CS 4349 Lecture—November 8, 2017

Main topics for #lecture include applications of #maximum_flow and #minimum_cut such as #edge_disjoint_paths, #maximum_matching, #assignment_problem, and #baseball_elimination.

Prelude

- Homework 9 due.
- Homework 10 due Wednesday, November 15th.

Faster Algorithms for Maximum Flow and Minimum Cut

- As we'll see today, maximum flows and minimum cuts are used for a lot more than just transporting supplies or blowing up train tracks.
- And they're used so often, that computing them faster is still an active area of research.
- Jeff Erickson has a nice table of showing improvements over the years.
- The latest result by James Orlin is from 2012. You can compute maximum flows and minimum cuts in only O(VE) time.
- Very few people understand this algorithm, and I am not one of them, so I'm certainly not going to try teaching it to you.
- But for your homework and exams, you should claim maximum flow and minimum cut take O(VE) time to compute. Put this on your cheat sheet!

Edge-Disjoint Paths

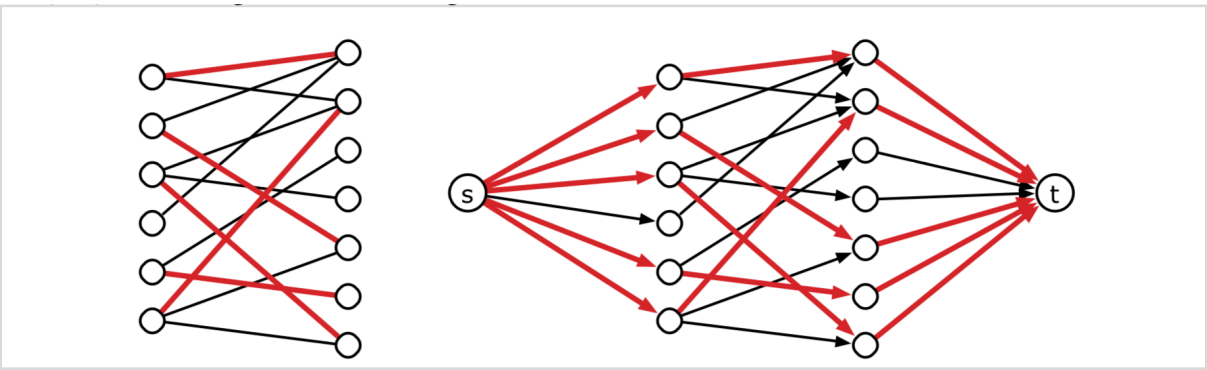
- Given directed graph G and vertices s and t, find the maximum number of edge-disjoint paths from s to t. These may not be vertex disjoint.
- We can assign each edge a capacity of 1.
- As we saw Monday, the maximum flow will assign an integer, 0 or 1, to every edge.
- We can then find some path in the subgraph of flow 1 edges, grab it, reduce the flow on those edges by 1, and recurse to find a number of paths equal to the value of our flow.
- And it will be a maximum number of edge disjoint paths: If there are k of them, then there is a flow of value at least k: put 1 unit on each edge of these paths.
- We can do all this in O(VE) time with Orlin's algorithm, but this is overkill. The (s, t)-cut ({s}, V \ {s}) has capacity at most V - 1, so the maximum flow has value at most V - 1. Basic Ford-Fulkerson takes O(EV) time as well.

Vertex Disjoint Paths

- We could also ask for a bunch of vertex disjoint paths.
- To do that, set up a flow problem again, except we assign a limit l(v) = 1 to the amount of flow on each vertex's incoming edges.
- Then we apply the solution to Homework 9 Problem 2 Part D.
- Split each vertex v into two vertices, v_in and v_out. All incoming edges to v now go to v_in. All outgoing edges from v now come from v_out. Add a single edge between them with capacity(v_in ➔ v_out) = l(v).

Maximum Matchings in Bipartite Graphs

- Let G = (U union W, E) be an undirected bipartite graph: There are no edges going between vertices of U or between vertices of V.



- A *matching* is a subset of edges where every vertex is incident to *at most one* of those edges. Basically, we're trying to match up pairs of vertices from U and W.
- We want a matching with the maximum number of edges.
- To do that with flows, we change the graph a bit.
  1. Orient all the edges from U to W.
  2. Add new vertices s and t.
  3. Add edges from s to every vertex of U.
  4. Add edges from every vertex of W to t.
  5. Assign all edges a capacity of 1.
- So now we can compute a maximum flow f^* using Orlin or Ford-Fulkerson.
- It assigns 0 or 1 to every edge. At most one unit of flow enters every vertex of U and at most one unit leaves every vertex of W. So the edges between them carrying 1 unit of flow form a matching of size |f^*|.
- And any matching can be turned into a flow of the same value just by giving 1 unit to the matching edges and send one unit into/out of every vertex that got matched.
- Grabbing the matching from f^* takes O(E) time. Computing f^* takes O(VE) time since the maximum flow value is at most |U|.

Assignment Problems

- Maximum Matching is a special case of the unweighted binary assignment problem:
- We're given two disjoint finite sets X and Y representing two different kinds of resources which we'd like to assign to one another. For example, they could be:
  - web pages and servers
  - computational jobs and machines
  - hospitals and interns
  - customers and ice cream flavors
- Our goal is to choose the largest possible collection of pairs (x, y) where x in X, y in Y, subject to some constraints:
  - Each element x in X can appear in at most c(x) pairs.
  - Each element y in Y can appear in at most c(y) pairs.
  - Each pair (x, y) can appear in the output at most c(x, y) times.
- Usually, each bound c(x), c(y), c(x, y) is a small non-negative integer or infinity. The pairs are the assignments.
- Maximum matching was just a special case where c(z) = 1 for all vertices z, and c(x, y) = 0 or 1 depending on whether pair xy was an edge in the graph.
- For a more interesting example, maybe customer x is hungry enough to purchase c(x) scoops of ice cream, we have c(y) scoops of flavor y to sell, and the costumer x will get bored of flavor y after eating c(x, y) scoops of it. We want to maximize the number of scoops we sell total.
- The algorithm for this problem is almost the same as for maximum matching.
- Make a directed graph G = (V, E) where V = X union Y union {s, t}.
  - s ➜ x gets capacity c(x)
  - y ➜ t gets capacity c(y)
  - x ➜ y gets capacity c(x, y)
- We then compute a maximum flow |f^*| which will be integral since the capacities are integral.
- Now the flow value on each edge x ➜ y will be the number of times you pair up x and y.
- And like before, any assignment can be represented by a flow of the same value by just assigning to each edge x ➜ y the number of times x and y are paired. You assign to each edge from s or t the number of times that element is paired.
- We actually do want to use Orlin's algorithm in this case, because the value of the maximum flow is only bounded by the sum of the constraint values. The whole thing takes O(VE) = O(n^3) time, where n = |X| + |Y|.
- Remember, running times should be in terms of the input size, not any data structures you build while running the algorithm.

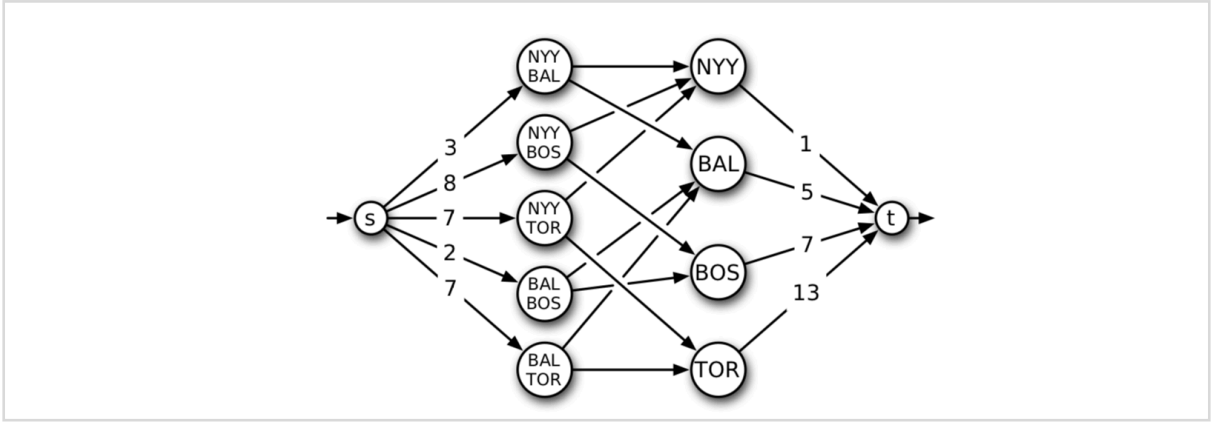
Baseball Elimination

- One last example for today.
- Every year, members of the various divisions in Major League Baseball compete for a spot in the playoffs. The goal is to have the best win-loss record in the division, but sometimes a team becomes "mathematically eliminated" weeks before the regular season ends, because it's impossible for them to lead their division.
- For example, here are the standing for the American League East on August 30, 1996:

Team	Won–Lost	Left	NYY	BAL	BOS	TOR	DET
New York Yankees	75–59	28		3	8	7	3
Baltimore Orioles	71–63	28	3		2	7	4
Boston Red Sox	69–66	27	8	2		0	0
Toronto Blue Jays	63–72	27	7	7	0		0
Detroit Tigers	49–86	27	3	4	0	0	

- That's the teams win-loss record, how many games they had left to play total, and how many they have left to play with the four other teams in this table.
- So the Detroit Tigers are in poor shape. But maybe they can still win the division.
- If they win every other game, they'll be ahead of where New York is right now. But New York has games left to play, and every loss for New York is a win for another team. If New York wins even two games, they'll beat Detroit no matter what. So maybe New York loses every game? But then Boston wins those 8 games with New York, and they end up beating Detroit anyway. So maybe New York wins one game against Boston…
- OK, this is getting ridiculous. Let's just write an algorithm to figure this out for us.
- Say we're given two arrays W[1 .. n] where W[i] is the number of games team i has already won, and G[1.. n][1 .. n] where G[i][j] is the number of times team i will play team j before the end of the regular season.
- We want to determine if team n can at least tie for most wins at the end of the regular season.
- This is an assignment problem: we want to assign a winner to each game.
- Let R[i] = sum_{j} G[i][j] be the number of games left for team i.
- If team n wins all R[n] remaining games, then they come in first place if and only if every other team wins at most W[n] + R[n] - W[i] of its remaining games.
- OK, so here's the assignment problem.
  - X is the (n - 1 choose 2) team pairs g_{i, j} playing games excluding team n (ex. NYY BAL)
  - Y is n - 1 teams t_i that might win these games
  - c(g_{i, j}) = G[i][j], the number of games that pair will play
  - c(t_i) = W[n] - W[i] + R[n], the number of games that team can win if team n is going to be the leader
  - c(g_{i, j}, t_i) = c(g_{i,j}, t_j) = infinity. We have no limit on how many times a team can win for each pair. c(g_{i, j} , t_k) = 0 if k ≠ i and k ≠ j.
- Here's the graph we'd build for this particular example. I just left out the edges of 0

capacity.



- Team n can lead if and only if we can find an assignment that uses every game. In other words, we need to saturate every edge out of s.
- This flow network has O(n^2) vertices and edges. So Orlin's algorithm will take O(VE) = O(n^4) time.
- Back to the 1996 American League East. To saturate edges leaving s, we'd need a flow of value 27. But the total capacity of edges entering t is only 26. I'm afraid Detroit is mathematically eliminated.