by hand may be helpful, but you do **not** need to know or understand any algorithm to solve this problem.)



Solution: ∞.
∞ on $s \to t$



Solution: 5.
3 on $s \to a$
2 on $s \to b$
1 on $a \to b$
2 on $a \to t$
3 on $b \to t$



Solution: 5.
3 on $s \to t$
2 on $s \to a$
2 on $a \to t$

1 Differential Treatment [13 marks]

1. Consider the execution of the Gale-Shapley algorithm with **women proposing**, and imagine that it maintains a "marker" for every proposing person w, denoting the ranking of the person to whom she will next propose (if she makes another proposal). (I.e. her most preferred match is rank 1, second most preferred is rank 2, etc.)

   Fill in the circles next to the correct choices in order to complete the following narrative which justifies a bound on the worst-case running time of Gale-Shapley. **[5 marks]**

   SOLUTION: Inline below.

   Every iteration of Gale-Shapley moves one proposer's marker to a **LESS** preferred rank.

   Assuming constant time access to the preferences of all participants, each iteration (proposal and acceptance/rejection) requires time $O(1)$ in the worst case.

   Since there are only $O(n^2)$ preferences total, for all proposers, $O(n^2)$ is a(n) **upper** bound on the total running time of Gale-Shapley.

2. Consider this problem: Find and return a pair of any two **different** numbers in an array of n numbers. The array may contain duplicates but **does** contain at least two distinct values. **[3 marks]**

   Fill the circle next to the **best big-O bound** for the worst-case performance of an efficient algorithm to solve this problem if the array is...

   (a) ... unordered: **SOLUTION:** $O(n)$. Our algorithm is to "grab" the first element and then compare it one after another against each subsequent element. As soon as one is different from the first, we return that pair. (Is it efficient? Well, it is if we can lower-bound the problem by $\Omega(n)$. Briefly, proving a lower bound $\Omega(n)$ on the worst case: If we do not look at every item, an adversary can give us only duplicates at each item we do look at and "hide" the one non-duplicate element somewhere we do not look. So, a correct algorithm must in some case look at all the items.)

   (b) ... known to be sorted: **SOLUTION:** $O(1)$. The first and last items must be different from each other. Just return them.

3. **Choose the data structure** for each problem below that most efficiently supports a solution. Choose the **best** answer. If there are multiple best answers, just pick one. **[3 marks]**

   (a) Given a string with $2n$ characters, determine if it is a palindrome (i.e., reads the same forward and backward).

   Chosen data structure: **[1 mark]**

   SOLUTION: stack. The key word here to suggest a stack is "reverse". In practice, you'd probably just access the string from both ends, but if you are going to use a data structure, the stack is clearly the best choice.

   (b) Given an odd query integer q, determine if a sequence of n integers contains two integers that sum to q

   Chosen data structure: **[2 marks]**

   A hash table. Here's a brief two-pass version, although you can do it in one pass instead. (That's not asymptotically faster, but it may be practically a bit faster.)

   Go through the sequence and place each item into the hash table. (We treat the hash table as a set; so, just associate each item with, say, the value *true*.)

Go through each element i again and check if $q - i$ is in the table. If so, return success (because $i + (q - i) = i$). If we never find such an element, return failure.

(A priority queue is a natural choice but not as efficient and requiring a much more complex algorithm. We gave partial credit to that choice because we felt that the hash table based algorithm required a bit too much "sudden during-exam inspiration" for the value and style (multiple choice) of this problem. Stacks and queues are clearly not useful for this problem, but priority queues and hash tables (which actually attend to key values) both at least have potential.)

4. DFS and BFS can produce different trees on the same (undirected, connected) graph depending on which node they are run on and what order children are visited. Which of these stays the same for a given graph regardless of these choices? Fill in the boxes next to **all** that apply: **[2 marks]**

   SOLUTION: The heights of both DFS and BFS trees can change. The height of a DFS tree can change even when run on the **same** node, depending on the order that children are chosen. The height of a BFS tree is consistent on the same node, but when run on a different node can be different. (That's why we need many BFS runs to find the diameter of a graph.)

   However, the number of dashed edges in the BFS/DFS tree remains constant throughout. There's two ways to argue this. One is: The number of edges in any tree is $n - 1$, which doesn't change since the graph doesn't change, and the number of dashed edges is just m minus this quantity, i.e., $m - n + 1$. Since m also doesn't change, the number of dashed edges cannot change. The other argument is: The dashed edges aren't part of the tree anyway. So, there are always 0 dashed edges in the tree regardless. The latter is technically true but doesn't lend much insight :)

2 eXtreme True And/Or False [15 marks]

Each of the following problems presents a scenario and a statement about that scenario. For each one, fill the circle by the best of these choices:

- The statement is **ALWAYS** true, i.e., true in *every* instance matching the scenario.
- The statement is **SOMETIMES** true, i.e., true in some instance matching the scenario but also false in some such instance.
- The statement is **NEVER** true, i.e., true in *none* of the instances matching the scenario.

Then, **justify** your answer as follows:

ALWAYS answer: give a small instance that fits the scenario for which the statement is true and briefly sketch the key point(s) in a proof that the statement is true for all instances that fit the scenario.

SOMETIMES answer: give a small instance that fits the scenario for which the statement is true and a small instance that fits the scenario for which the statement is false.

NEVER answer: give a small instance that fits the scenario for which the statement is false and briefly sketch the key point(s) in a proof that the statement is false for all instances that fit the scenario.

Here are the problems:

1. **Scenario:** Any SMP instance with $n \geq 2$ in which two men share the same preference list. **Statement:** The Gale-Shapley algorithm run with men proposing terminates after exactly n iterations. **[5 marks]**

   SOLUTION: NEVER.

   Proof that statement is false in all instances: Each iteration produces at most one engagement. So, we need at least n iterations to get all men engaged. Two men have the same first preference; so, one of them will get rebuffed or rejected by their shared top choice, "wasting" one of our n proposals without creating an engagement.

   False instance: Any that fits the scenario will do. Here's among the simplest:

   ```
   m1: w1 w2   w1: m1 m2
   m2: w1 w2   w2: m1 m2
   ```

   We need three proposals (with m_2 being rebuffed/rejected within the first two) to finish.

2. **Scenario:** A simple, connected, undirected, unweighted graph with $n \geq 2$. **Statement:** The minimum distance among all longest *simple* paths between pairs of vertices in the graph is equal to the maximum distance among all shortest paths between pairs of vertices in the graph.

   SOLUTION: SOMETIMES.

   Note: a simple graph has no self-loops (edges from a node to itself) and no multi-edges (multiple edges between the same pair of vertices). A "graph loop" or "self-loop" is an edge from a node to itself. It is **NOT** the same thing as a cycle. (Every self-loop is (or at least allows) a teeny cycle, but any simple cycle of length greater than 1 is not a self loop.)

   Note also that the "maximum distance among all the shortest paths" is just the diameter.

   Finally, note that it doesn't matter whether we say "simple" or not for the shortest path. The shortest path is always going to be simple. Repeated vertices along the path will just be a waste, making the path longer.

   True instance: In fabulous ASCII art: `A -- B`. In other words, a two node, connected, undirected graph. The only simple path of length 1 between the only pair of nodes is both the minimum longest and maximum shortest path.

   False instance: Also in fabulous ASCII art: `A -- B -- C`. A line of three vertices. The diameter is 2 from `A` to `C`. The minimum among all the longest simple paths is the longest simple path from either `A` to `B` or `B` to `C`, which is 1. (A complete three node graph also works, since then all the longest simple paths between vertices are of length 2 while the diameter is 1.)

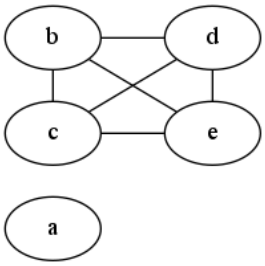
3. **Scenario:** A simple, undirected graph (with no self-loops) with $n \geq 2$ and $m \geq \frac{n^2}{5}$. **Statement:** The graph is connected. **[5 marks]**

   SOLUTION: SOMETIMES.

   True instance: In fabulous ASCII art: `A -- B`. $n = 2$, and $m = 1$, which is greater than or equal to $\frac{n^2}{5} = \frac{4}{5}$. (Specifically, m is greater than $\frac{4}{5}$.) However, the graph is connected.

   False instance: This one requires a bit more than ASCII art. The insight here is that we can "set aside" a bit of the graph to be the disconnected part (or even set aside large subgraphs) and still have **many** edges by making the remaining part dense. To make this work, we need to go all the way up to $n = 5$, at which point $\frac{5^2}{5} = 5$. (It's not critical that this be an integer; that's a coincidence. See the previous instance for a non-integral case where the math on the "scenario" works just fine.)

   Here's our graph:



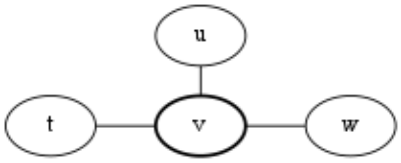
We put in one extra edge because it makes a pleasing little clique of four nodes at the top. (A *clique* is a complete subgraph.) You could delete any one of the edges and still have a true instance.

Note that $m = 6$, which is greater than or equal to 5 as required. However, the graph is disconnected.

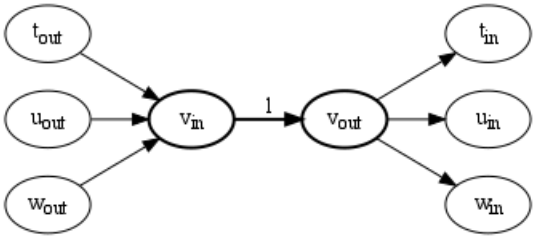
3 Slapping on a Bandwidth-Aid [12 marks]

Consider a variant of EDP that we call VDP (vertex-disjoint path problem). The input is the same, but the output is the maximum number of distinct, **vertex-disjoint** paths from distribution vertices to the aid vertex. Specifically, no two paths from distribution vertices to aid vertex can share any vertex **except** their end point (the aid vertex) and potentially their start point (if they begin at the same distribution point but travel different paths to the aid vertex).

We now describe a solution to VDP via reduction to NBP. The core of the reduction is to limit travel "through" a vertex by transforming a vertex like v in VDP (shown with only its immediate neighbors):



into a pair v_{in} and v_{out} in NBP:



Here is our (on track but broken) reduction:

Convert an instance of VDP to an instance of NBP:

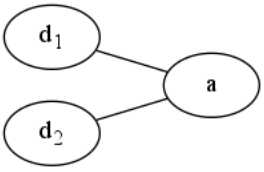
1. Generate new vertices in V_{NBP} as follows:
   (a) For the aid vertex a, generate a vertex $a_{in} \in V_{NBP}$.
   (b) For each distribution vertex $d \in D$, generate a vertex $d_{out} \in V_{NBP}$.
   (c) For each other vertex $v \in V_{VDP}$, generate two vertices $v_{in}, v_{out} \in V_{NBP}$ **and** an edge $(v_{in}, v_{out}, 1) \in E_{NBP}$.
2. For each undirected edge $(u, v) \in E_{VDP}$, if possible generate two edges: $(u_{out}, v_{in}, \infty)$ and $(v_{out}, u_{in}, \infty)$ in E_{NBP}, skipping cases where a corresponding vertex does not exist.
   (E.g., an edge (v, a) with the aid vertex can produce only an edge from v_{out} to a_{in}, since a_{out} does not exist.)
3. Generate one additional vertex v' in V_{NBP}.
4. For each distribution point $d \in D$, generate an edge $(v', d_{out}, \infty) \in E_{NBP}$.

5. Finally, let $s = v'$ and $t = a_{in}$.

Convert a solution to NBP to VDP:

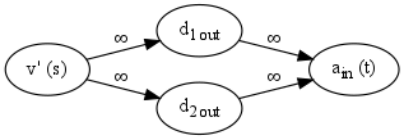
Let the solution to VDP be the solution to NBP.

On the next pages, you consider and comment on small VDP instances and this reduction. In each instance, the vertices labelled d_1 or d_2 are **distribution points**, the vertex labelled a is the **aid vertex**, and others are "regular" vertices.

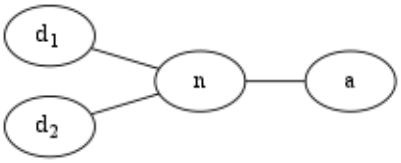


1. Consider this VDP instance, for which the correct solution is 2:

   (a) Draw the NBP instance created by the reduction from this VDP instance. **[3 marks]**
   SOLUTION: Below. Note that we annotated in parentheses which node is the source s and which is the target t.

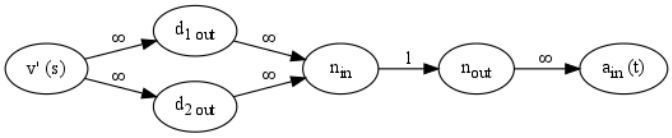


   (b) Give the solution to this instance produced by the reduction:
   SOLUTION: ∞, which is an error.



2. Consider this VDP instance, for which the correct solution is 1:

   (a) Draw the NBP instance created by the reduction from this VDP instance. **[4 marks]**
   SOLUTION: Below. Note that we annotated in parentheses which node is the source s and which is the target t.

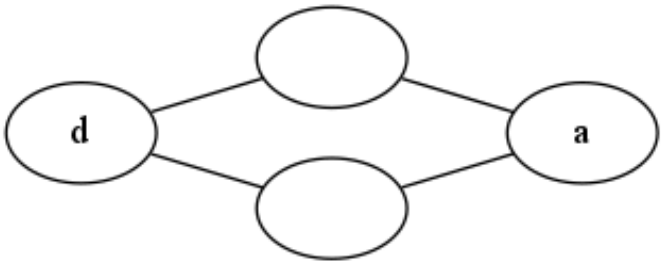


   (b) Give the solution to this instance produced by the reduction:
   SOLUTION: 1, which is correct.

3. The instances above highlight a problem with the reduction. Fill in the blanks below to describe the **very small** change needed to fix the reduction on the previous page. **[3 marks]**

   SOLUTION: Change the ∞ weights in step 2 to 1's.

   Note that changing the ∞ weights in step 4 to 1's makes the solution to the instance above correct but does so by **INTRODUCING A BUG** into the reduction. Consider a graph like this:



   The incorrect reduction of reducing the step 4 weights to 1's will cause the source v' to have a weight-1 edge to d_{out}. That's the **only** edge from the source. So, the maximum possible bandwidth from s

to t is 1 (and indeed that'll be the answer to this instance). However, the **correct** answer here is 2 instead. With ∞ weight on the source edge, we can reach that answer of 2.

What we're really encoding by changing ∞ to 1 in step 2 is that a vertex-disjoint path must also be edge-disjoint. (After all, if we repeat an edge across two paths, we must repeat the vertices on either end of that edge.) So, there's no point in allowing more than a bandwidth of 1 to pass across one of these edges we introduce.

However, longer solutions than ours did exist that made more restricted changes to step 2. As long as you constrained all $d_{out} \to a_{in}$ style edges to be of weight 1, your solution would work.

That said, it is **NOT** enough to just say that (of the two produced edges) we only need $(u_{out}, v_{in}, \infty)$ to be changed. Remember that EDP edges are undirected. Thus, you don't know if the edge from a distribution point to an aid vertex looks like (d, a) or (a, d).

4 BONUS

Sample solutions to bonus problems? Nah.. how about you post yours instead? ☺