CS 124 Problem Set 6
HARVARD ID: 10939860

1. **Solution.**

To find the expected number of moves, we will find a recurrence. Letting E_i be the expected number of moves to reach n starting from position i, we can write out the first few values of the recurrences to find a pattern. We simply use the problem's instructions for the random walk:

$$E_0 = \frac{1}{2}E_0 + \frac{1}{2}E_1 + 1$$

$$E_1 = \frac{1}{2}E_0 + \frac{1}{2}E_2 + 1$$

$$E_2 = \frac{1}{2}E_1 + \frac{1}{2}E_3 + 1$$

$$E_3 = \frac{1}{2}E_2 + \frac{1}{2}E_4 + 1$$

which we can generalize to

$$E_i = \frac{1}{2}(E_{i-1} + E_{i+1}) + 1$$

We can solve for E_1 in terms of E_0 from the first equation and then plug in to the second equation to solve for E_2:

$$\frac{1}{2}E_0 - 1 = \frac{1}{2}E_1$$

$$E_1 = 2(\frac{1}{2}E_0 - 1) = E_0 - 2$$

$$E_0 - 2 = \frac{1}{2}E_0 + \frac{1}{2}E_2 + 1$$

$$\frac{1}{2}E_0 - 3 = \frac{1}{2}E_2$$

$$E_2 = E_0 - 6$$

Using the same logic for the following equations by plugging in each equation to the next, we get that

$$E_3 = E_0 - 12$$

$$E_4 = E_0 - 20$$

By pattern recognition, we notice that

$$E_i = E_0 - (i^2 + i)$$

Since we know that $E_n = 0$, we plug in n to this recurrence:

$$0 = E_0 - (n^2 + n)$$

$$E_0 = n^2 + n$$

So,

$$E_i = (n^2 + n) - (i^2 + i) = (n^2 + n) - i(i + 1)$$

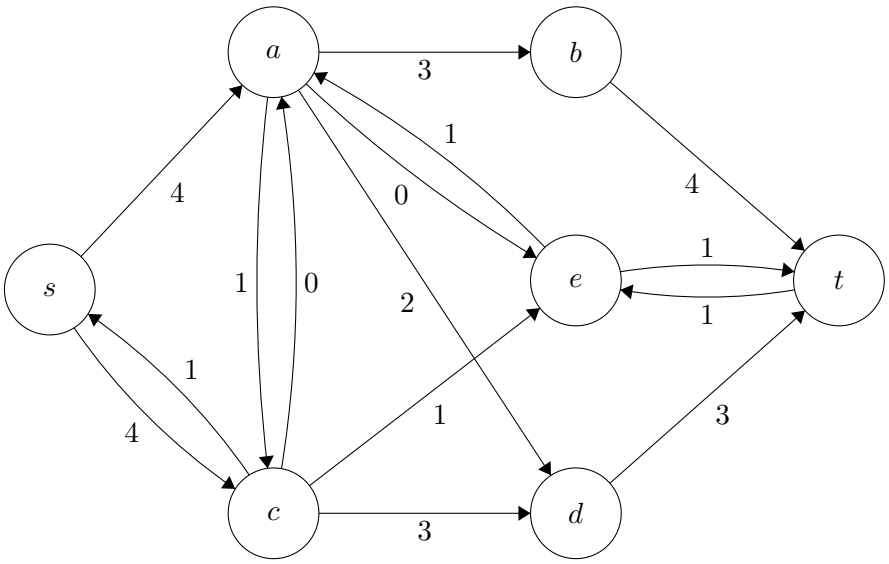
We will use the second form of the solution to prove that our solution to the recurrence is correct. Using this as our base case, we will prove correctness by induction, showing that E_{i+1} is equal to the next iteration of $(n^2 + n) - i(i + 1)$:

$$\begin{aligned} E_{i+1} &= (n^2 + n) - (i + 1)^2 - (i + 1) \\ &= n^2 + n - i^2 - 3i - 2 \\ &= n^2 + n - (i + 2)(i + 1) \end{aligned}$$

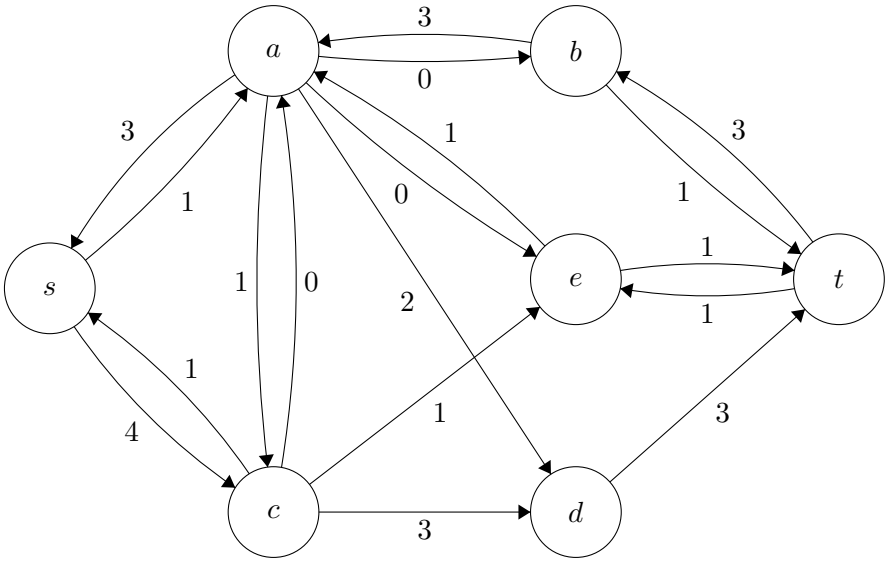
Our solution shows that $E_{i+1} = n^2 + n - (i + 2)(i + 1) = n^2 + n - ((i + 1) + 2)(i + 1)$, and thus the recurrence is correct for all $n, i \geq 0$.

2. **Solution.**

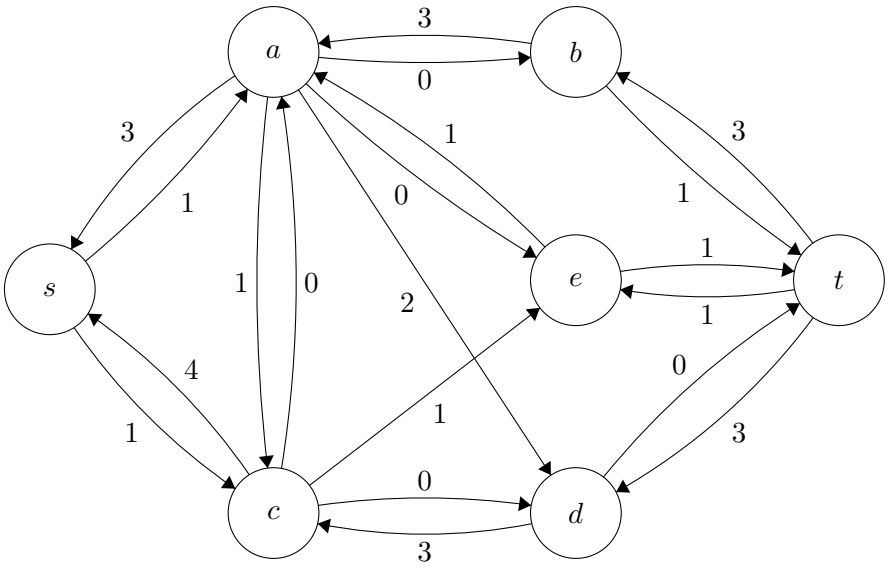
To find the minimum flow between s and t, we will run the Ford-Fulkerson algorithm, drawing the residual graph at each step. The max flow in step 1 with path $s- > c- > a- > e- > t$ is 1:



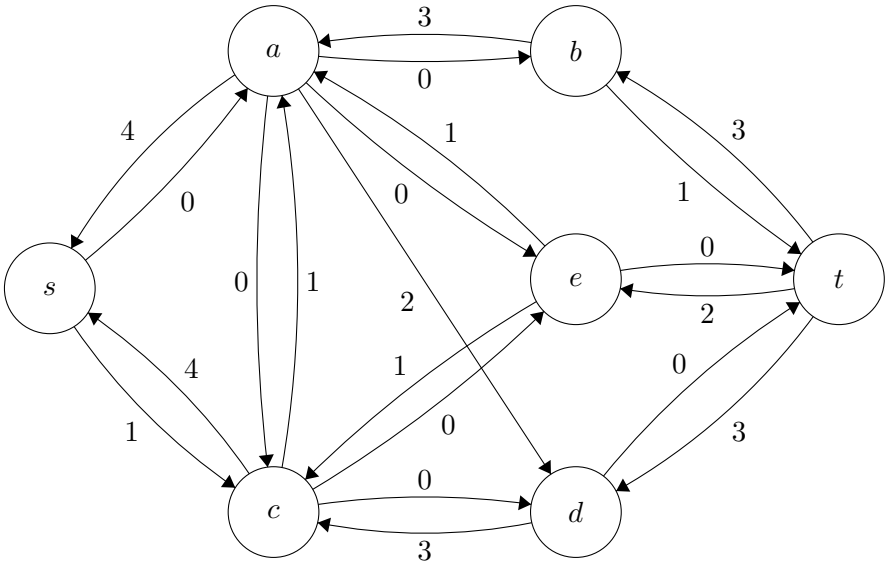
The max flow at step 2 with path $s- > a- > b- > t$ is 3:



The max flow in step 3 with path $s- > c- > d- > t$ is 3:



And finally, the max flow at step 4 with step $s- > a- > c- > e- > t$, the final step in the algorithm, is 1:

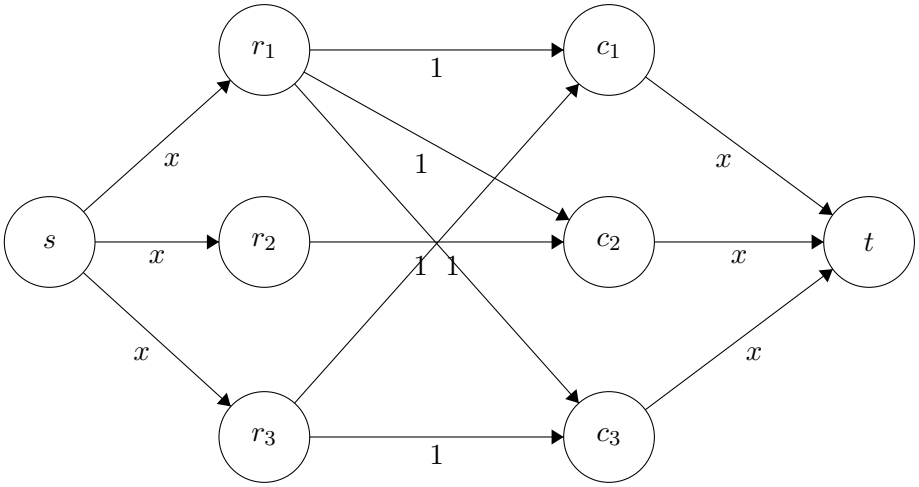


Now, our algorithm is finished because a path can no longer be found from s to t with sufficient flow. Thus, adding the max flows in each of the residual steps, we find that the maximum flow from s to t is 8.

To find the minimum cut, we find the vertices that are reachable from start node s, so we have our cut to be $S = \{s, c, a, d\}$, with $V - S = \{b, e, t\}$. We can confirm that this is the minimum cut, since the capacities going into the cut is equal to that of the maximum flow, which makes sense, since all flow must cross the cut at some point. Adding the flow from the given graph of edges that cross into the cut from S into $V - S$, we have $c_{ab} + c_{ae} + c_{ce} + c_{dt} = 3 + 3 + 1 + 1 = 8$, which we have found to be the maximum flow, so the given cut is indeed the minimum cut.

3. **Solution.**

We can set this up as a flow problem, with a similar setup to the hospital and patient problem discussed in section. Consider the graph shown below. There are $2n + 2$ nodes in total, with s, a source node, t, a sink node, $r_1, ..., r_n$ representing the i^{th} row, and $c_1, ..., c_n$ representing the j^{th} column. This is a rudimentary example to show a representation of a plot with 3 rows and 3 columns:



In this representation, the edge from r_i to c_j gives the subplot of the $r_i c_j$ location on the plot of land. The 1 value attached to the edge can be thought of a boolean value. If there exists an edge with capacity 1 in subplot $r_i c_j$ (because each subplot can only hold one tree), then that subplot has soil and can support plant growth. If subplot $r_i c_j$ has rocky ground and cannot support the growth of a tree, then there is an edge between r_i and c_j with capacity 0, though for simplicity those edges have been left out of the above graph, since they are irrelevant.

The value of x in the graph will ultimately be p, which we are trying to maximize. With this setup, we treat x as a counter that will give us the maximum value for p that maintains the restrictions given by the problem. To do this, set x as 1, run the Ford-Fulkerson algorithm to completion, and check if all edges going into the sink node t are used to their full capacity (left with 0). If this condition is met, run Ford-Fulkerson on the graph again with $x = 2$. Continue this, incrementing x by 1 each time until the algorithm does not use up all capacity of the edges with x capacity originally. When this happens, set p to be the final p where Ford-Fulkerson gave the result of all capacity being used. The final precise plot will have trees in all subplots with an edge that has capacity 1.

This algorithm works because it makes sure that every row and column has the same number of trees, which is p, because of the orientation of the graph. p will always stay the same for all rows and columns since x is associated with many edges and we update them collectively. We also know that p will be maximized because we take the final value for x that is "successful"; that is, the largest value that makes sure that the number of trees and columns is still the same before that condition can no longer be met.

The runtime for this algorithm is the runtime of Ford-Fulkerson, which is $O(f^*E)$. Here, our max flow (f^*) is n^2, due to the possible number of edges from $r_1, ..., r_n$ and $c_1, ..., c_n$. The number of edges (E) is $n^2 + n$, for the n^2 possible edges just stated added to the $2n$ edges between the source to the rest of the graph and the graph to the sink. Thus, the runtime for this algorithm is $O(n^2(n^2 + 2n)) = O(n^4)$.

4. **Solution.**

- We can simply formulate this adopted network flow problem into a linear program. Letting f_{uv} be the amount of flow between two nodes on the edge (u, v), and c_{uv} be the capacity of the edge (u, v), we set up the following LP goal and constraints.

  Goal:

  $$max \sum_i f_{si}$$

  Constraints:

  $$f_{uv} \leq c_{uv}$$

  $$f_{uv} \geq 0$$

  And for every vertex w besides $w = s$ and $w = t$,

  $$\frac{1}{2} \sum_{(u,w)} f_{uw} - \sum_{(w,v)} f_{wv} = 0$$

  These constraints are logical: for capacity, we say that the flow of an edge must be less than or equal to its capacity; for nonnegativity, we say that the flow of an edge must be at least 0;

and for conservation, which is what makes this different than a typical network flow problem, we say that half of the flow into all edges must equal the flow that comes out.

- For this question, we take a similar approach. We let f_{uv} be the amount of flow between two nodes on the edge (u, v), and let c_{uv} be the capacity of the edge (u, v) (using *cap* instead of c since we use c for the cost in this problem), and c_e is the fixed cost for each unit of flow through the edge e. We will use two LP problems to solve. First, we maximize the flow going through the graph, which is illustrated below:

  Goal:

  $$max \sum_i f_{si}$$

  Constraints:

  $$f_{uv} \leq cap_{uv}$$

  $$f_{uv} \geq 0$$

  And for every vertex w besides $w = s$ and $w = t$,

  $$\sum_{(u,w)} f_{uw} - \sum_{(w,v)} f_{wv} = 0$$

  This will give us the maximum flow. From here, set the answer to this LP problem, $max \sum_i f_{si}$, equal to some constant x for future reference. Next, we will use a second LP program to minimize the cost of the maximum flow. Consider the following linear program:

  Goal:

  $$min \sum_{(u,v)} c_{uv} * f_{uv}$$

  Constraints:

  $$f_{uv} \leq cap_{uv}$$

  $$f_{uv} \geq 0$$

  And for every vertex w besides $w = s$ and $w = t$,

  $$\sum_{(u,w)} f_{uw} - \sum_{(w,v)} f_{wv} = 0$$

  $$\sum_i f_{si} = x$$

  The last constraint guarantees correctness, because it ensures that the only paths that are taken are paths that have flow equal to the maximum flow, and we then minimize the cost per unit of those paths.

5. **Solution.**

   To create a linear time algorithm to compute the new maximum flow after adding 1 to a given edge e, we will assume that we are given the final residual after running an algorithm such as Ford-Fulkerson, and we will work with that graph.

After adding 1 to a certain edge in the final residual graph, there are two options. This will either give a new path possible from s to t (where s is the source and t is the sink), which will happen if the edge in the residual graph was initially 0. In this case, we add one to the final max flow that we are given. In the other case, 1 is added to the capacity of an edge that can't be reached anyway in the residual graph. In this case, our given max flow does not change, since it won't be affected by this edge.

To determine which case the given edge falls under, run a depth-first search from s to t in the residual graph after adding 1 to the capacity of the given edge. Since DFS can find if a path between two nodes exists, if a path from s to t does indeed exist, then add 1 to the max flow, and if no path exists, then return the original max flow. This algorithm works correctly because it checks if the edge that has 1 added to it affects the overall max flow or not in the original graph with DFS, determining whether or not we need to increase the max flow by 1.

In the case where a given edge e is decreased by 1, we again work with the final residual graph, and we again have two cases. In the first case, after subtracting from an edge in the residual graph, it stays greater or equal to 0, which occurs when that edge in the residual graph was initially positive. In this case, everything done up to this point was valid (that even after subtracting that edge still has the capacity to handle the amount of flow necessary). In this case, the max flow does not change. In the other case, an edge becomes negative after subtracting 1 from it, meaning that in the residual graph it was initially at a capacity of 0. In this case, something we did to find the max flow was not valid, because the algorithm must have run through an edge with 0 capacity. In this case, we must subtract one from the total max flow given.

To determine which case the given edge falls under, we will again DFS, but this time running it backwards (from the sink to the source). If there is a valid path from t to s, then we know that the given edge forwards must be negative, because a negative flow backwards will give a positive flow. So, if this backwards DFS finds a path, then subtract 1 from the total max flow, and if it does not find a path, then there is not a negative edge when going forwards in the graph, so return the original max flow. This algorithm works correctly, because it checks to see if a given edge is negative or not once some edge's capacity is decreased by 1, decreasing the overall max flow if an edge is negative, since the max flow would not be valid in this case.
The runtime for both algorithms is simply the runtime of DFS, which is $O(|V| + |E|)$, since we are assuming that we are given the residual graph of the given graph.

6. **Solution.**

Similar to the problem in section, let the row strategy be (y_1, y_2, y_3, y_4) and the column strategy be (x_1, x_2, x_3, x_4), meaning row i is chosen with probability i and column j is chosen with probability x_j. Because a positive payoff goes to the row player, we want to maximize the row strategy's expected payoff and minimize the column strategy's expected payoff.

Consider the following linear program to maximize the row strategy, letting a be the maximum payoff:

Goal:

$$a = max(3y_1 + 6y_2 - 3y_3 - 7y_4, y_1 - 2y_2 - 2y_3 + 4y_4, -2y_2 + 3y_3 - 5y_4, -4y_1 - 3y_3 + 7y_4)$$

Constraints:

$$a - 3y_1 - 6y_2 + 3y_3 + 7y_4 \leq 0$$
$$a - y_1 + 2y_2 + 2y_3 - 4y_4 \leq 0$$
$$a + 2y_2 - 3y_3 + 5y_4 \leq 0$$
$$a + 4y_1 + 3y_3 - 7y_4 \leq 0$$
$$y_1 + y_2 + y_3 + y_4 = 1$$

Next, we must write a linear program to minimize the column strategy, letting b be the minimum payoff:

Goal:

$$b = min(3x_1 + x_2 + 4x_4, 6x_1 - 2x_2 - 2x_3, -3x_1 - 2x_2 + 3x_3 - 3x_4, -7x_1 + 4x_2 - 5x_3 + 7x_4)$$

Constraints:

$$b - 3y_1 - 6y_2 + 3y_3 + 7y_4 \geq 0$$
$$b - y_1 + 2y_2 + 2y_3 - 4y_4 \geq 0$$
$$b + 2y_2 - 3y_3 + 5y_4 \geq 0$$
$$b + 4y_1 + 3y_3 - 7y_4 \geq 0$$
$$x_1 + x_2 + x_3 + x_4 = 1$$

I plugged these equations into Mathematica to act as an LP solver. I got the row player's strategy as:

$$(y_1 = 8/53, y_2 = 161/583, y_3 = 221/583, y_4 = 113/583)$$

and the column player's strategy as:

$$(x_1 = 81/583, x_2 = 11/53, x_3 = 234/583, x_4 = 147/583)$$

Mathematica shows that the total value or equilibrium of the game was -224/583, meaning that the game is biased to the column player, and that the row player always has a tendency to lose. Thus, since the game is fair when the equilibrium is 0, the column player should compensate the row player 224/583 units in order to make the game fair.

I collaborated on this problem set with Nathan Lee, Sam Bieler, and Kate Schember.