Johnson Hui

4909153502

Lecture 3:30-5:00

Collaborators: Eddie Lou, Kunal Kataria, Justin Lu

**Problem 1**

**Algorithm**

We want to construct a directed graph from USC players to UCLA players with a source node connecting to all USC players and a sink node that all UCLA players are connected to. Each edge between a USC player and the source node should have a capacity of 3. Similarly, all edges connecting the UCLA players to the sink node will also have a capacity of 3. Finally, we draw an edge with a capacity of 1 connecting a USC player to a UCLA player if the USC player has a higher rating than the UCLA player.

After constructing this graph with the given conditions above, we then run the Ford-Fulkerson’s algorithm to find the max flow of our graph. The edges of the new graph will dictate which UCLA players the USC player will play against with the USC player winning the game. Note that if an arbitrary USC player does not have 3 edges with flow after running Ford-Fulkerson, then we can match this USC player with another arbitrary UCLA player who also does not have 3 edges with flow.

**Proof of Correctness:**

First, we want to show that given the flow network we constructed, every USC and UCLA player will play at most 3 games. Since each USC player has a maximum incoming flow of 3 from the source node and a maximum outgoing flow of 1 for every outgoing edge, all of which are only connected to UCLA players, we can conclude that after running Ford-Fulkerson’s, each USC player will only play at most 3 games. Similarly, since each UCLA player has a maximum incoming flow of one for every incoming edge and a maximum outgoing flow of 3 to the sink node, UCLA players will also have at most 3 games since they only have outgoing edges to the sink node and none towards each other.

Next, we know that running Ford-Fulkerson’s on our flow network returns us an optimal salutation because we only connect an edge between some arbitrary USC and UCLA player if the USC player has a higher rating. In other words, all edges connected between a UCLA and USC players are scenarios where the USC player wins. By running Ford-Fulkerson, we are maximizing the flow of the graph, or in this case, maximizing the number of USC wins. We are also able to pair up the remaining USC players arbitrarily to the remaining UCLA players in the end because if they would have beaten the UCLA player, they would have been connected after running Ford-Fulkerson.

**Problem 2**

**Algorithm**

We first construct a flow network G = (V,E) where V represents the set of n useful programs along with the source node s and sink node t. The source and sink node are connected to all n program nodes in the graph. The edges connecting the source node and some arbitrary node i in the set of n programs will have a capacity of xi. Similarly, the edge connecting some arbitrary node i in the set of n programs will have a capacity of yi. Finally, for every neighboring pair of n programs we add directed edges (i,j) and (j,i), each with a capacity of a(i,j). After constructing this flow network, we run Ford Fulkerson to find the minimum cut in the graph and the remaining edges connected to s and t are the ones the program uses.

**Proof of Correctness**

First, we want to find some constant value P that is compromised of the total value of company one, company two, and the annoyances or P = Σi xi + yi . Given P = Σi xi + yi, we can denote ΣiεS1 xi + ΣjεS2 yj  as P - ΣiεS1 yi + ΣjεS2 xj. Going off this, we can rewrite the original equation as p(S1,S2) = P - ΣiεS1yi - ΣjεS2xj - ΣiεS1 jεS2 a(i,j) where p(S1,S2) is the optimal solution. From the equation above, we can see that maximizing p(S1,S2) is the same as minimizing p’(S1,S2) = ΣiεS1yi + ΣjεS2xj + ΣiεS1 jεS2 a(i,j). In other words, given some constant maximum value P, if we want to optimize p(S1,S2) after removing either company one or two’s programs, we would want to minimizing p’(S1,S2) or the minimum cut of the flow network.

Given that an s-t cut (S1,S2) corresponds to the partition of programs into set x and y, we need to evaluate how any arbitrary cut c(S1,S2) relates to p’(S1,S2). We can group all c(S1,S2) into three categories.

1. Edge (s,j) where jεS2; note that this edge represents capacity xj

2. Edge (i,j) where iεS1 and jεS2; note that this edge represents the annoyance capacity aij

3. Edge (i,t) where iεS1; note that this edge represents capacity yi

If we add these three scenarios together, we get ΣiεS1yi + ΣjεS2xj + ΣiεS1 jεS2 a(i,j), which is exactly p’(S1,S2). Thus, we set up the flow network such that the capacity of all c(S1,S2) is equal to p’(S1,S2). Finally, since we want to minimize p’(S1,S2), which is the equivalent of maximizing p(S1,S2), we just have to find the cut of minimum capacity, which we can do using Ford Fulkerson.

**Problem 3**

a. If we are given a feasible height assignment h for G, then we know that for any two arbitrary nodes (u,v) in G, they satisfy the condition that |h(u) – h(v)|<=le where l is the length of the edge between u and v. Given this condition, if we are given the shortest path from s to t, then any arbitrary two nodes (u,v) along this path should also satisfy the condition |h(u)-h(v)| <= le. If we want to calculate the maximum height from s-t, we can do so by summating the maximum height of every two adjacent nodes along the path. Since every maximum height is less than or equal to the length of the edge connecting them, the total maximum height would be less than or equal to the summation of edge lengths along this path, which would be the shortest path. Therefore, the maximum possible s-t stretch is no more than the shortest path from s to t because if one pair of nodes have a height greater than a length of the edge, we will no longer have a feasible height assignment and the rope will break.

b. If we designate a start node s where h(s) = 0, we can denote the height of any arbitrary vertex v in the path s-t as the shortest path from s to v. Given part a, we are allowed to assign this height value because |h(v)-h(s)| = |h(v)-0| <= shortest path between v and s. In other words |h(v)| = shortest path between v and s is a feasible height assignment that does not break the maximum stretch. By this logic, the stretch |h(t)-h(s)| is equal to the shortest path between s and t because the h(t) = shortest path between s and t and h(s) = 0. Note that this also satisfies the condition that for any two arbitrary nodes u and v that are connected to each other, |h(u)-h(v)| <= le where le represents the edge connecting h(u) and h(v). We know this because if h(u) = shortest path from s to u and h(v) = shortest path from s to v, then |h(u)-h(v)| is just the length of the edge connecting the two. In other words, since the only difference between the shortest path from s to u and the shortest path from s to v is the edge connecting u and v, the equation |h(u)-h(v)| = le is satisfied.

c. Given part b, since we know that the max stretch = shortest path, we can just run Dijsktra’s algorithm once because it outputs the shortest path between two nodes. After finding such a path, given some fixed start node, the shortest path length to every other node from the start node would just be those node’s heights.

**Problem 4**

a.

i. Given an undirected graph G = (V, E) and a nonnegative integer k ≤ |V |, determine whether G has an independent set of size at least k.

ii Given an undirected graph G = (V, E), compute the maximum size of an independent set.

**Algorithm 1**; Given i find ii

Int j = 1

While(j <= |V|)

Run method i on j to see whether G has an independent set of size atleast j

If method i returns true

Increment j by 1

Else

Return j -1

Endwhile

Return |V|

**Algorithm Explanation**

Essentially, we loop through the total number of vertices starting at j = 1 and check whether G has an independent size of at least that current number.  If method i returns true, we increment j by 1 and continue the loop again until method(i) returns false. If we ever get an instance where method i returns false, we know that for that number k, G does not have an independent size of at least k size. Therefore, we can conclude that the maximum size of an independent set must be k-1 because if it wasnt, method(i) would have returned false for k-1 and we would have never iterated to that number k. If method(i) never returns true and we finish the while loop, we know that the maximum size of an independent set in G must be |V| since we have determined that G has an independent set of size atleast j=|V| and it is impossible to have an independent set of size |V|+1.

**Runtime Analysis**

The worst case would be for you to complete the while loop and iterate until you reached |V|. Therefore, given that method i has some runtime O(x), the total runtime would be O(|V| \* x) since every operation other than method i has a runtime of O(1).

**Algorithm 2 Given method ii find problem i**

Run method (ii) to find the maximum independent set of size m

Compare your value k to m, which you just calculated

If k <= m

Return true

Else

Return false

**Algorithm 2 Explanation**

Given method ii, if we want to find out if G has an independent set of size atleast k, we run method ii to first find the maximum independent set of size m. If k is less than or equal to m, we return true and false if otherwise. We can conclude this because if m is the maximum independent size set in G and k > m, then we know that G does not have an independent set of size atleast k because m is the absolute maximum.

**Runtime Analysis**

Since every operation besides method ii runs in O(1), the total runtime of algorithm 2 would be O(y) given that method(ii) runs in O(y).

**Explanation of polynomial-time equivalence**

Given the runtimes of algorithm 1 and algorithm 2, we know that they are of polynomial-time equivalence because both runtimes depend on method(i) and method(ii)’s runtimes. In algorithm i, if O(x) has a polynomial runtime, then running method i |V| times where |V| is the number of vertices in the graph, making the total runtime |V|\*O(x), which is still polynomial. Similarly, for algorithm 2, the total runtime is O(y), which is the total runtime of method(ii). Since we are able to solve both scenarios given that we need the other and that the total runtime is just the runtime of the other method, method(i) and (ii) are polynomial-time equivalence.

b.

ii Given an undirected graph G = (V, E), compute the maximum size of an independent set.

iii Given an undirected graph G = (V, E), compute an independent set of maximum size.

**Algorithm 1 Given ii, find iii**

Set allNodes //contains the independent set of maximum size

Int orig = Run method ii on G

Iterate from n = 1 to |V|

Remove n from the graph G, (let’s call the resulting graph called G’)

Int m = Run method ii on G’

If m < orig

allNodes.add(m)

Add n back to the G’ to create G

Endfor

Return allNodes

**Algorithm 1 Explanation:**

Essentially, you first run method ii on G to find the maximum size of the independent set. From there, you iterate through all nodes in G and run method ii on the graph G’ = (V’,E’) where G’ is a subgraph of G where the current node we are examining is removed. If the maximum size of the independent set of G’ is less than G, that must mean that the node we removed was a node in the maximum size of the independent set in G. We know this because removing that node causes the maximum size to decrease by 1. Similarly, by the same logic, if we remove a node and the maximum size of G’ is the same as G, we know that the node we removed is not part of the maximum size of the independent set in G. After evaluating a node, we plug it back into the graph and move on to evaluate the next node.

**Runtime Analysis**

Given that method ii runs in O(x), the total runtime is O(x) + O(|V|\*x). We get O(x) from the first call to method ii and we get O(|V|\*x) because we perform O(x) operations for each node in G.

**Algorithm 2 Given iii, find ii**

Set allNodes = run method iii on G

Return size of allNodes

**Algorithm 2 Explanation**

Given method iii that computes an independent set of maximum size, all we need to do is find the cardinality of the set returned by method iii to compute the maximum size of an independent set.

**Runtime Analysis**

Given that method iii runs in O(y), the total runtime would be O(y) since every other operation in the algorithm takes O(1).

**Explanation of polynomial-time equivalence**

Given the runtimes of algorithm 1 and 2, we know that problem ii and iii are polynomial-time equivalent because they both depend on the runtime of each other to be solved. For algorithm 1, since we call method ii |V|+1 times, if method ii had a runtime of polynomial or higher time, then the total runtime of algorithm 1 would be equivalent to the runtime of method ii. Similarly, our runtime for algorithm 2 is also dependent on the runtime of method iii. Therefore, since both algorithm 1 and 2, which solve problems iii and ii respectively, depend on the runtimes of method ii and iii respectively, problem ii and iii are polynomial-time equivalent.