Homework 1

Michael Wurtz and Frederick Robinson

13 January 2010

1 Problem 1.4

We will adapt the *Gale-Shapley* algorithm to this problem. Consider the following algorithm

1. Initially all hospitals and all students are free.
2. While there is a free hospital who has not offered to every student, pick a free hospital.
    (a) The hospital offers to the first student in it's preference list it has not offered to yet.
    (b) If the student has not agreed to work at a hospital yet he agrees automatically.
    (c) If he is already working for another hospital, he chooses whichever hospital he has higher preference for.
        i. If he likes the one he has agreed to work for already he stays with it, and the hospital asks the next most preferred candidate on its list.
        ii. If he prefers the offering hospital, he switches to it. Mark the hospital he switched away from as free, and if the hospital he moved to now has its preferred number of students, mark it as not free.

*Proof:*To begin we prove a lemma

Lemma 1.1. *Once a student has been made an offer he is always assigned to precisely one hospital, and moreover he prefers whatever hospital he is assigned to at least as much as all other hospitals that have offered to him.*

Proof. The only time a student is offered is in 2(c)ii, and in this case he always chooses the hospital he prefers. □

Now I claim that this algorithm will always produce a stable assignment of students to hospitals

Proof. We show that there are no instabilities of the first type

We proceed by induction:

Base Case: Clearly, when the algorithm starts there are no such instabilities as no students are yet assigned to hospitals.

Inductive Hypothesis: In any given assignment the algorithm will not create an instability of the first type.

Since a hospital asks students in order of its preference and if a student is not yet assigned to a hospital he automatically accepts the offer the algorithm cannot create an instability of the first type.

For, by Lemma 1.1 once a student is assigned he stays assigned. Thus, when a hospital gains a new student it must be that each student which it prefers more is already assigned to a hospital.

Now it remains to show that the algorithm does not produce matchings with instabilities of the second type. We'll use induction again

Base Case: Clearly, when the algorithm starts there are no such instabilities as no students are yet assigned to hospitals.

Inductive Hypothesis: In any given assignment the algorithm will not create an instability of the second type. If a student s has been assigned to a hospital h then the hospital h has already offered to each student s' which it prefers more than s. If s' prefered h to its current hospital h' or is unassigned he would have accepted. After accepting, h' can only improve his preference, and from then on, will prefer wherever he is working to h.

Thus, again the algorithm can produce no such instabilities. □

Furthermore I claim that the algorithm always produces a perfect matching

Proof. Suppose towards a contradiction that the algorithm does not produce a perfect matching. In particular suppose that when the algorithm terminates there is a hospital, say h which does not have as many students as it wants.

h must have made an offer to each student, otherwise the algorithm would not have ended, however by Lemma 1.1 once a student has been made an offer he stays assigned.

This is a contradiction since, by assumption the number of available spots is less than the number n of students, and again by Lemma 1.1 each student that h made an offer to must be assigned to precisely one hospital. □

Complexity: The algorithm runs in $O(n \cdot m)$ time as each hospital offers a position to each student only once.

2 Problem 2.3

$$f_2(n) = \sqrt{2n}$$

$$f_3(n) = n + 10$$

$$f_6(n) = n^2 \log n$$

$$f_1(n) = n^{2.5}$$
$$f_4(n) = 10^n$$
$$f_5(n) = 100^n$$

We will demonstrate the inequality on growth rates for each successive pair of functions. Since we have transitivity this will suffice to show that a given function in the list grows more slowly than each function following it.

1. $f_2(n)$ is $O(f_3(n))$

   $f_2(n) = \sqrt{2n} \Rightarrow f_2(n) \leq \sqrt{2}n$ since multiplying by $\sqrt{n} \geq 1$ is an increasing transformation. Similarly, $f_2(n) \leq \sqrt{2}n + 10\sqrt{2}$. But clearly, $\sqrt{2}(n+10)$ is $O(f_3(n))$ since $6f_3(n) = 6(n+10) \geq \sqrt{2}(n+10)$ for every $n \geq 1$. Since we have shown that some function $\sqrt{2}(n+10) > f_2(n)$ is $O(f_3(n))$ we know that $f_2(n)$ is $O(f_3(n))$

2. $f_3(n)$ is $O(f_6(n))$

   Observe that $f_3(n) = n+10 \leq n+10n \leq 11n^2$. Hence, since the logarithm function is always greater than 1 past some n_0 we have further that $f_3(n) \leq 12n^2 \log n$ for all $n \geq n_0$. Then $f_3(n)$ is $O(f_6(n))$ by definition.

3. $f_6(n)$ is $O(f_1(n))$

   There exists some n_0 such that the function $\sqrt{n}$ is greater than $\log n$ given $n \geq n_0$. Thus, for each $n > n_0$ we have that $f_6(n) = n^2 \log n \leq n^2\sqrt{n} = n^{2.5}$ and so $f_6(n)$ is $O(f_1(n))$ by definition.

4. $f_1(n)$ is $O(f_4(n))$

   We can show that $10^n > n^3$ for all $n > 1$ fairly easy by induction. *Base Case*: For $n = 5$ we we have $10^4 = 100000 > 125 = 5^3$. *Inductive Hypothesis*: If $10^n > n^3$ for some n we can show that $10^{n+1} > (n+1)^3$. For $10^{n+1} > (n+1)^3 \Leftrightarrow 10(10^n) > 1 + 3n + 3n^2 + n^3$ and since we are assuming that $10^n > n^3$ and we know that after some $n_0 = 4$ we have $1 < n^3, 3n < n^3, 3n^2 < n^3$ we can further deduce that after some n_0 then $10(10^n) > 4(10^n) > 1 + 3n + 3n^2 + n^3$. So we have shown that for $n \geq n_0 = 5$ the function 10^n is strictly greater than n^3. Thus, $f_1(n)$ is $O(f_4(n))$ as claimed.

5. $f_4(n)$ is $O(f_5(n))$

   Since composition with the function $g(x) = x^2$ increases the value of any input $x > 1$ and $f_4(n) = 10^n > 1$ for all $n > 0$ we have in particular $f_4(n) = 10^n < (10^n)^2 = 100^n$. So clearly we have $f_4(n)$ is $O(f_5(n))$ as desired.

3 Problem 2.7

Given such a song one of the best ways to compress it into a script is to store each new line in order (but only once). This sort of script would be translated back into the words of the song by first reading the first line, then restarting, and reading until you reach the next new line. This process would be repeated, restarting the song every time a new line is reached until the entire song had been read through once without stopping.

This method of compression is optimal (without resorting to some sort of compression on the contents of the lines themselves) as it contains the least amount of information necessary to reconstruct the song. Suppose there were a more efficient method of compressing the song. Then it would necessarily contain less information than this method on some input. This is however impossible for that would entail leaving out some of the song.

If the length of a line k of a song is given by $l_k < c$ we can compute the length of the entire song as

$$n = \sum_{p=1}^{k} \sum_{m=1}^{p} l_m$$

however the length of the script that is generated by just specifying the new lines is only

$$f(n) = \sum_{m=1}^{k} l_m$$

For a fixed script length $f(n)$ the size of the song generated by this script increases as we decrease the length of the lines. Thus the size of the compressed script relative to that of the song increases as the line length increases. So, in the worst case we have $l_k = c$ for each k. Now we can evaluate the summations and get more concrete information about the compression ratio.

So, in the worst case we compute

$$n = \sum_{p=1}^{k} \sum_{m=1}^{p} l_m = \sum_{p=1}^{k} \sum_{m=1}^{p} c = \frac{1}{2}ck^2 + \frac{1}{2}ck$$

$$f(n) = \sum_{m=1}^{k} l_m = \sum_{m=1}^{k} c = kc$$

Thus we get the compression ratio in the worst case as

$$\frac{\text{Comressed}}{\text{Uncompressed}} = \frac{kc}{\frac{1}{2}ck^2 + \frac{1}{2}ck} = \frac{2}{k+1}$$

In particular we can check that in the case where $k = 1$, that is, when there is only one verse the compression ration is 100% as we would expect.

Another way of saying this is: Given $o = \frac{n}{c}$ lines of input the compression yields in the worst case

$$\begin{aligned} n &= \frac{1}{2}ck^2 + \frac{1}{2}ck \\ \Leftrightarrow 2\frac{n}{c} &= k^2 + k \\ \Leftrightarrow 0 &= k^2 + k - 2\frac{n}{c} \\ \Leftrightarrow k &= \frac{-1 \pm \sqrt{1 - 4(-2\frac{n}{c})}}{2} \\ \Leftrightarrow k &= \frac{-1}{2} \pm \frac{1}{2}\sqrt{1 + 8\frac{n}{c}} \\ \Leftrightarrow k &= \frac{1}{2}\left(-1 \pm \sqrt{1 + 8\frac{n}{c}}\right) \end{aligned}$$

We may discard the negative solution, showing that we end up with $\frac{1}{2}\left(-1 + \sqrt{1 + 8o}\right)$ lines of output.

Finally we conclude that our compression is $O(\sqrt{n})$.

4 Problem 3.2

Consider the following algorithm

Perform a depth first search, marking nodes as we go

1. For all nodes: if node is marked, continue to next node. Otherwise, mark current node as "searching" and do the following for all nodes connected by an edge to this node:
   (a) If marked "searching" there is a cycle. Remember current node and do the following while not at the remembered node (terminate when it reaches the remembered node)
      i. Output current node
      ii. Back up one node using the stack from the depth first search
   (b) If not marked
      i. Repeat search (go one level deeper with dfs) on current node, travel down all edges but the one we came from.
      ii. Change marking to "visited"
   (c) Go back to previous node (backing up in the depth first search)
2. If the algorithm searches the entire graph but has not terminated, terminate and report that there is no cycle

Correctness: This algorithm will visit all nodes at least once, as the first *for* loop checks all nodes. Furthermore we will prove that if the algorithm ever reaches a "searching" node, it has correctly found a cycle.

Lemma 4.1. *If the algorithm encounters a "searching" node then it has found a cycle*

Proof. If the algorithm reaches a node marked "searching" it must be that it has progressed through a series of edges from that node back to itself since the only nodes marked "searching" are connected by a path to the current node. □

Proof. There are two cases

Case 1: There is a cycle

Assume there is a cycle, that is, a list $\{v_1, v_2, \ldots, v_n\}$ so there is an edge (v_i, v_{i+1}) for all $i \neq n$ and an edge $(n, 1)$.

This algorithm must either terminate by finding another cycle (which is fine), or at some point one of the nodes in the cycle list above is visited. This one node will be marked searching.

A depth first search will visit all connected nodes before going back up a level. Since the search reaches every node, it must eventually go full circle and find the first node in the cycle marked "searching" (again assuming it does not find other connected cycles).

Case 2: There is no cycle

In this case the program will not falsely claim it has found a cycle (by Lemma 4.1).

The program must terminate as it only searches each node once. Thus it must terminate with no cycle □

Runtime: We know that the depth first search only follows a given edge once. Once the search has followed an edge it has visited the node on either end of the edge. This means that it has marked these nodes, and will not revisit them. The first *for* loop guarantees that all nodes (even disconnected portions of the graph) will be visited. It will only check each node once to make sure that it has been visited. Thus the algorithm is $O(n + m)$ as desired.

5 Problem 3.9

For any graph, if there exists a path of length $n/2$ (where n is the total number of nodes in the graph) from s to t then there is some node on this path, so that, if it is removed there is no longer a path of lenth $n/2$ from s to t.

Proof. If $G = (V, E)$ is a graph, let $M(G, n, n')$ be a path of minimum length from n to n', that is, a set $\{n_1, n_2, \ldots, n_k\}$ of minimum size so $(n, n_1) \in E, (n_i, n_i + 1) \in E \; \forall 1 \leq i < k, and (n_k, n') \in E$.

Suppose G is a graph with n nodes, including s and t so $d = |M(G, s, t)|$, and $d + 1 > n/2$. (The number of nodes between two nodes plus one is the distance).

For any node p, let $G - p = (V - p, E - (\text{set of all edges with p}))$

Assume towards a contradiction that for any node $p \in V, M(G - p, s, t)$ exists. That is, there is no node which can be removed to make s and t disjoint.

It is clear that $|M(G-p,s,t)| \geq d$, otherwise, $M(G-p,s,t)$ would be the shortest path from s to t in G. But d was defined as this distance.

Let $M(G,s,t) = \{p_1, p_2, \ldots, p_d\}$. Let a_i be the ith element in $M(G-p_i, s, t)$. $a_i \neq p_i$ as $a_i \in G - p_i$. The distance between a_i and s (in $G - p_i$) is i, which implies that the distance between a_i and s (in G) is also i - any element reached through p_i is further away than i from s, so the path from s to a_i is not affected by the inclusion of p_i.

Thus $a_i \neq p_j$ for $j < i$, as the distance between p_j and s is j, not i.

Clearly $a_i \neq p_i$, as $a_i \in G - p_i$. $a_i \neq p_j$ for $j > i$, as this would imply $M(G,s,t) = \{a_1, \ldots, a_i, p_j + 1, \ldots, p_d\}$, a shorter path with size $d - (j - i)$.

Also, $a_j \neq a_k$ for $k \neq j$, because the distance in G from s to a_j is j, but to a_i is i.

Thus all the nodes in $\{a_1, a_2, \ldots, a_d, p_1, p_2, \ldots, p_d\}$ are unique. s is also unique, having the only zero-distance from itself, and t is unique, having the only $d+1$ distance from s in that set. This makes $2d+2$ unique elements, and $n \geq 2d+2$. Then $d+1 > n/2 \geq d+1$, a contradiction. Thus such a graph does not exist, and if $|M(G,s,t)| > n/2$, there is a node p so $M(G-p,s,t)$ does not exist. □

Now consider the following algorithm to find such nodes

1. Do a breadth first search from t, looking for s. Maintain a table of nodes vs which node it was reached from.
2. Each time an edge is followed from n_1 to n_2, if n_2 is not found, change the table entry for n_2 to be n_1, and mark n_2 as found.
3. When s is found, backtrack to t, labelling the nodes in the path with numbers:
   (a) Make a counter variable start at 1
   (b) While not at t
   (c) Label current node as counter variable
   (d) Look up the table entry for current node, go to that node, and increase counter.
4. (for clarification on numbering) s will be 1, next is $2, \ldots, t$ is k (k will be greater than $n/2$)
5. Create an array k bits long of $'0's$
6. Mark all nodes but those in the path as not done.
7. For all nodes
   (a) If marked "done" or a number, go to next node
   (b) Make two variables, min and max
   (c) DFS:

   i. On reaching a numbered node, update min and max (replace min/max with that number if that number is smaller than min/larger than max, replace both if min/max is empty)
   ii. On reaching a non-numbered/non-"done" node, mark that node done and continue DFS

  (d) Change all bits in the array to '1' with index in the interior of min to max.

8. Choose a zero-entry in the array (not node 1 or k), cut the graph at the node with that number.

Correctness: The algorithm always outputs a node that, when cut, will make s and t disjoint.

Proof. If a node's array position is marked '1', then removing that node will not make s and t disjoint because, if a node p was marked 1, there was in the inner loop at some time a min < p and max > p.

This means there was a second path completely outside the known path that went from min to max, and a new path that does not use the node marked 1 can be made like $\{s, p_1, \ldots, p_{min}, \text{insert outside path}, p_{max}, p_{max+1}, ..., t\}$ because there does exist a node that can be cut, all array bits will not be changed to 1, there must exist some 0's at the end.

We need to show any node which can be cut without disrupting a path from s to t will be marked 1.Let the path found in the first part of the algorithm be $P_1 = \{p_1, p_2, p_3, \ldots, p_j\}$ where $p_1 = s$ and $p_j = t$.

Assume that a node p_i on the central path can be cut while still having a second path from s to t. Call a shortest second path $P_2 = \{s, a_1, \ldots, a_k, t\}$. Let $m = \max\{r | r < i \text{ and } p_r \in P_2\}$. Let $M = \min\{r | r > i \text{ and } p_r \in P_2\}$. m exists as $s = p_1 \in P_2$, and M exists as $t = p_j \in P_2$.

The path P_2 travels from p_m to p_M without using any elements of P_1. Thus p_m and p_M are connected outside of the path P_1, and in the iterated DFS, min will be changed to m (or less) and max will be changed to M (or more).

Then p_i will be marked as 1 in the array.

Thus, a node is marked 0 after the algorithm has run if and only if cutting it will make s and t disjoint. □

Runtime: The first DFS is $m + n$.

Keeping track of the table takes at most m steps, as each node entry is changed at most once.

Backtracking from s to t takes as many steps as there are nodes in the path, which is $O(n)$, as $n/2 < k < n$.

We next do a DFS on pieces of the graph, searching for numbered pieces of the path we marked. Each edge is followed at most twice, and each node is checked once, so the actual search is $O(m + n)$.

However, inside the search we change a variable amount of array indices. $max - min - 1$ array spots are changed at this step. For an outside path to reach from p_min to p_max, at least $max - min - 1$ nodes need to be used - if less, then there would be a shorter path, by following $\{s, p_1, ..., p_min, outsidepath, p_max, ..., t\}$, but a bfs was used so there is no shorter path.

Thus for every array position changed, at least one node was marked "done", so changing this array throughout the algorithm takes $O(n)$ time.

Finding a zero in the array is just a simple scan of $k < n$ elements, and is $O(n)$.

Therefore our entire algorithm is $O(m + n)$ as desired.