PSET 6
CS124: Data Structures and Algorithms
Prof. Mitzenmacher

Dhilan Ramaprasad
dhilanramaprasad@college.harvard.edu

Collaborator: No one :(
Others: A little advice from Ryan G.

I ***still*** haven't been to Esther/Amy's office hours in a long time. I miss going to their office hours :(
I ***still*** haven't been to Rachel/Rose's section in a long time. I miss going to their section :(

Now, I, uh, really, uh, like, miss Professor Mitzenmacher :(

I ***still*** don't miss Piazza.

1 Partially Reflecting Boundary—Random Walk

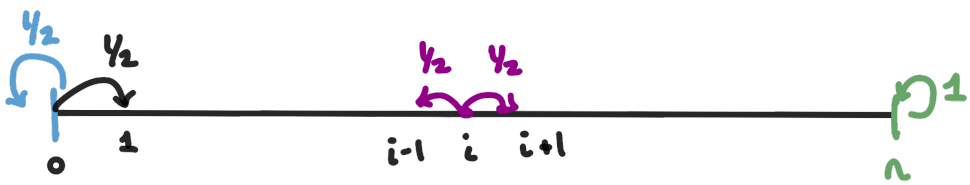


Figure 1: *Probabilistic Model of Random Walk*

Given a partially reflecting boundary at 0, we can model our probabilistic model of step-advancement ("moves") as in Figure 1 above.

Now, let S represent the number of remaining steps (or moves) to reach n. We can take:

$$S_N = 0 \tag{1}$$

$$S_i = \frac{1}{2}S_{i-1} + \frac{1}{2}S_{i+1} + 1 \qquad \text{for } 1 \leq i \leq n-1 \tag{2}$$

Our base case for $i = 0$ is worth noting, too:

$$S_0 = \frac{1}{2}S_1 + \frac{1}{2}S_0 + 1 = 2\left(\frac{1}{2}S_1 + 1\right) = S_1 + 2 \tag{3}$$

A first look at a graphical interpretation for various values of n indicates a concave-down quadratic function (unrolling method shown in detail below when solving for constants) which we can represent as the general factored form with constants a, s, and r:

$$E_i = -a(i-s)(i-r)$$

Of course, given our base case where $E_n 0$, we can rewrite our expected form:

$$E_i = -a(i-\mathbf{n})(i-r)$$

Now our goal is to find a and r. Performing a similar analysis as to that done in Section 8, we can unwrap some terms:

$$E_n = 0 \tag{4}$$

$$E_{n-1} = \frac{1}{2}E_{n-2} + 1 = -a(n-1-n)(n-1-r) = a(n-1-r) \tag{5}$$

$$E_{n-2} = \frac{1}{2}E_{n-3} + \frac{1}{2}E_{n-1} + 1 = -a(n-2-n)(n-2-r) = 2a(n-2-r) \tag{6}$$

We can perform a back-substitution now that we know the value of E_{n-2} which is required to evaluate the left hand

side of Equation 5:

$$\frac{1}{2}(2a(n-2-r))+1 = a(n-1-r)$$
$$1 = a$$

And we find a further reduced expected form:

$$E_i = -1(i-n)(x-r) \tag{7}$$

A method to solve for r was not immediately clear to me, so I returned to experimentation. Given $n = 3$, I could create a set of expected coordinates (as I did above to predict the concave-down quadratic function):

$$n = 3 \tag{8}$$
$$E_3 = 0 \tag{9}$$
$$E_0 = E_1 + 2 \qquad \text{from Equation 3} \tag{10}$$
$$E_1 = \frac{1}{2}E_0 + \frac{1}{2}E_2 + 1 \implies E_2 + 4 \qquad \text{implied via substitution} \tag{11}$$
$$E_2 = \frac{1}{2}E_1 + \frac{1}{2}E_3 + 1 \implies 6 \tag{12}$$

Back-substitution, again, yields a full set of coordinates:

$$(3,0),\ (2,6),\ (1,10),\ (0,12)$$

Using our most up-to-date general form (Equation 7), we can determine r for the specific case of $n = 3$:

$$E_2 = -(2-3)(2-r) = 6$$
$$r = -4$$
$$\text{Guess: } r = -n-1$$

With our guess of $r = -n - 1$, we can re-write our final generalized form for expected remaining steps given n and current position, i:

$$E_i = -(i-n)(i-(-n-1)) \tag{13}$$
$$E_i = -(i-n)(i+n+1) \tag{14}$$

By unrolling with different values for n, it can be confirmed that this general solution to the recurrence is correct.

2 MAX FLOW = MIN CUT

I apologize for not drawing this out in Tikz—time constraints led me to stick to handwritten graphics.

Using the **Ford-Fulkerson** algorithm, I performed the method of augmenting paths on the given graph:

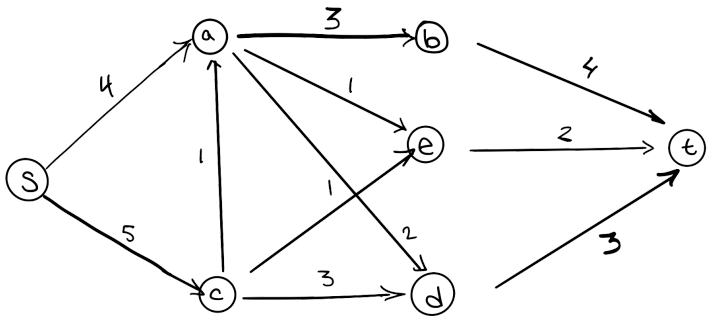
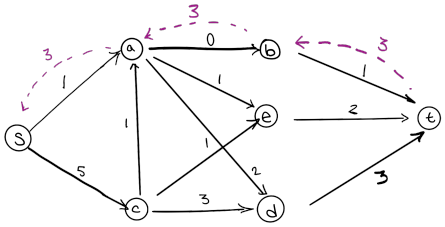
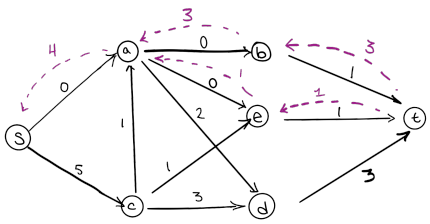


Figure 2: *Original Graph*

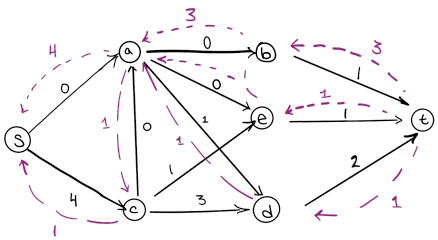
Residual graphs during Ford-Fulkerson and component depth-first search:



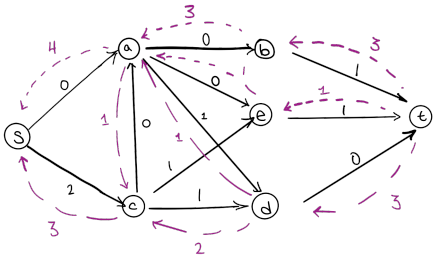
(a) *S-A-B-T*



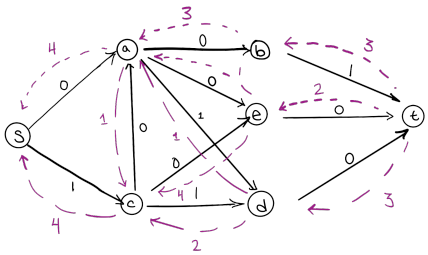
(b) *S-A-E-T*



(c) *S-C-A-D-T*



(d) *S-C-D-T*



(e) *S-C-E-T*

By the step shown in Figure 3e, we see no more paths from s to t, so the Ford-Fulkerson algorithm stops. Considering the "back-edges" (purple, dashed) from t, we can determine our maximum flow:

MAX FLOW = MIN CUT = 8

By the Ford-Fulkerson algorithm, we can also find, specifically, our minimum s-t cut. On the iteration at which the algorithm stopped, we can partition our graph into vertices reachable from s: $\{s, a, c, d\}$ and those which are not reachable from s (**note:** we do include the back-edges (purple, dashed) of the residual graph in this determination, and if there were a path from s to t, we would know that the Ford-Fulkerson algorithm had not yet completed). Naturally, our partition and the minimum cut can be seen:

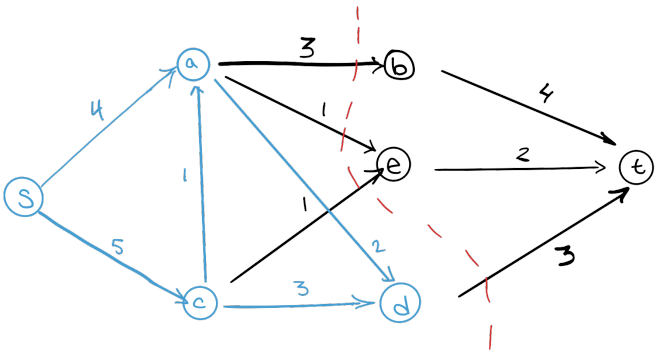


Figure 4: *Minimum Cut (Dashed, Red)*

By visual analysis, we see that the *cut capacity* = 8 = *max flow* $\implies$ we have a *min cut.*

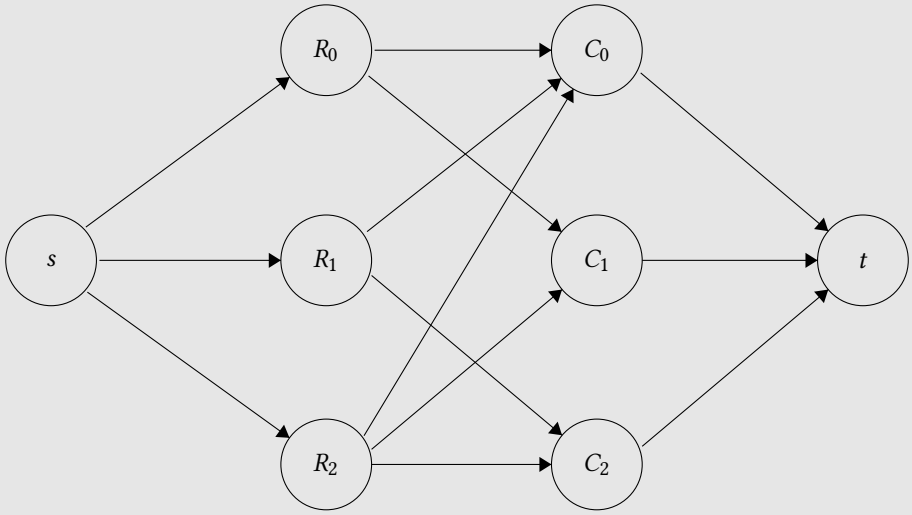
3 Palm Trees in Polynomial Time

In Section 9, we were presented a patients-to-eligible-hospitals setup similar to this problem (minus the symmetry afforded to us by the given $n \times n$ matrix). If we treat our given plots of land similar to the patient-hospital setup, we can imagine n rows as a set of n nodes, and our n columns as a separate set of n additional nodes. Arability of plots can be represented as edges with weight 1 "shipping" a palm tree from the corresponding *row* node to *column* node (i.e., we are eligible to plant in this (row, column) plot as indicated by the available edge; however, in the end, we need not necessarily use this flow). A "super-source" can supply p flow to each row; however, a "super-sink" can again only take p flow from each column. (Note, we could do the transpose, and achieve a similar network.)

Consider the setup for a 3×3 square plot of land:

1	1	
1		1
1	1	1

Table 1: *Subplots ($1 \implies$ arable)*



Edges from S to R_i: weight = p
Edges from R_i to C_i: weight = 1
Edges from C_i to t: weight = p

Now, all that remains is running the Ford-Fulkerson or Edmond-Karp algorithm to determine maxflow.

If **MAXFLOW = $\mathbf{n \cdot p} \implies \exists$ solution.**

The above is true, because each column, C_i must supply p flow for a solution to exist, and there are n columns $\implies$ n C_i nodes.

Runtime analysis:

There are $2n$ nodes and at most n^2 vertices, so using E-K we would expect:

$$\sim O(n^4)$$

Using F-F, we would expect:

$$\sim O(n^2 \cdot n \cdot p) = O(n^3 p)$$

The setup outlined above is predicated on knowledge of the correct p which can be easily found via an $O(n^2)$ algorithm wherein two size-n arrays keep track of totals of plots with soil per row and column (iterative process updated through an entire traversal of $n \times n$ matrix). Locating the global minimum in the two size-n arrays will provide an **upper-bound** for p (ideally reachable). It should be noted that a loose upper-bound for p is n.

Because this one-time p-finding-process is much smaller than the run of E-K or F-F, it is omitted from the asymptotic analysis; however, now knowing an upper bound for p we can revise our runtime for F-F: $\implies O(n^4)$.

Finally, it should also be noted that the process described above must be run for each value of p we try up to the upper bound of p (note @759 on Piazza mentions that if $\exists$ a satisfiable p such that maxflow = np, then $p - 1$ also has a satisfiable maxflow = $(n)(p-1)$, so we would run E-K or F-F p times before guaranteeing success or no-success).

Revised runtime analysis:

Running E-K or F-F p times, given $p \sim O(n)$:

$$\sim \mathbf{O(n^5)}$$

Supposedly, in speaking to Jaylen W., this runtime can be reduced as well via the note @759 on Piazza wherein we can "save" our residual graphs each run of F-F [if $\exists$ a satisfiable p such that maxflow = np, then $p - 1$ also has a satisfiable maxflow = $(n)(p-1)$]; thereby, far reducing the majority of our runtime. I am not sure how to prove correctness for this extension, so I will just leave it as a note.

3.1 Failure

It is worth noting that I had believed I could solve this problem in ~ $O(n^2)$ runtime via a greedy algorithm—after all, who doesn't love a simple, easy to implement greedy alg? Unfortunately, it seems, this "Sudoku" type problem (as my friend Benny put it) is not so easy...that's why Sudoku is challenging, I guess. Here's some of my failed code with a failed output:

```
Original Land:
0 0 0 1
0 1 1 0
1 1 0 1
1 1 1 0
Tracker:
1 2 3 3
2 3 2 2
P: 1
Tracker, now:
1 1 1 1
1 1 1 1
Final Land:
0 0 0 1
0 1 0 0
0 0 0 1
0 0 0 0
```

Figure 5: *Solvable if planted palm trees on anti-diagonal*

```
void fillTracker(vector< vector<int> > &land, vector< vector<int> > &tracker,
int dimension) {
    for (int i = 0; i < dimension; i++) {
        for (int j = 0; j < dimension; j++) {
            if (land[i][j] == 1) {
                tracker[0][i]++; // increment row count
                tracker[1][j]++; // increment col count
            }
        }
    }
}

int findmin(vector< vector<int> > &tracker, int dimension) {
    int globalmin = tracker[0][0];

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < dimension; j++) {
            globalmin = (tracker[i][j] < globalmin) ? tracker[i][j] : globalmin;
        }
    }
    return globalmin;
}

void setmin(vector< vector<int> > &tracker, int dimension, int p){
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < dimension; j++) {
            tracker[i][j] = p;
        }
    }
}
```

```
void planters(vector< vector<int> > &land, vector< vector<int> > &tracker,
int dimension, int p) {
    for (int i = 0; i < dimension; i++) {
        for (int j = 0; j < dimension; j++) {
            if ((tracker[0][i] > 0) && (tracker[0][j] > 0) && (land[i][j] == 1)){
                land[i][j] = 1;
                tracker[0][i]--; // decrement row count
                tracker[1][j]--; // decrement col count
            }
            else {
                land[i][j] = 0;
            }
        }
    }
}

// RANDOM MATRIX GENERATOR
void randm(vector< vector<int> > &land, int dimension) {
    random_device seed;
    static mt19937 generator(seed());
    uniform_int_distribution<int> dis(0, 1);
    for (int i = 0; i < dimension; i++) {
        for (int j = 0; j < dimension; j++) {
            land[i][j] = dis(generator);
        }
    }
}

int main(int argc,char* argv[])
{
    int dimension = 4;
    if (argc == 2){
        dimension = atoi(argv[1]);
    }

    vector<int> dimensionlong(dimension, 0);
    vector<vector<int> >land(dimension, dimensionlong), tracker(2, dimensionlong);
    randm(land, dimension);
    fillTracker(land, tracker, dimension);
    int p = findmin(tracker, dimension);
    setmin(tracker, dimension, p);
    planters(land, tracker, dimension, p);
    return 0;
}
```

4 Reduction to Linear Programming

4.1 Half 'n Half

Let's first begin by defining variables to represent our maximization goal as well as explicit and inherent constraints, taking our network as a directed graph.

It is not specified, so I will take we have a source (or sources) s_i whose supply is unconstrained. We have a destination t whose "in-flow" is the sum of all out-flows of nodes with directed edges to t.

$$\textbf{max} \sum_{\forall(v,t)\in E} f_{(v_i,t)}$$

$$\frac{1}{2} \sum_{\forall(u,v)\in E} f_{(u,v)} = \sum_{\forall(v,w)\in E} f_{(v,w)}$$

Finally, we have to constrain all flows to capacities, k, along edges!

$$0 \leq f_{(u,v)} \leq k_{(u,v)} \qquad \forall(u,v) \in E$$

This reduction to LP is synonymous, as we equivalently indicate that half the flow into a vertex is lost, and we maximize for the final flow into t.

4.2 Always low prices

Now, we add an additional goal. Given a found max-flow found in Section 4.1, we seek to minimize cost, c.

Using the linear program from the previous section, we can assign our **MAX FLOW = F**. Now, we must minimize cost while ensuring that our flow reaching t sums to the previously found F.

$$\textbf{min} \sum_{\forall(u,v)\in E} c_{(u,v)} f_{(u,v)}$$

$$\sum_{\forall(v,t)\in E} f_{(v_i,t)} = F$$

Finally, we have to constrain all costs to be non-negative!

$$0 \leq c_{(u,v)} \qquad \forall(u,v) \in E$$

This reduction to LP is synonymous with the problem statement, as we equivalently constrain our flow to node t to be the previously found maximum flow and merely minimize for cost given that we're meeting the aforementioned constraint.

5 Two-Player Game

5.1 Row Player

For the row player, we wish to maximize the payout := Z. We can write the following constraints given probabilities, r_i, corresponding to the row player's selection of each row.

$$Z \leq 3r_0 + 6r_1 - 3r_2 - 7r_3$$
$$Z \leq r_0 - 2r_1 - 2r_2 + 4r_3$$
$$Z \leq 0r_0 - 2r_1 + 3r_2 - 5r_3$$
$$Z \leq -4r_0 + 0r_1 - 3r_2 + 7r_3$$

Re-arranging, we have our full constraints for the row player as follows:

$$Z - 3r_0 - 6r_1 + 3r_2 + 7r_3 \leq 0 \tag{15}$$
$$Z - r_0 + 2r_1 + 2r_2 - 4r_3 \leq 0 \tag{16}$$
$$Z + 2r_1 - 3r_2 + 5r_3 \leq 0 \tag{17}$$
$$Z + 4r_0 + 3r_2 - 7r_3 \leq 0 \tag{18}$$
$$r_0 + r_1 + r_2 + r_3 = 1 \tag{19}$$
$$r_0,\ r_1,\ r_2,\ r_3 \geq 0 \tag{20}$$
$$Z \textbf{ unrestricted} \tag{21}$$

5.2 Column Player

For the column player, we wish to minimize the payout := W. We can write the following constraints given probabilities, c_i, corresponding to the column player's selection of each row.

$$W \geq 3c_0 + c_1 + 0c_2 - 4c_3$$
$$W \geq 6c_0 - 2c_1 - 2c_2 + 0c_3$$
$$W \geq -3c_0 - 2c_1 + 3c_2 - 3c_3$$
$$W \geq -7c_0 + 4c_1 - 5c_2 + 7c_3$$

Re-arranging, we have our full constraints for the row player as follows:

$$W - 3c_0 - c_1 + 4c_3 \geq 0 \tag{22}$$
$$W - 6c_0 + 2c_1 + 2c_2 \geq 0 \tag{23}$$
$$W + 3c_0 + 2c_1 - 3c_2 + 3c_3 \geq 0 \tag{24}$$
$$W + 7c_0 - 4c_1 + 5c_2 - 7c_3 \geq 0 \tag{25}$$
$$c_0 + c_1 + c_2 + c_3 = 1 \tag{26}$$
$$c_0,\ c_1,\ c_2,\ c_3 \geq 0 \tag{27}$$
$$W \textbf{ unrestricted} \tag{28}$$

5.3 Solved

Using `online-optimizer.appspot.com` (i.e., the first link on Google), I optimized for both the Row and Column Players.

5.3.1 Row Player

The following probabilities should be used for the Row Player's strategy:

$$r_0 = 0.1509434$$
$$r_1 = 0.2761578$$
$$r_2 = 0.3790738$$
$$r_3 = 0.193825$$

5.3.2 Column Player

The following probabilities should be used for the Column Player's strategy:

$$c_0 = 0.1389365$$
$$c_1 = 0.2075472$$
$$c_2 = 0.4013722$$
$$c_3 = 0.2521441$$

5.4 Value

The value of the game is $\approx$ **−0.38** payout to the **row** player,
which represents $\approx$ **0.38** winnings for the **column** player.
The column player should thus pay the row player that amount to create a fair starting point.