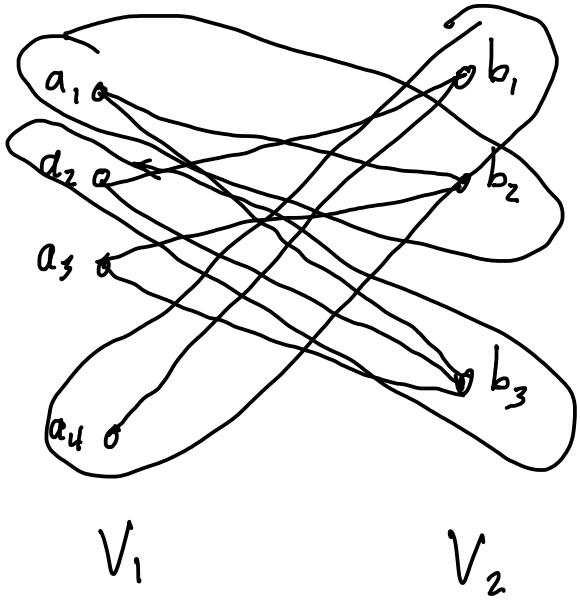
Network Flows: definition, Ford-Fulkerson algorithm and theorem (max flow = min cut)

Applications
Linear Programming

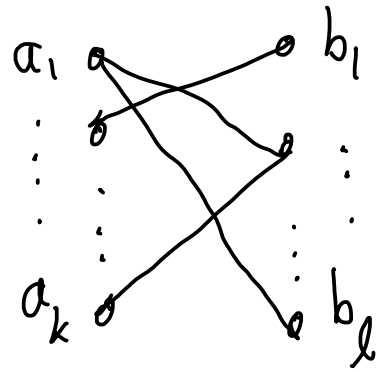
Problem: Maximum Bipartite Matching

IN: undirected bipartite graph G
OUTPUT: a "matching" in G — a subset M subsetof E with no two edges sharing an endpoint — with the maximum size

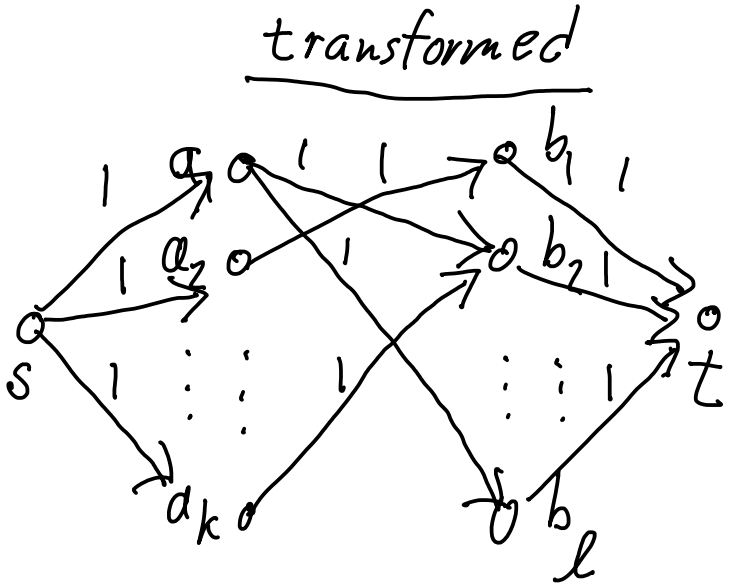


---

original input



transformed

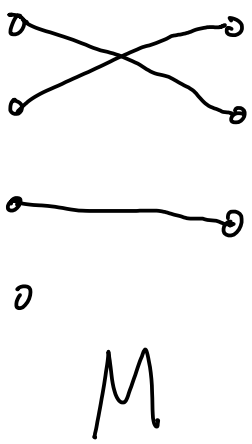


1. Create network N from G, as illustrated above

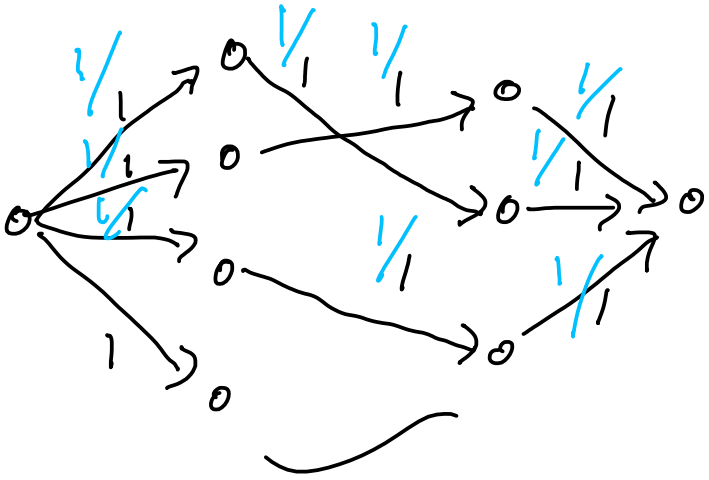
2. Find a maximum flow in N
3. Output M = { (a_i,b_j) : f(a_i,b_j) = 1 }
   (edges in the original graph with flow = 1)

Runtime? polynomial time: linear for steps 1 & 3, polynomial for 2
Correctness?
Notice: we solved matching with a "reduction": at a high level, transforming one problem into another so that solutions correspond to each other.

- Q: How do we know that every solution to original problem can be transformed into a solution to the network flow problem?
  A: Every matching of size |M| can be transformed into a flow on N with value |f| = |M|.

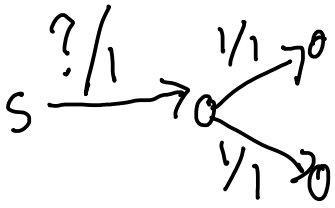


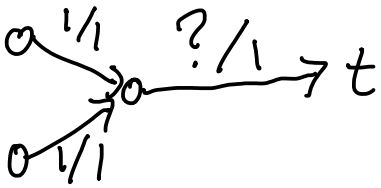
M



Since this is true for every matching, this means maximum flow value ≥ maximum matching size.

Q: How do we know every valid flow corresponds to some matching?
A: The only possible sources of problems are situations like this:



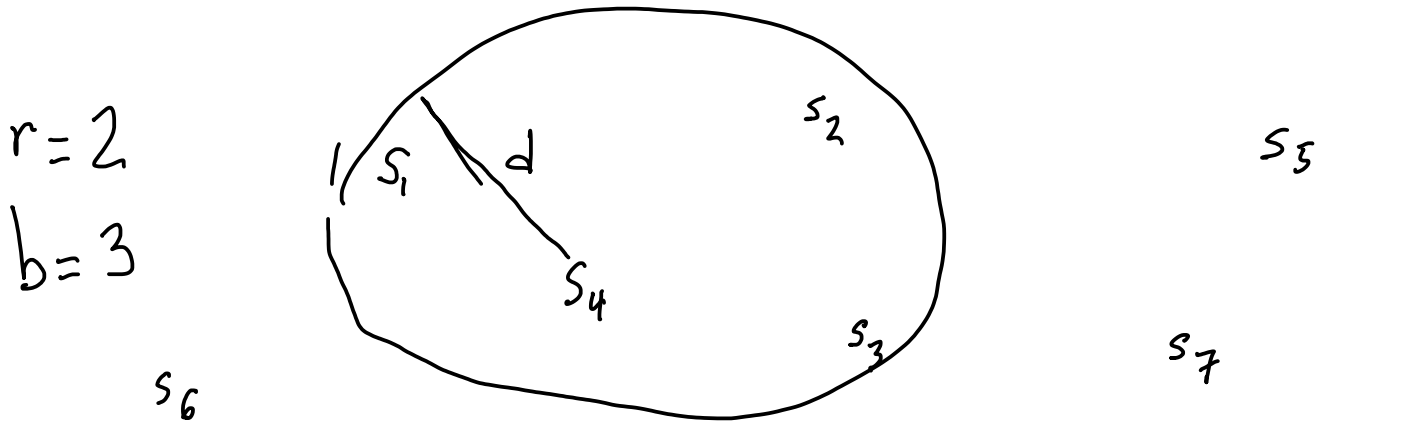


Consequence: every valid flow with integer values yields a matching $M = \{ (a_i,b_j) : f(a_i,b_j) = 1\}$ with $|M| = |f|$
(No two edges of M can share an endpoint because that would require the flow to break conservation.)
This means maximum matching size ≥ maximum flow value.

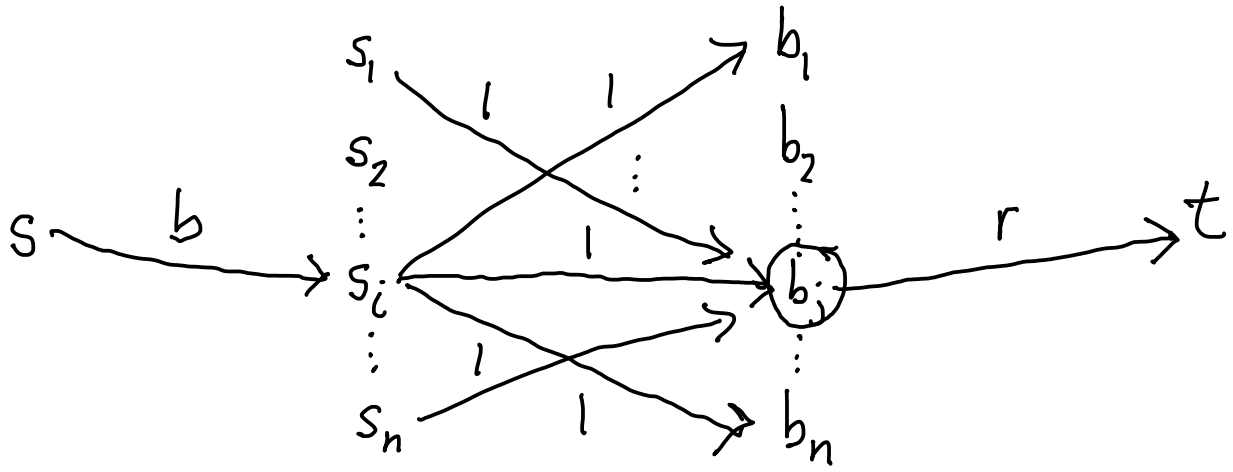
So maximum matching size = maximum flow value.

Solving a problem using Network Flows means:
1. Giving an explicit algorithm based on creating a network, running Ford-Fulkerson, and extracting a solution to the original problem based on the network solution.
2. Arguing that solutions to both problems correspond to each other (both directions)

Backup Sensor Problem (from Problem Set 5, Fall 2014)
Input: Sensors s1,s2,...,sn (each sensor si has real-number coordinates (xi,yi)), distance parameter $d \in R+$, redundancy parameter $r \in Z+$, and backup parameter $b \in Z+$ with $b \geq r$.
Output: Backup sets B1,...,Bn where $Bj \subseteq \{s1,...,sn\}-\{sj\}$ for each j, every sensor in Bj is within distance d of sj , every Bj contains at least r elements, and every sensor belongs to at most b backup sets.
(If this is not possible, output the special value nil.)



Key Idea: matching between sensors s1,s2,…,sn and backup sets B1,B2,…,Bn.



Create network N:

- edge (si,bj) with capacity 1 iff i ≠ j and dist(si,sj) ≤ d
- edge (s,si) with capacity b (for every i)
- edge (bj,t) with capacity r (for every j) — NOTE: solving the easier problem with size of each backup set exactly r

Find max flow

Output Bj = { si : f(si,bj) = 1 }

IF |Bj| < r (for some j), THEN there is no solution!

Correctness:

- Consider a collection of backup sets B1,…Bn — not necessarily full, but where no sensor belongs to more than b backup sets and sensors in a backup set are all within distance d of the target sensor.
  This yields a valid flow in N:
  f(s,si) = number of backup sets containing si
  f(si,bj) = 1 iff si belongs to Bj
  f(bj,t) = size of Bj
  Here, |f| = sum of sizes of Bj (over all j)

So maximum flow value ≥ maximum sum of sizes of all backup sets.
NOTE: This only works if NO backup set has size > r; in that case, simply remove sensors from backup sets until they all have size ≤ r — this is still a perfectly good solution to the original problem.

- Consider any valid integer flow in N and let
  Bj = { si : f(si,bj) = 1 }.
  Then, each si belongs to at most b backup sets (because c(s,si) = b).
  Also, each Bj has size at most r (because c(bj,t) = r).
  Finally, sum of sizes of backup sets = |f| (= f^in(t)).
  So maximum sum of sizes of backup sets ≥ maximum flow value — as long as each backup set has size at most r.

Consequence: for backup sets limited to size r,
maximum flow = maximum sum of sizes of backup sets.
So maximum flow = rn, then we have a solution; else, no solution.

Political Advertising Problem

	urb.	sub.	rur.
roads	-2	5	3
guns	8	2	-5
farm	0	0	10
gas	10	0	-2

Numbers represent 1000's of votes gained/lost per $1000 in advertising, for each platform and riding type.
Target: at least 50,000 urban votes, 100,000 suburban, 25,000 rural.
Goal: spend as little as possible to achieve the goal.

What are we looking for? Define variables:

x1 = $1000's spent on advertising “building roads”
x2 = ... for guns
x3 = ... for farm
x4 = ... for gas

Important: these are REAL valued.

Objective: minimize x1 + x2 + x3 + x4

Constraints:

x1, x2, x3, x4 ≥ 0
-2 x1 + 8 x2 + 0 x3 + 10 x4 ≥ 50 — constraint for urban votes
5 x1 + 2 x2 + 0 x3 + 0 x4 ≥ 100
3 x1 - 5 x2 + 10 x3 - 2 x4 ≥ 25

In general, a Linear Program consists of

- variables $x_1, x_2, \ldots, x_n$ (real numbers)
- objective function: $c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$, to be either minimized or maximized, where $c_1, \ldots, c_n$ are real valued constants
- constraints: every constraint of the form
  $$a_1 x_1 + a_2 x_2 + \ldots + a_n x_n \leq / = / \geq b$$
  where $a_1, \ldots, a_n, b$ are real valued constants

NOTE: “>” and “<” are NOT allowed in constraints!

NOTE: If variables are restricted to be integers only, then we have an “Integer Program.”

There are polytime algorithms to solve Linear Programs.
There are no known polytime algorithms to solve Integer Programs (in fact, there very likely are none).

Simplex Algorithm: works very well in practice BUT
worst-case time actually exponential…
“Interior point methods”: worst-case polytime BUT
in practice worse than Simplex…