Section 1: Solutions

Review of Graph Concepts

- **Degree:** The number of edges connected to a vertex.
- **Acyclic Graph:** A graph without cycles.
- **Tree:** An undirected, acyclic graph.
- **Path:** A list of vertices $v_0, v_1, ..., v_k$ in a graph such that (v_i, v_{i+1}) is an edge in the graph for all $0 \leq i < k$.

1. Gale-Shapley

Consider the following stable matching instance:

$r_1 : h_3, h_1, h_2, h_4$
$r_2 : h_2, h_1, h_4, h_3$
$r_3 : h_2, h_3, h_1, h_4$
$r_4 : h_3, h_4, h_1, h_2$

$h_1 : r_4, r_1, r_3, r_2$
$h_2 : r_1, r_3, r_2, r_4$
$h_3 : r_1, r_3, r_4, r_2$
$h_4 : r_3, r_1, r_2, r_4$

(a) Run the Gale-Shapley Algorithm with riders proposing on the instance above. When choosing which free rider to propose next, always choose the one with the smallest index (e.g., if r_1 and r_2 are both free, always choose r_1).

Solution:

The steps of the Gale-Shapley Algorithm with the rider with lowest index proposing first:

r_1 chooses h_3	(r_1, h_3)
r_2 chooses h_2	$(r_1, h_3), (r_2, h_2)$
r_3 chooses h_2	$(r_1, h_3), (r_3, h_2)$
r_2 chooses h_1	$(r_1, h_3), (r_2, h_1), (r_3, h_2)$
r_4 chooses h_3	$(r_1, h_3), (r_2, h_1), (r_3, h_2)$
r_4 chooses h_4	$(r_1, h_3), (r_2, h_1), (r_3, h_2), (r_4, h_4)$

(b) Run the Gale-Shapley Algorithm with riders proposing on the same instance. But now, when choosing which free rider to propose next, always choose the one with the largest index. Do you get the same result?

Solution:

The steps of the Gale-Shapley Algorithm with the rider with highest index proposing first:

r_4 chooses h_3	(r_4, h_3)
r_3 chooses h_2	$(r_3, h_2), (r_4, h_3)$
r_2 chooses h_2	$(r_3, h_2), (r_4, h_3)$
r_2 chooses h_1	$(r_2, h_1), (r_3, h_2), (r_4, h_3)$
r_1 chooses h_3	$(r_1, h_3), (r_2, h_1), (r_3, h_2)$
r_4 chooses h_4	$(r_1, h_3), (r_2, h_1), (r_3, h_2), (r_4, h_4)$

We ended up with the same result!

(c) Now run the algorithm with horses proposing, breaking ties by taking the free horse with the smallest index. Do you get the same result?

Solution:

The steps of the Gale-Shapley Algorithm with horses proposing:

h_1 chooses r_4	(r_4, h_1)
h_2 chooses r_1	$(r_1, h_2), (r_4, h_1)$
h_3 chooses r_1	$(r_1, h_3), (r_4, h_1)$
h_2 chooses r_3	$(r_1, h_3), (r_3, h_2), (r_4, h_1)$
h_4 chooses r_3	$(r_1, h_3), (r_3, h_2), (r_4, h_1)$
h_4 chooses r_1	$(r_1, h_3), (r_3, h_2), (r_4, h_1)$
h_4 chooses r_2	$(r_1, h_3), (r_2, h_4), (r_3, h_2), (r_4, h_1)$

No, the result is different when we have the horses propose as opposed to the riders.

2. A Quick Proof

Is it possible to have a stable matching instance with more than 2 stable matchings? If so, give an instance and at least 3 stable matchings. If not, prove that every instance has at most 2 stable matchings.

Solution:

Consider the following instance:

$r_1 : h_1, h_2, h_3, h_4$
$r_2 : h_2, h_1, h_4, h_3$
$r_3 : h_3, h_4, h_1, h_2$
$r_4 : h_4, h_3, h_2, h_1$

$h_1 : r_2, r_1, r_4, r_3$
$h_2 : r_1, r_2, r_3, r_4$
$h_3 : r_4, r_3, r_2, r_1$
$h_4 : r_3, r_4, r_1, r_2$

This instance has four stable matchings:
$(r_1, h_1), (r_2, h_2), (r_3, h_3), (r_4, h_4)$
$(r_1, h_1), (r_2, h_2), (r_3, h_4), (r_4, h_3)$

$(r_1, h_2), (r_2, h_1), (r_3, h_3), (r_4, h_4)$
$(r_1, h_2), (r_2, h_1), (r_3, h_4), (r_4, h_3)$

3. Induction Review

Consider the following claim:

Let $P(n)$ be "Every tree with at least n nodes has at least two nodes of degree-one."

(a) What is the correct "skeleton" of the inductive step (i.e., the right things to assume and the right target)?
Solution:

We must start with "Let T' be an arbitrary tree with $k+1$ nodes."

Our conclusion will be that T' has at least two nodes of degree-one, so $P(k+1)$ holds.

(b) Prove the claim by induction.
Solution:

Let $P(n)$ be "Every tree with at least n nodes has at least two nodes of degree-one." We prove the claim by induction on n.

Base Case: $n = 3$. There is only one undirected tree with three nodes. It has two nodes of degree-one.

Inductive Hypothesis: Suppose $P(n)$ holds for $n = 3, ..., k$ for an arbitrary $k \geq 3$.

Inductive Step: Let T' be an arbitrary tree with $k+1$ nodes. Let u be a vertex of T' of degree-one (this first vertex exists by the fact), and call its neighbor v. Let T'' be the tree created by deleting u from T'.

Observe that, since u was degree-one, the only simple paths that used (u, v) had u as an endpoint (as once we use (u, v) to arrive at/leave u we cannot reuse it to leave/arrive). Thus T'' is still a connected tree, and we can apply the IH to T'' to conclude there are at least two vertices w_1, w_2 of T'' that are degree-one.

We now find the two degree-one nodes in the original tree T'. We know that u has degree-one (and is not the same as w_1 or w_2 since u was deleted to create T''). Since u has degree-one, it can only attach to one of w_1, w_2, thus at least one (the other one) of w_1, w_2 is an additional node of degree-one, as required.

Therefore, T' has the required degree-one vertices. Since T' is an arbitrary tree with $k+1$ vertices, we have shown $P(k+1)$.

4. Find the Bug: Failed Induction

In this problem you will fix an incorrect induction proof.

Let's do a little bit of problem setup. Suppose you have a stable matching instance with n horses and n riders. Of the n horses, 5 of the horses are **popular**. That is, every rider's list has those 5 horses as their first 5 choices (in some order, not necessarily the same for each rider). Similarly, you have 5 **popular** riders, such that every horse has those 5 riders as their top choices.

Let $P(n)$ be "In every stable matching instance with n horses, n riders, of which 5 horses and 5 riders are popular: in every stable matching, popular horses are matched only to popular riders."

Spoof. We will show $P(n)$ holds for all $n \geq 5$ by induction on n.

Base Case $(n = 5)$
With 5 horses and riders, every horse and rider is popular. Since every stable matching pairs every agent, every agent is matched to a popular agent.

Inductive Hypothesis: Suppose $P(n)$ holds for $n = 5, ..., k$ for an arbitrary integer $k \geq 5$.

Inductive Step: Let $h_1, ..., h_k$, $r_1, ..., r_k$ be k horses and riders, with $h_1, ..., h_5, r_1, ..., r_5$ being the popular agents. We add agents h_{k+1} and r_{k+1}. By popularity, h_{k+1} has $r_1, ..., r_5$ (in some order) as its 5 favorite agents and r_{k+1} has $h_1, ..., h_5$ (in some order) as their 5 favorite agents. Further, let r_{k+1} and h_{k+1} be each other's 6^{th} choices (i.e. top choice outside the popular riders.

Now, consider any stable matching in the old (size-k) instance, and create a stable matching for the new instance by pairing h_{k+1} with r_{k+1}.

We now show that this matching is stable for the new instance. Since it was stable for the small instance, the only possible blocking pairs must involve h_{k+1} or r_{k+1}. By IH, every popular agent is matched to another popular agent. Regardless of where h_{k+1} and r_{k+1} was added to the popular agent's list, they fall after the popular agents, so h_{k+1} and r_{k+1} cannot form a blocking pair with the popular agents. And since they have each other as their next choices, they cannot form a blocking pair with anyone else. Thus we have that there are no blocking pairs, and the matching is stable. The popular agents remain matched to each other, as required. □

(a) There are at least two errors in this proof. Describe them! **Solution:**

The first mistake is in the setup of the inductive step. We need to show a claim for every instance of size $k + 1$. Instead we build a particular instance of size $k + 1$. In this example, the mistake is quite fundamental – there is no reason in the problem that r_{k+1} and h_{k+1} should have each other as their 6^{th} choices. That is, if we introduced a recursive definition of stable matching instances and changed this to structural induction, we would have more steps to do beyond the one here (In contrast to the tree problem where all the cases are actually handled).

The second bug is again a mistake with handling a for-all. This time, the quantifier on the "every stable matching" part of the statement. We don't check every stable matching! We check every matching we built by starting with a stable matching on the small instance – how do we know there aren't stable matchings where r_{k+1} is matched to h_4 (for example)? We need to start with an arbitrary stable matching and argue whether the popular agents are matched or not.

(b) Write a correct proof of this claim. Do NOT use induction. Use a proof by contradiction instead. **Solution:**

Suppose, for the sake of contradiction, that there is a stable matching instance with 5 popular riders and horses, and there is a stable matching M for this instance so that some popular horse h is not matched to a popular rider. Since there are the same number of popular horses and riders, there is a popular rider, r, that is also not matched to a non-popular agent. We claim that r and h form a blocking pair. Indeed, since each is popular, they are each in the top 5 of each other's lists, but each is matched to a non-popular agent, which must be 6^{th} or lower on both lists. Thus r and h would rather be with each other than with their matches, so they form a blocking pair. But M was supposed to be a stable matching. A contradiction! So every popular horse must be matched to a popular rider.

5. Proof Practice: Proving Code Correct

Recall Dijkstra's Algorithm from 332.

```
Dijkstra(Graph G, Vertex source)
     initialize distances to ∞, source.dist to 0
     mark all vertices unprocessed
     initialize MPQ as a Min Priority Queue
     add source at priority 0
     while(MPQ is not empty){
          u = MPQ.removeMin()
          foreach(edge (u,v) leaving u){
```

```
            if(u.dist+weight(u,v) < v.dist){
                  if(v.dist == ∞) //if v not in MPQ
                        MPQ.insert(v, u.dist+weight(u,v))
                  else
                        MPQ.decreaseKey(v, u.dist+weight(u,v))
                        v.dist = u.dist+weight(u,v)
                        v.predecessor = u
            }
      }
      mark u as processed
}
```

Consider the following claim:

For all n, the n^{th} time a vertex, v, is removed from `MPQ`, `v.dist` contains the true shortest path distance from `source` to v.

(a) Prove the claim by induction. **Solution:**

We prove the claim by induction on n.

Base Case: $n = 0$, before any iterations, `source` is the closest vertex (at distance 0 – it is the closest, because all edge weights are non-negative). The distance from `source` to itself is 0, as stored.

IH: Suppose for the first k iterations, at the top of the while loop, the vertex removed from `MPQ` had the correct distance.

IS: Now consider the closest vertex at the $(k+1)^{\text{st}}$ iteration, `u`. The shortest path from `source` to `u` visits a vertex w right before `u`.

w is already processed (it is closer to `source` than `v`, because there are no negative edge weights) – when it was, by IH the distance from `source` was correct. We then updated `u.dist` with the distance from `source` to w plus the edge weight from w to `u`. Observe that this is the length of the path from `source` to `u`. We will not update `u.dist` again, because if we ever see another edge ending at `u`, the source of that edge is a vertex being processed, and thus Dijkstra's knows the correct distance to that vertex (and would be updating with the distance along the path through the other vertex). Thus when `u` is closest at the start of the $k+1)^{\text{st}}$ iteration, we have the true shortest path distance.

thus by the principle of induction, every vertex removed from `MPQ` had the true distance when it was removed.

(b) Prove the claim by contradiction. **Hint:** It's *extreme*ly helpful to think about the closest vertex where something is wrong, rather than just any old iteration.
During your proof, you may use the following fact without proof: Every non-infinite value of `v.dist` is the length of a path from `v.dist` (not necessarily the shortest path). **Solution:**

Suppose for the sake of contradiction, that there is a vertex which, when removed from `MPQ` has the wrong distance, and let u be the closest such vertex (in true distance terms). Let v be the vertex before u on the shortest path from `source` to u. Since `u.dist` only decreases, and is always the length of a `source`-to-u path, we note that `u.dist` must have been more than its true value. Thus v (which is closer to the source) had a smaller value of dist when processed than `u.dist` did. So, v must have been processed before u (since we are processing in increasing order of `dist`). Since u was the first vertex with an incorrect distance, `v.dist` had the correct value when processed. But since v had the correct distance when processed, v would have updated `u.dist` to `v.dist` $+ w(u, v)$. But that's exactly the distance from `source` to u. So u had the correct distance value in it. Since there are no negative length edges and every non-infinite value of `dist` is a true path, we will never change the value again, so it must have been correct when processed. A contradiction!

Thus the distance is always correct when removed from the priority queue.

6. Practice A Reduction

You have a set of r riders and h horses, but unfortunately, $2h < r < 3h$, i.e. there are many more riders than horses. You wish to setup a set of 3 rides which will give each rider exactly one chance to ride a horse. To keep things fair among the horses, you wish for each to have exactly 2 or 3 rides.

Because it's winter, by the time the third ride starts it will be very dark, so every rider would prefer *any* horse on the first two rides over being on the third ride. Between the first two rides, each rider doesn't have a preference over time of day, and have the same preference over horses. If a rider must be on the third ride, it has the same preference list for that ride as well.

Each horse has a single list over riders, which doesn't change by ride. Since horses love their jobs, they prefer to being one of the horses on the third ride to one of the ones left home.

Design an algorithm which calls the following library **exactly once** and ensures there are no pairs r, h which would both prefer to change the matching and get a better result for themselves.

`BasicStableMatching`
Input: A set of k horses and k riders. Each horse has a preference list of all k riders, and each rider has a preference list of all k horses.
Output: A stable matching among the k horses and k riders.

(a) Give a 1-2 sentence summary of your idea.

(b) Give the algorithm you're going to run.

(c) Give a 1-2 sentence summary of the idea of your proof.

(d) Write a proof of correctness.

(e) Give the running time of your algorithm; briefly justify (1-3 sentences)

Solution:

(a) Create an instance where each horse has 3 copies (one per ride) and extra "fake" riders to balance the sides. Modify the preference list to represent what the agents want in the problem.

(b) We will create a `Basic` instance with $3h$ riders and $3h$ horses. For each horse h_i in the original instance, create three horses $h_i^{(1)}, h_i^{(2)}, h_i^{(3)}$. Each copy of the horse starts with h_i's original list.

For each rider r_j, create a list as follows: from r_j's original list, put $h_i^{(1)}$ followed by $h_i^{(2)}$ in place of h_i in the original list. Then at the end, add another copy of the original list with each h_i replaced by $h_i^{(3)}$.

To make the total number of riders $3h$, add "dummy" riders[a] $d_1, ..., d_\ell$ until the number of riders and horses is equal. Each dummy will have a list of the $h_i^{(3)}$, followed by the $h_i^{(2)}$ and $h_i^{(1)}$ (the h_i can be in any order relative to each other, as long as the time-of-day ordering is followed). Finally add the dummies to the end of the lists of all horses (in any order).

We now have an instance with $3h$ riders and $3h$ horses, and every list contains all the other agents. Run the `BasicStableMatching` algorithm, then delete the dummy riders, and leave any horse whose partner was deleted unmatched.

(c) The `Basic` algorithm doesn't produce blocking pairs, so we won't either (once we delete the dummies)

(d) We claim the result is a correct assignment. First, observe that each (real) rider is matched, and no horse

is free on the first two rides. Since each horse prefers the real riders to the dummies and each rider prefers any of the first two rides to the third, a dummy rider matched with a horse on the first two rides would have created a blocking pair (the horse on the first two rides with any rider assigned to the third ride). Thus no horse is free on the first two rides.

It remains to show there is no blocking pair among matched agents. Suppose, for contradiction, there is a pair r, h_i where r and h_i would both prefer to be paired on ride j (over their current state). Then, by construction of the lists, r prefers $h_i^{(j)}$ on its preference list and $h_i^{(j)}$ prefers r on its preference list. This would have been a blocking pair for the `Basic` instance. But the algorithm produces a stable matching, which by definition has no such blocking pairs, a contradiction!

(e) $\Theta(h^2)$. We have $3h$ agents on each side, so the guarantee on `BasicStableMatching` gives a $\Theta(h^2)$ guarantee for that call. All the other operations (copying lists, creating agents, etc.) can be done in time linear in the size of the final instance (since it's just copy-pasting) which is also $\Theta(h^2)$ ($\Theta(h)$ agents, each with lists of length $\Theta(h)$).

[a] a "dummy" is like a dummy for clothing (a mannequin) it *looks like* a real rider, but doesn't actually represent a real rider. Just like a mannequin looks like a real person but isn't one. Dummies are a very common tool in reductions.