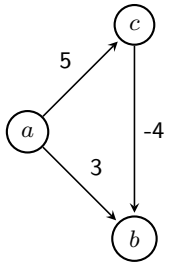
Collaborators

Ben Nelson, Corbin Baldwin and I collaborated for this assignment.

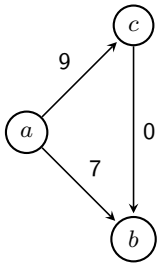
Question 1: Negative edges ..... **[4]**

We saw that Dijkstra's algorithm (as such) requires all the edge weights to be non-negative in order to work. A student suggests a simple fix. Given a graph G with negative weight edges (but no negative cycles), he suggests computing the least weight $w_{\min}$, and then adding $|w_{\min}|$ to all the edge weights. This would make all the weights non-negative. He claims that finding the shortest path on this new graph yields the shortest path in G as well.

Is there something wrong in the reasoning above? Explain with an example (of some fixed size).



Consider the case shown above where we need to find the shortest path from node a to node b. The edge lengths are shown next to the edges. It can be seen that the shortest path from a to node b is through c, with a length of $5 + (-4) = 1$. The 1-hop path from a to b is longer with a length of 3. If we add $|w_{\min}| = |-4| = 4$ to each each, we get the new graph shown below.



The shortest path now becomes the one directly from a to b. The reason why this fix does not work is that $|w_{\min}|$ is added to each edge length and this may cause the shortest path in the original graph to no longer be the shortest one in the new graph with the changed edge weights, when the shortest path consists of multiple hops and the weight of each hop goes up by $|w_{\min}|$.

Question 2: Fattest path ..... **[6]**

Let G be a directed graph in which every edge e has a *thickness* t_e. Given u and v, find the path from u to v that maximizes the least-thick-edge on the path. [i.e., we want a path in which *every* edge is as thick as possible. Note that the length of the path does not matter.]

You will get partial credit if your algorithm runs in polynomial(m, n) time (m and n are the number of edges and vertices, as usual). To receive full credit, it should run in $O((m + n) \log n)$ time. [*Hint:* You might want to modify Dijkstra's algorithm and use its run time analysis as a blackbox.]

Algorithm 1 Modified Dijkstra's Algorithm

```
1: procedure ModifiedDijkstra'sAlgorithm(graph G{V, E}, thickness_function w, start_node s)
2:     InitializeGraph(G,s)
3:     visited_nodes_set = ∅
4:     max_priority_q = G.V                          ▷ based on vertex least thick edge
5:     while max_priority_q ≠ ∅ do
6:         a = Extract_Max(max_priority_q)
7:         visited_nodes_set = visited_nodes_set ∪ {a}
8:         for each vertex b ∈ {neighbors of a} do
9:             Relax(a, b, w)
10:        end for
11:    end while
12: end procedure
13: procedure InitializeGraph(graph G{V, E}, start_node s)
14:    for each vertext a ∈ G.V do
15:        a.least_thick_edge = 0
16:        a.predecessor = NIL
17:    end for
18:    s.least_thick_edge = ∞
19: end procedure
20: procedure Relax(current_node a, unvisited_neighbor b, thickness_function w)
21:    if b.least_thick_edge < min(w(a, b), a.least_thick_edge) then
22:        b.least_thick_edge = min(w(a, b), a.least_thick_edge)
23:        b.predecessor = a
24:    end if
25: end procedure
```

The algorithm shown above is a slightly modified version of Dijkstra's Algorithm [1]. All the edges are first initialized with 0 as the value of the least thick edge on the path that they are on. The starting node is initialized with ∞ for the least thick edge. The set of visited nodes is initially empty. A max priority queue based on the least thick edge for each vertex is used. The algorithm then repeatedly removes the max node from the priority queue which will contain the frontier of unvisited nodes. The extracted node will be the one that lies on the path with the maximum least thickest edge. The running time of this algorithm will be the same [1] as the one for Dijkstra's Algorithm $O((n+m)\log n)$.

To prove correctness of the algorithm described above, we can prove that for each neighboring node of a node added to the set of visited nodes and removed from the max priority queue, the *least_thick_edge* for the path to it and the predecessor will have the correct values. Consider the base case of when the *least_thick_edge* attribute of the neighbors of the starting node s is being updated. Each of these neighbors will get the thickness of the edge from node s to it. And that will be the correct value of the least thick edge on the path to it. The predecessor will be node s and this is correct. Now assume that at a particular step, the node removed from *max_priority_q* will update the correct value of the attribute *least_thick_edge* for the path to them and for the predecessor for all its neighbors. The next node removed from *max_priority_q* will be the one with the maximum value of the attribute *least_thick_edge*. Every neighbor of this node will be checked to see if its *least_thick_edge* attribute has a value less than the minimum of thickness of the edge to it and attribute *least_thick_edge* of the previous node. If it is less, then it will be replaced by the minimum of thickness of the edge to it and attribute *least_thick_edge* of the previous node and its predecessor will be the node that was removed from the max priority queue. So the path to the neighbor node changes to the one through the last removed node as this is the one with maximum least thickness. If it is not less, then the best path to the neighbor node will not be through the last removed node and whatever path it already has, has a larger least thick node. So in

both the cases the values of *least_thick_edge* and predecessor will be correct. This proves the correctness of the algorithm.

After the algorithm ends, each node, including v, will have the value of the *least_thick_edge* of the path to it and its predecessor for the path from u. So the path to any node from u can be traced back using the predecessor attribute. The use of the max priority queue maximizes the least-thick-edge on the path to each node.

Question 3: Answering distance queries ..... **[12]**

Let $G = (V, E)$ be an *undirected* graph with all edge weights equal to 1. Let $d(i, j)$ denote the length of the shortest path between i and j. Now, suppose we wish to answer queries from the user of the kind "what is $d(u, v)$"? for different u, v.

One option is to compute answers to each query as it comes. This takes $O(m+n)$ time using BFS, as the graph is unweighted. Another option is to solve the so-called All-Pairs-Shortest-Path (APSP) problem and store all the answers. This returns the answer in $O(1)$ time, but uses $O(n^2)$ additional memory – which can be cumbersome for large graphs (think $n = 10^8$ – common in real networks). The goal of this problem is to see if there is middle ground, if we allow *an approximation*. The proposed algorithm does the following:

(pre-processing): choose a random subset S of vertices (the size is specified later). For each $s \in S$, do a BFS and store the values $d(s, u)$ for all $u \in V$.

(query): at query time, given u, v, return $\min_{s \in S}\{d(u, s) + d(v, s)\}$.

(a) **[2]** Prove that for any choice of S, the value we output for a query is $\geq d(u, v)$.

(b) **[3]** Suppose we obtain S by randomly including every vertex of U with probability r/n. (Thus the expected size of S is r.) What are the expected pre-processing time and the memory usage of the algorithm?

(c) **[7]** Suppose that $d(u, v) > 5n/r$ for some pair of vertices u, v. Prove that with probability > 0.99, we obtain the right answer to the distance query. [*Hint:* consider the shortest path from u to v and the vertices on it.]

The moral is that if G is sparse, then by picking say $r = \sqrt{n}$, we can do much better than APSP, and get right distance values for all the "long paths" with high probability.

(a) For any u and v, the shortest path between them may or may not include a vertex that is in S. In the case that the shortest path between them includes a vertex in S, then $\min_{s \in S}\{d(u, s) + d(v, s)\}$ will be equal to $d(u, v)$ for the case when the $s \in S$ that is in the shortest path between u and v is considered. The reason is that subpaths of shortest paths are shortest paths (by Lemma 24.1 [1]). Consider the case when the shortest path between u and v contains a node $s_i \in S$. $d(u, s_i) + d(v, s_i)$ will be equal to $d(u, v)$ by the above Lemma and $\min_{s \in S}\{d(u, s) + d(v, s)\}$ will be equal to $d(u, v)$. For the case where the shortest path between u and v does not include a vertex in S, a path between u and v through any $s \in S$ will be longer than the shortest path between them. So we can conclude that the query will always output a value $\geq d(u, v)$.

(b) Assuming that it takes constant time to decide whether to pick a vertex for the set S or not, it will take $O(n)$ to look at every vertex and decide whether to include it or not. For every vertex $s \in S$, we would need to do a BFS in $O(m + n)$ time and then compute and store $d(s, u)$ for every $u \in V$. This will take a total of $O(n + r \times (m + n)) = O(r \times (m + n))$ time and will need $O(n \times |S|) = O(n \times r)$ storage to store the distance to each of the n vertices from every vertex $s \in S$.

(c)

$$d(u,v) > 5\frac{n}{r}$$

$\Rightarrow$ Number of nodes in shortest path between u and $v > 5\frac{n}{r}$, since each edge is of length 1

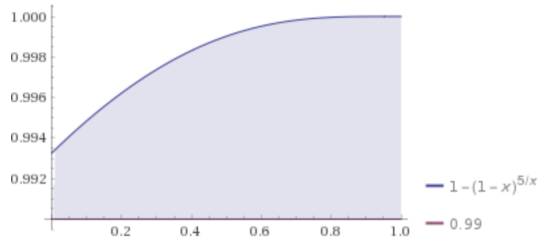
$$\Pr[\text{a specific node in shortest path} \in S] = \frac{r}{n}$$

$$\Pr[\text{a specific node in shortest path} \notin S] = 1 - \frac{r}{n}$$

$$\Pr[\text{no node in shortest path} \in S] < \left(1 - \frac{r}{n}\right)^{5\frac{n}{r}}$$

$$\Pr[\text{some node in shortest path} \in S] > 1 - \left(1 - \frac{r}{n}\right)^{5\frac{n}{r}}$$

Shown below is a plot for $1-(1-x)^{\frac{5}{x}} > 0.99$ that was done using https://www.wolframalpha.com.



We can see that for all values of $\frac{r}{n}$ other than 0, the value of $1 - \left(1 - \frac{r}{n}\right)^{5\frac{n}{r}}$ is more than 0.99. This means that $\Pr[\text{some node in shortest path} \in S] > 0.99$. Since we have earlier shown that when the shortest path includes a node in S, the query returns the shortest path, we can conclude that when $d(u,v) > 5\frac{n}{r}$ we obtain the right answer to the distance query with probability > 0.99. This can also be shown to be true with more rigor as follows:

$$\begin{aligned}
x, y \in [0,1] : x &< y \\
-x &> -y \\
1 - x &> 1 - y \\
\ln(1-x) &> \ln(1-y) \\
\frac{5}{x}\ln(1-x) &> \frac{5}{y}\ln(1-y), \text{ since } \frac{5}{x} > \frac{5}{y} \\
\ln(1-x)^{\frac{5}{x}} &> \ln(1-y)^{\frac{5}{y}} \\
(1-x)^{\frac{5}{x}} &> (1-y)^{\frac{5}{y}} \\
-(1-x)^{\frac{5}{x}} &< -(1-y)^{\frac{5}{y}} \\
1-(1-x)^{\frac{5}{x}} &< 1-(1-y)^{\frac{5}{y}}
\end{aligned}$$

This means that $1 - \left(1 - \frac{r}{n}\right)^{5\frac{n}{r}}$ is a strictly increasing function. To find the value of this term at values of of $\frac{r}{n}$ close to 0:

$$
\begin{aligned}
\text{Let } \ell &= (1-x)^{\frac{5}{x}} \\
\ln \ell &= \ln\left((1-x)^{\frac{5}{x}}\right) \\
&= \frac{5}{x}\ln(1-x) \\
\lim_{x\to 0} \ln \ell &= \lim_{x\to 0} \frac{5}{x}\ln(1-x) \\
&= \lim_{x\to 0} \frac{5\frac{-1}{1-x}}{1}, \text{ by L'Hopital's rule} \\
&= -5 \\
\Rightarrow \lim_{x\to 0} \ell &= \frac{1}{e^5} \\
\Rightarrow \lim_{\frac{r}{n}\to 0} 1-\left(1-\frac{r}{n}\right)^{5\frac{n}{r}} &= 1-\frac{1}{e^5} \\
&= 0.9933 \\
&> 0.99
\end{aligned}
$$

Since $\lim_{\frac{r}{n}\to 0} 1-\left(1-\frac{r}{n}\right)^{5\frac{n}{r}} > 0.99$ and $1-\left(1-\frac{r}{n}\right)^{5\frac{n}{r}}$ is a strictly increasing function, we can conclude that when $d(u,v) > 5\frac{n}{r}$ we obtain the right answer to the distance query with probability > 0.99.

Question 4: Flows and cuts – basics ..... **[10]**

(a) **[5]** As we mentioned in class, the image segmentation problem can be modeled as the following graph question: let G be a weighted undirected graph (weights non-negative), and let S and T be two subsets of the vertices. Find the smallest cut in G that separates S from T. In other words, find a subset of the edges with minimum total weight, such that after removing these edges, there is no path left from any $s \in S$ to $t \in T$.

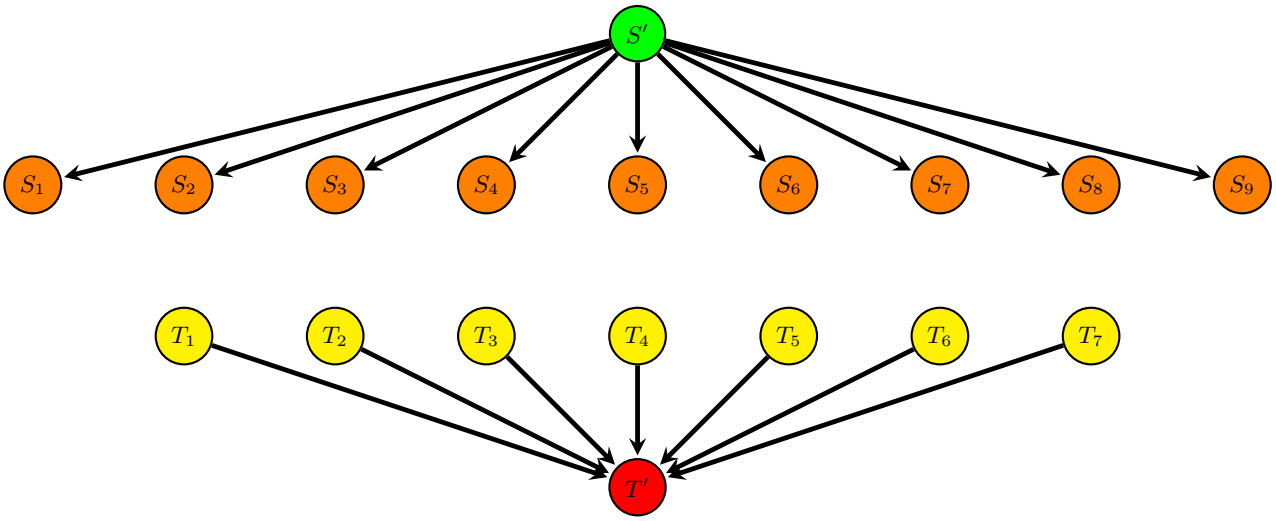
(Note that in the standard formulation of cuts, S and T are singletons.) [*Hint:* find a way to use the min cut algorithm we saw in class in a blackbox manner.]

(b) **[5] Vertex disjoint paths.** Let G be an unweighted directed graph. We saw how to construct multiple *edge*-disjoint paths from two given vertices u and v by simply viewing the graph as a flow network with every edge having a unit capacity, and finding the max flow from u to v.

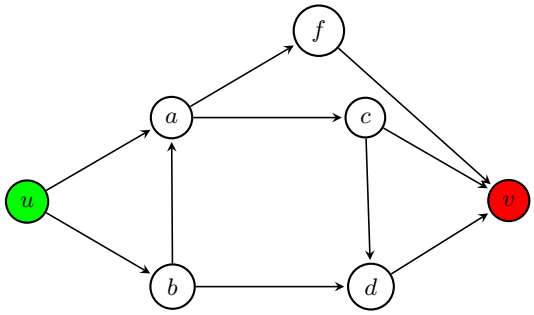
Now, suppose we wish to find the maximum number of paths possible from u to v that do not share any *vertices*. Show how to cast this as a max-flow problem (of size polynomial in the size of G).

(a) We need to make some modifications to the graph in order to solve this. For each weighted undirected edge in the graph, replace it with two directed edges with the same weight going between the two edges that the undirected edge used to connect. In the bipartite graph below, the nodes in subsets S and T are represented by $\{S_1, \ldots, S_9\}$ and $\{T_1, \ldots, T_7\}$. The number of nodes is just an example and there is no limitation on the number of nodes. The edges in the original graph G have not been shown for simplicity. Create a source node S' and a sink node T' and connect them to all nodes in S and T respectively as shown below, with edges of infinite capacity. Find the maximum flow from S' to T' and find the min cut using the last residual graph. The min cut will be on the paths between subsets S and T and the required subset of the edges with minimum total weight, such that after removing these edges, there is no path left from any $s \in S$ to $t \in T$, will be the edges in G corresponding to the directional edges that were removed from the modified graph.

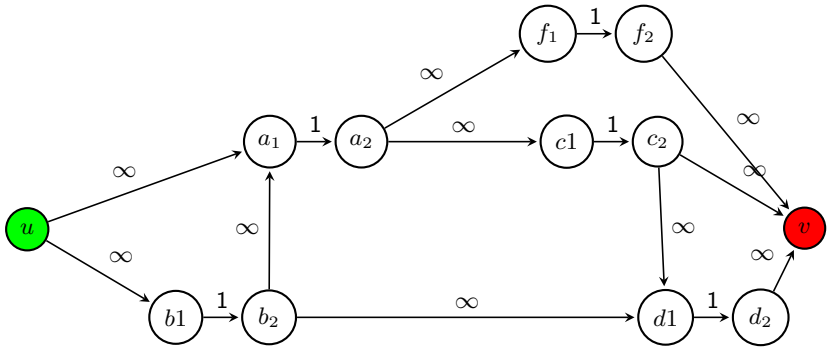
None of the edges between nodes within S or within T will be cut as they will never be part of a min cut. The min cut will always be on the paths between S and T.



(b) Consider an example of an unweighted directed graph shown below. We need to find the maximum number of paths possible from u to v that do not share any *vertices*.



This can be solved for the graph shown above and any unweighted directed graph by replacing every vertex other than u and v by a pair of vertices connected by a directed edge in a modified graph. The modified graph will have $n-2$ extra edges and so this new problem is clearly a polynomial in G as it now has n vertices and $m+n-2$ edges. Every edge leading into a node in the original graph, will be replaced by the same edge leading into the first node in the new node pair with the directed edge leaving the new node. Every edge leaving a node in the original graph, will be replaced by the same edge leaving the second node in the new node pair with the directed edge entering the new node. All edge weights or capacity between the new nodes will be made 1 and the capacity between the other nodes will be made infinite. This construction is shown below:



Now if we find the max flow between u and v, the flow through the original nodes, which have now been split into two nodes, will be restricted to one path due to the edge capacity of 1. The max flow will end up finding the maximum number of paths possible from u to v that do not share

any vertices. The maximum number of paths will be the result of the infinite capacity of the edges corresponding to the ones in the original graph, since the algorithm will keep trying to push more flow and will only be limited the edges having capacity 1. Since each vertex of the original graph will only be used once for a single path, the paths will not share any vertices.

Question 5: Matchings ..... **[7]**

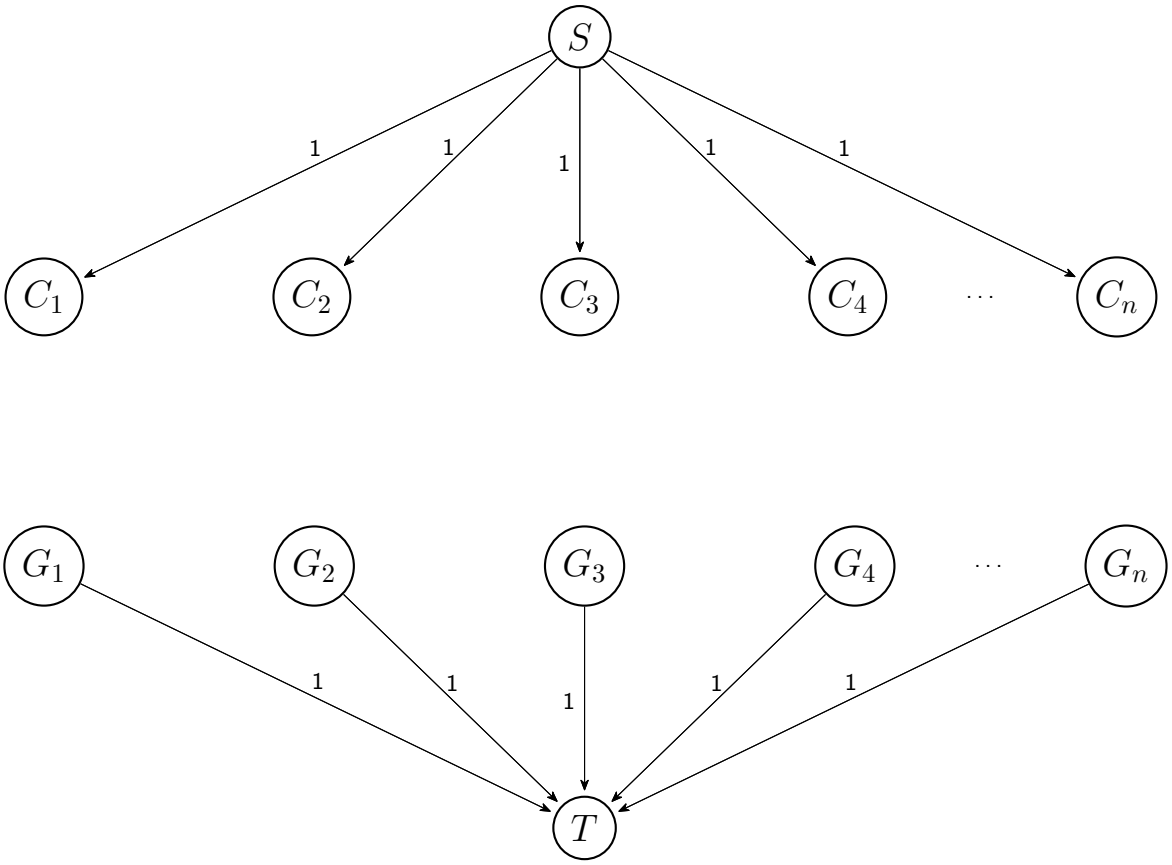
Consider the matching problem that we have encountered before, but with *binary* weights. i.e., suppose we have n children and n gifts, and every child has a 0/1 happiness value associated with each gift. The goal is to assign the gifts to the children, so as to maximize the total happiness. The question we wish to understand is: is there an assignment in which the total happiness is n? (I.e., can every child be assigned a gift to which he/she has a happiness value of 1?)

(a) **[2]** Let $\Gamma(i)$ denote the set of gifts for which child i has a happiness value equal to 1. One trivial case in which the total happiness cannot be made n is if there is a set R of children such that $\cup_{r \in R} \Gamma(r)$ has size $< |R|$. Give a short reason why this is so.

(b) **[5]** Let us call a set R as above a *trivial obstruction* (to the presence of an assignment of total happiness n). Prove that whenever the optimum total happiness is $< n$, such a trivial obstruction must exist. [*Hint:* use the max-flow min-cut theorem!]

(a) Since $| \cup_{r \in R} \Gamma(r)| < |R|$, whatever kind of matching is done for the children in set R there will always be some child who will not get a gift. The reason is that the size of the set of the union of gifts that each child in set R likes is less than the number of children in the set R. So the total happiness cannot be made n.

(b) The matching of gifts to children can be solved by constructing a bipartite graph with nodes representing children on one side and nodes representing gifts on the other. For every gift that a child likes, construct a directed edge of capacity 1 from the child to the gift. Put a node representing the source node on the side of the bipartite graph with the nodes representing the children. Connect the source node using directed edges of capacity 1 to each child with the edge pointing away from the source. Construct a sink node on the other side of the graph. Connect each gift node to the sink node using directed edges of weight 1 going into the sink. This construction is shown below:



The source node is represented by S, the child nodes are represented by $\{C_1, \ldots, C_n\}$ and the gift nodes by $\{G_1, \ldots, G_n\}$ and the sink node by T. The capacities of the edges going from the source node and into the sink node is shown by the number 1 next to the directed edges. The edges between the child and gift nodes have not been shown.

We can now find the max flow from the source S to the sink T. Since the edge capacities into the child nodes and out of the gift nodes are each 1, the maximum flow possible will be n. The flow from the child to the gift nodes will correspond to one of the optimum matchings of children to gifts for maximum happiness. In case the maximum happiness found through maximum flow is less than n, it means that at least one child node and at least one gift node do not have any flow through them. Now consider the case when $R = \{C_1, \ldots, C_n\}$. In this case $|\cup_{r \in R} \Gamma(r)| < |R|$. If this were not the case there would have been perfect matching and maximum happiness would have been n. If $|\cup_{r \in R} \Gamma(r)|$ had been equal to $|R|$ ($= n$ in this case), the maximum happiness would have been n since it would have been possible to find a gift for each child through max flow. So the trivial obstruction was found for the case when $R = \{C_1, \ldots, C_n\}$.

Question 6: Matching games ..... **[11]**

Alice and Bob play the following game: Alice starts by naming an actress A1. Bob must then name an actor B1 who has appeared in a movie with A1. Then Alice must name an actress A2 who appeared with B1, and so on. (Alice must always pick from the set of actresses, and Bob must pick from the set of actors.) The catch is that the players are not allowed to name anyone they have named already. The game ends and a player loses if he/she cannot name an actor/actress who hasn't been named already.

Suppose we are given as input a set of all "allowed" movies and their casts, and suppose that the total number of actresses is equal to the total number of actors. We can construct a bipartite graph between actresses and actors, in which there is an edge iff the two have appeared together in a movie. Let us call this graph G.

(a) [**4**] Prove that if G has a perfect matching, then there is a winning strategy for Bob. (I.e., no matter how Alice plays, Bob can win.)

(b) [**7**] If G does *not* have a perfect matching, prove that Alice has a winning strategy. [*Hint:* consider small examples; start with a maximum matching, and think of how Alice might want to start the play.]

(a) Consider the case when there is only 1 actor and 1 actress. Since there is perfect matching, after Alice selects the 1 available actress, Bob can choose the actor paired with that actress in the perfect matching. Alice will not have any more actresses to choose and thus Bob will win. This is the inductive base case. Assume that when there are n pairs of actresses and actors, Bob wins. In the case when there are $n+1$ pairs, Alice has the option of choosing the actress who has appeared with the actor that Bob last chose (from the n pairs), if there is one available. Since there is perfect matching, when Alice chooses that actress, Bob is assured that there will be an actor who has appeared with the actress last chosen by Alice. Since there are no more actresses to choose from, as all of the $n+1$ pairs have been used up, Bob will win. On the other hand, if Alice does not have an actress who has appeared with the actor that Bob last chose, Bob wins.

Based on the above inductive proof, we can say that Bob always wins and his strategy will be to try an choose an actor such that Alice does not have an actress to choose. If that does not work, Bob will always win when they run out of actors and actresses to choose from.

(b) Since G does not have perfect matching, there cannot be a chain of actors and actresses who appeared together on the bipartite graph, including all nodes that starts on Alice's side and ends on Bob's side, since this would mean perfect matching. If such a chain had existed, then Alice would not be able to win as the chain would always end at Bob's end and Alice would not then have an actress to choose. Since such a chain cannot exist, there are the following remaining possibilities:

1. There exists an actress who has not appeared with any actor. If that's the case, then Alice can choose that actress and Bob will not have anybody to choose. Alice will then win.
2. If the above case does not exist, then there could be a chain that ends on Alice's side and she could choose that and win.

There cannot be a chain that ends at Bob's side with Alice not having any of the opportunities listed above in a case when perfect matching does not exist. So Alice always has a winning strategy.

References

[1] "24. Single-Source Shortest Paths." *Introduction to Algorithms*, by Thomas H. Cormen et al., MIT Press, 2009, pp. 648-662.