Solutions for 611 Homework 3

Mahim Agarwal, Hanwen Xiong, and Jackson Warley

October 27, 2017

1 Solution to Question 1

Main Idea: Consider $T = (T_1, T_2, ..., T_n)$ be the array containing time required(T_i) to finish i^{th} problem.

1. Traverse through complete array and determine the sum of all elements in T. i.e. $Sum = \sum_{i=1}^{n} T_i$
2. If sum is not divisible by 3 ($sum\%3! = 0$), return False as we can not divide the array such that 3 students spend same amount of time on solving the problems.
3. else find the subset of the input array which gives us the sum of $sum/3$ using Dynamic Programming, similar to knapsack problem.
4. delete the items that sums up to $sum/3$ from above and re-run the dynamic programming step ($Step3$).
5. If after second iteration $Step4$ returns True, then return true as the final result else return False.

So we are essentially finding two disjoint subsets of the original array which sums up to $sum/3$ where sum is the total sum of all elements in the array.

Pseudo code:

$T = \{T_1, T_2, ..., T_n\}$ is the input array and we create a new dynamic programming array d where $d[i][j]$ is true if a sum of j can be created using first i elements of T, elements in T from T_1 to T_i.
here, i and j range from 0 to n where n is the total number of elements in T.

1. Initialize dynamic programming array, d, such that first row(index 0) is zero and first column(index 0) is all ones.

   $d[0][:] = 0(false)$ and $d[:][0] = 1(true)$ and $sum = 0$

   In other words, we cannot create any sum using 0 elements and thus first row should be 0 while we can create $sum = 0$ using any number of elements and thus first column is 1
2. for ($i = 1$ to n):

   $sum = sum + T_i$

3. if (sum is not divisible by 3):

   return False

4. for ($k = 1$ to 2)

   for ($j = 1$ to $sum/3$):

   for ($i = 1$ to n):

   if ($T_i > j$):

   $d[i][j] = d[i-1][j]$ i.e. We cannot select this(i^{th}) element to create a sum of j and thus whatever is the value(true or false) above, for using $i-1$ elements to create sum j will also be the value of using i elements.

   else:

   $d[i][j] = (d[i-1][j]$ or $d[i-1][j-T_i])$ i.e. either we select this element to make the sum j or we don't select this element.

5. If ($d[n][sum/3] == 0$): return False
   else del_elements(d, T) i.e. Determine all the elements which make up the $sum = sum/3$ (using Algorithm below), delete them and re-run the algorithm($step4$ to $step5$) on the remaining elements.

6. return True

del_elements:

According to the algorithm, for any particular sum we only update the value from 0 to 1 when we have included that particular element to create the sum otherwise we either keep inserting zeros or ones depending on the value above (at $[i-1][j]$).
Thus, to find the elements which constitute the sum, we traverse the last column of d starting from the bottom most ($d[n][sum/3]$) to find the index(k) where 1 occurs for the last time, i.e. above which there are all zeros.
We know that this element is included in creating the required sum. We delete it and move to $d[k-1][sum/3 - T_k]$.
Continue doing this until we reach zero sum.

Algorithm

index is the set of indexes of elements that needs to be deleted.

1. $index \leftarrow \phi$

2. for $j = sum/3$ to 0:

   for $i = n$ to 1:

   if($d[i-1][j] == 0$):

   $index \leftarrow index + i$

   $j = j - T_i$

3. return $T(1....n)$ excluding the indexes in $index$.

Time Complexity

Running time for calculating total $sum = O(N)$
Running time for one iteration to create dynamic array, $d = O(N * sum/3)$
Running time for deleting selected elements of first iteration $= O(N * sum/3)$
So, total running time $= O(N) + 2 * O(N * sum/3) + O(N * sum/3) = O(N * sum)$

2 Solution to Question 2

Our algorithm is as follows.

```
function ALG(G = (V, E))
    run Floyd-Warshall on G
    let D be the resulting table
    for (u, v) ∈ E do
        if w(u, v) > D[u, v] then
            set w(u, v) = D[u, v]
        end if
    end for
    return G with the updated weights
end function
```

Claim 1: ALG terminates in polynomial time.

Proof. Since Floyd-Warshall always terminates, and nothing can cause ALG to consider an edge more than once, ALG will terminate after it has inspected every edge. The single run of Floyd-Warshall requires $\mathcal{O}(|V|^3)$ time, as we showed in class. The remaining work to be done is a single scan through the edges of the graph, with the constant time work of updating $w(e)$ done for each edge. Thus the runtime of ALG is $\mathcal{O}(|V|^3 + |E|) = \mathcal{O}(|V|^3)$ since $|E| = \mathcal{O}(|V|^2)$. □

Claim 2: ALG produces a graph in which every edge satisfies the triangle inequality.

Proof. We call an edge e *bad* in G if it violates the triangle inequality in G. Otherwise, we say e is *good*.

Firstly, any time an edge weight is reduced by ALG, the number of bad edges decreases by one. As proof, suppose that on some iteration of the loop, $e = (u, v)$ has its weight reduced from w_{old} to $D[u, v]$. Since $D[u, v]$ is the shortest path between u and v by definition, e is no longer bad, so G has lost one bad edge. However, it could be that reducing $w(e)$ caused some previously good edge $e' = (u', v')$ to become bad. To see that this is not the case, first note that the only way reducing $w(e)$ could have affected $\delta(u', v')$ is if the shortest path P between u' and v' used e. If this was the case, then P originally incurred the distance w_{old} to travel from u to v (or vice versa). But we reduced $w(e)$ to $D[u, v] < w_{old}$, meaning that there was already a path Q of length $D[u, v]$ between u and v, which would contradict the assumption that P was the shortest path between u' and v' (since P could have taken Q instead of e and been shorter). Thus, decreasing $w(e)$ removed one bad edge from the graph and preserved the number of good edges.

Secondly, note that any time ALG examines an edge but does not reduce its weight, the number of bad edges is unchanged, since no modifications were made to the graph. Since every edge is either good or bad and every edge is either reduced or not reduced by ALG, we may now conclude that the number of bad edges in G never increases, and it decreases by one for each bad edge in G – in other words, there are no bad edges remaining after ALG terminates. □

3 Solution to Question 3

Algorithm:

Create an empty set D and two dictionaries δ_1 and δ_2. Set $\delta_1[t] = 0$
$\#D$ is used to hold "determined" nodes where we can determine it doesn't underestimate $\delta(u, t)$.
$\#\delta_1$ is used to hold $\delta(u, t)$ found at current layer breadth-wise
$\#\delta_2$ is used to compute $\delta(u, t)$ at the next layer using δ_1
for $u \in \delta_1$ **do**
  if $\delta_1[u] < d[u]$ **then**
    return -1
  else if $\delta_1[u] == d[u]$ **then**
    $D.add(u)$
  end if
  for v pointing to u **do**
    if $\delta_2[v]$ exists **then**
      $\delta_2[v] = min\{\delta_1[u] + w(v, u), \delta_2[v]\}$
    else
      $\delta_2[v] = \delta_1[u] + w(v, u)$
    end if
  end for
  if δ_1 is empty **then**
    $\delta_1 = \delta_2$, empty δ_2
  end if
end for
return 0 if $len(D) == |V|$ else -1

Proof of Correctness:

Claim 1: $\delta_1[v] = \delta(v, t)$ at some point $\forall v \in V$.

Proof. Suppose the shortest path of value $\delta(v, t)$ contains n edges. Since we traverse from path containing one edge from t, two edges from t ... all the way up to the furthest edges , say max edges from t where $max \geq n$. Thus, when we are at the nth egdes away from t $\delta_1[v] = \delta(v, t)$ since we always pick the shortest path at each layer.

□

Claim 2: Our algorithm will only return 0 when $d[v] = \delta(v, t) \forall v \in V$.

Proof. Since D will only add node v when $d[v] \geq \delta(v, t)$, we know D doesn't take into account underestimation of $\delta(v, t)$. So the size of $D = |d[v] \geq \delta(v, t) \forall v \in V|$. From Claim 1 we can infer the algorithm will return -1 when it finds $d[v] > \delta(v, t)$. Therefore, D only contains correct nodes after exiting the main loop. The claimed $d[v]$ is correct iff it's correct for all nodes. Therefore, when size of $D = |V|$ we know our friend is correct.

□

Time Complexity: Traversing all nodes and edges in G costs $\mathcal{O}(|E| + |V|)$. Accessing entries in dictionary takes $\mathcal{O}(1)$. Therefore, $T(n) = \mathcal{O}(|E| + |V|)$. But in this case since we can assume $\mathcal{O}(|V|) \leq \mathcal{O}(|E|)$ we can write $T(n) = \mathcal{O}(|E|)$.

4 Solution to Question 4

Algorithm:

Run Ford-Fulkerson Algorithm to find maximum $s-t$ flow F
Get the residual graph G_F of F.
Start from s, use BFS to find all nodes reachable by s and put them into set A, Other nodes into set B.
Delete k or all edges between A and B.

Proof of Correctness:

We know Ford-Fulkerson will correctly find the maximum $s-t$ flow. Because A and B form an $s-t$ cut of G and nodes in B are not reachable from nodes in A in the residual graph of maximum flow, this cut is a min-cut and $f(A,B) = C(A,B)$. Otherwise there must be an edge from A to B in the residual graph which causes contradiction.

Since every edge in G has unit capacity we can at most reduce the maximum flow by 1 for each edge we remove. If we remove edges found in min-cut we are guaranteed to reduce the maximum flow by 1 due to capacity constraints. Therefore, by deleting edges from the min-cut we found in our algorithm will reduce the maximum flow by exactly $min\{k, |F|\}$.

Time Complexity: Ford-Fulkerson Algorithm takes $\mathcal{O}(|E|^2|V|)$. It takes $\mathcal{O}(|E|)$ time to get residual graph. BFS takes $\mathcal{O}(|E|)$ in this case. Removing edges takes $\mathcal{O}(|E|)$. In total $T(n) = \mathcal{O}(|E|^2|V|)$

5 Solution to Question 5

Main Idea: The size of minimum cover of a bipartite graph is equal to the size of its maximum matching.
Pseudo Code:

1. Find maximum matching of the bipartite graph:
   $M \leftarrow \phi$
   while there exists an augmenting path P
   $M \leftarrow M \bigoplus P$
   return M

2. For each of the free vertices(if any) on the L side, find an alternating path of non-matching and matching edges starting from the free vertex until the path terminates.

3. Let K_l = {all the vertices of L that are in this alternating path, including the free vertices} and K_r = {all the vertices of R that are in this alternating path}

4. return $(L - K_l) \cup (R \cap K_r)$

Proof of Concept: Consider following lemmas:

Lemma 5.1. *In the maximum matching graph, any alternating path starting at a free vertex of L must terminate at a matching vertex in L while the alternating path starting at a free vertex of R must terminate at a matching vertex in R.*

Proof. Consider an alternating path starting from a free vertex in L. Suppose, for the sake of contradiction, it terminates at a vertex in R.
Case 1: The R vertex is a free vertex(or non-matching vertex). That means, the alternating path starts and ends at free vertices, so it is an augmenting path which is not possible since we already have the maximum matching. Thus, case 1 is invalid.

Case 2: The R vertex is a matching vertex. If it is a matching vertex then it should have a matching edge going towards L as the alternating path only accounted for the non-matching edge up till now. Thus, the path cannot terminate at R.
Hence, any alternating path starting from free vertex in L must end at a matching vertex in L.
Similar proof holds for the R side. □

Lemma 5.2. *The size of minimum vertex cover of a bipartite graph is equal to the size of its maximum matching.*

Proof. In any matching of the graph, we will need at least one vertex to cover all the matching edges since no two matching edges have same vertex.
So, |*matching*| <= |minimum vertex cover| because each matching edge has at least one vertex which we need to consider for cover.
Now, let |maximum matching| < |minimum vertex cover|.
So, for maximum matching containing e number of edges, we will have e + f number of vertex to have a cover. Here e is one vertex for each of the matching edge while f are extra vertices required to have a cover.
Now, using *Lemma*2.1, these f number of vertices should have an edge to a matching vertex on the

other side(L or R) otherwise they will lead to an augmenting path.
Case 1: If this edge is a matching edge, then we have already accounted its vertex cover in e.
Case 2: If this edge is a non-matching edge, then it will be followed by a matching edge from other side back to its own side. For e.g. if L_1R_3 is a non-matching edge then there should be a R_3L_5 matching edge. According to our definition e should contain either R_3 or L_5. If it contains R_3, then both the edges are already covered. If it contains L_5 we can swap it and instead make R_3 the cover.
Thus $f = 0$ and |maximum matching| < |minimum vertex cover| is not possible.
Hence, |maximum matching| = |minimum vertex cover| □

Consider alternating paths from free vertices of L:
Using *Lemma*5.1 we can say that, all the vertices of R in this path will have one matching edge and one non-matching edge(either to the free vertex in L or to the matching vertex in L). Thus, selecting these vertices as cover will include all the edges from:

1. free vertices of L
2. and edges from those matching vertices of L which are not connected to any free vertex of R

Thus, $R \cap K_r$ will give us these required vertices of R which are in minimum vertex cover.
Similarly, consider alternating paths from free vertices of R:
Using *Lemma*5.1 we can say that, all the vertices of L in this path will have one matching edge and one non-matching edge(either to the free vertex in R or to the matching vertex in R). Thus, selecting these vertices as cover will include all the edges from:

1. free vertices of R
2. and edges from those matching vertices of R which are not connected to any free vertex of L

Thus, $L - K_l$ will give us all these required vertices of L which are not in minimum vertex cover. This will also include those L vertices which only have one matching edge.
Running Time:
Running time to find the maximum matching = $O(|E| \min(|L|, |R|))$
Running time to find the alternating paths = $O(|E||\text{free vertices in L }|)$
Running time return the final vertex cover by union and intersection = $O(|L| + |R|)$
So, total running time = maximum of ($O(|E| \min(|L|, |R|))$, $O(|E||\text{free vertices in L }|)$, $O(|L|+|R|)$)

6 Solution to Question 6

Let $V' = V \cup \{s, t\}$, $E' = E \cup \{(s, u)\}_{u \in L} \cup \{(v, t)\}_{v \in R}$, and $G' = (V', E')$. Assign a direction to each edge in E' such that s is a source, t is a sink, and edges in E are directed from L to R. Finally, give every edge unit capacity for the purpose of defining a flow on G'. We show that the following are equivalent:

1. There exists a perfect matching in G.
2. A maximum flow f on G' has size n.
3. A minimum cut C on G' has capacity n.
4. For all $S \subseteq L$, $|\Gamma(S)| \geq |S|$.

Proof. ($1 \iff 2$) Suppose G has a perfect matching M. Define the flow

$$f(u, v) = \begin{cases} 1 & \text{if } (u, v) \in M \\ 1 & \text{if } u = s \text{ and } v \in L \\ 1 & \text{if } v = t \text{ and } u \in R \\ -1 & \text{if } f(v, u) = 1 \\ 0 & \text{otherwise} \end{cases}$$

on G'. It is immediate from the definition that f is skew-symmetric. Since $c_{uv} = 1$ for all edges $(u, v) \in E'$, $f_{uv} \leq c_{uv}$. For each $u \in L$, u has a single incoming unit of flow, since there is exactly one edge from s to u and no edges from t or R to u by construction, and no edges from L to u because G is bipartite. Since M is perfect, there is exactly one $v \in R$ adjacent to u, and by construction u is emitting one unit of flow along (u, v). A similar argument applies to vertices in R, concluding the proof that f is indeed a flow. Now, we have

$$\begin{aligned} |f| &= \sum_{v \in V} f(s, v) \\ &= \sum_{v \in L} f(s, v) + \sum_{v \in R} f(s, v) + f(s, s) + f(s, t) \\ &= n + 0 + 0 + 0. \end{aligned}$$

Finally, note that f is maximum, since there are only n edges of unit capacity exiting s, which concludes the forward direction.

For the other direction, suppose that some maximum flow f on G' has size n. Since t has indegree n, and each incoming edge has unit capacity, it must be the case that every edge into t carries unit flow, and equivalently, each $v \in R$ emits unit flow. By conservation of flow, each $v \in R$ must receive unit flow, as well. Since the edges out of s have unit capacity, no vertex in L receives more than unit flow. Thus no two vertices in R recieve flow from the same vertex in L. Since our flow is integral, no two vertices in L can supply the same vertex in R with flow. Thus we have constructed a bijection between the vertices in R and those in L, which exactly identifies a perfect matching. □

Proof. (2 $\iff$ 3) This is the max-flow min-cut theorem. □

Proof. (3 $\iff$ 4) For the forward direction, suppose a minimum cut has capacity n. Let $S \subseteq L$. Then $(A, B) = (S \cup \{s\} \cup \Gamma(S), (L \setminus S) \cup (R \setminus \Gamma(S)) \cup \{t\})$ is a cut, so its capacity is at least n. The only edges in the graph are those from s to L, those from L to R, and those from R to t. Therefore we have

$$\begin{aligned} C(A, B) = |\Gamma(S)| + |L \setminus S| &\geq n \\ |\Gamma(S)| + (n - |S|) &\geq n \\ |\Gamma(S)| &\geq |S| \end{aligned}$$

as desired.

For the backward direction, we will use the chain of implications we've already proven, and show that (4) implies (1). Given (4), we can construct a matching by induction on n. If $n = 1$ then we have $|\Gamma(S)| = 1$, so the single existing edge is a perfect matching. Suppose that (4) holds for $n \leq k$. We either have $|\Gamma(S)| = |S|$ for some S, or we have $|\Gamma(S)| > |S|$ for all $S \subseteq L$. In the first case, let S^* be a subset of L with $|\Gamma(S^*)| = |S^*|$. Then there is a unique perfect matching from S^* to $\Gamma(S^*)$ including all the edges between them. Suppose toward a contradiction that there is some set $T \subseteq (L \setminus S^*)$ with fewer than $|T|$ neighbors in $(R \setminus \Gamma(S^*))$. Then

$$\begin{aligned} |\Gamma(S^* \cup T)| &= |\Gamma(S^*)| + |\Gamma(T) \setminus \Gamma(S^*)| \\ &= |S^*| + |\Gamma(T) \setminus \Gamma(S^*)| \\ &< |S^*| + |T| = |S^* \cup T| \end{aligned}$$

since S^* and T are disjoint. This contradicts that (4) held for the original graph, so it must not be the case that such a T exists. Therefore (4) holds in the subgraph with vertices $(L \setminus S^*) \cup (R \setminus \Gamma(S^*))$, and we can find a perfect matching for this subgraph by induction. Since the two matchings we've found are disjoint, their union is a perfect matching for G. We conclude that (4) implies (1), and therefore it implies (3). □

The fact that (1) and (4) are equivalent gives us the desired result.