The Wider World of Algorithms

Matchings, Flows, and Beyond

Algorithms: Design and Analysis, Part II

Stable Matchings

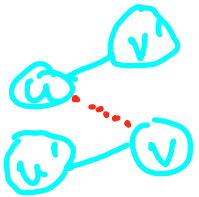
Consider two nodes sets U and V ("men" and "women").

For simplicity: assume $|U| = |V| = n$.

Each node has a ranked order of the nodes on the other side. (different for different nodes)

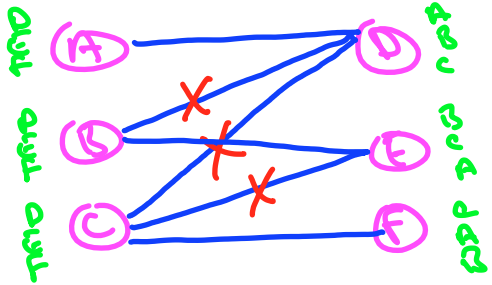
Examples: hospitals & residents, colleges & applicants.

Stable matching: a perfect matching (i.e., matches each node of U to a distinct node of V) such that:
if $u \in U$ and $v \in V$ are not matched, then either u likes its mate v' better than v, or v likes its mate u' better than u.



Gale-Shapley Proposal Algorithm

While there is an unattached man u:
- u proposes to the top woman v on his preference list who hasn't rejected him yet
- each woman entertains only the best proposal received so far



[invariant: current engagements = a matching]

Theorem: terminates with a stable matching after $\leq n^2$ iterations. [in particular, a stable matching always exists!]

Gale-Shapley Theorem

① each man makes $\leq n$ proposals $\Rightarrow$ $\leq n^2$ iterations.

② terminates with a perfect matching.
Why? If not, some man rejected by all women.
$\Rightarrow$ all n women engaged at conclusion of algorithm
$\Rightarrow$ all n men engaged at end, as well [contradiction]

③ terminates with a stable matching. Why? Consider some u, v not matched to each other.

Case 1: u never proposed to v.
$\Rightarrow$ u matched to someone he prefers to v.

Case 2: u proposed to v.
$\Rightarrow$ v got a better offer, ends up matched to someone she prefers to u.

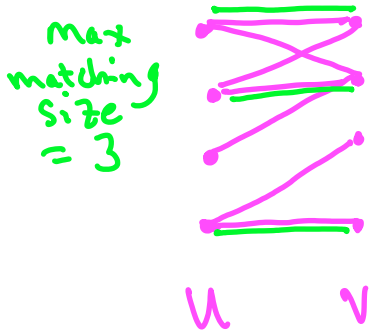
QED!

Tim Roughgarden

Bipartite Matching

Input: bipartite graph $G = (U, V, E)$. [each $e \in E$ has one endpoint in each of U, V]

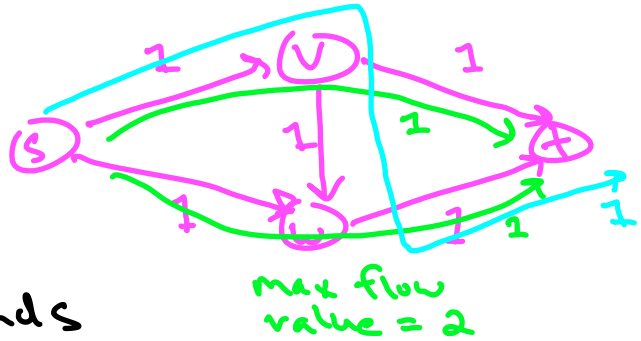
Goal: computing a matching $M \subseteq E$ [i.e., pairwise disjoint edges] of maximum size.



Fact: there is a straightforward reduction from this problem to the maximum flow problem.

The Maximum Flow Problem

Input: directed graph $G = (V, E)$.
- source vertex s, sink vertex t
- each edge e has **capacity** u_e

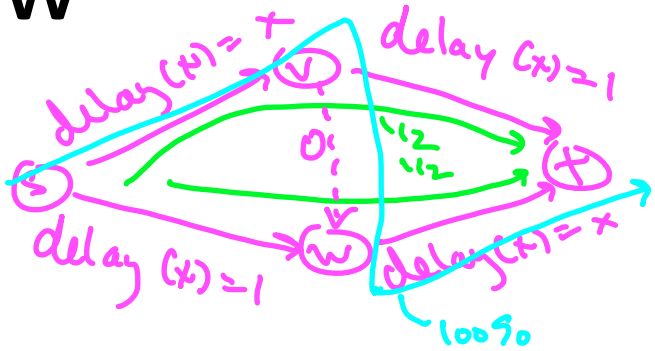


Goal: Compute the s-t "flow" that sends as much flow as possible.

Fact: Solvable in polynomial time. (e.g., via non-trivial greedy algorithms based on "augmenting paths")

Selfish Flow

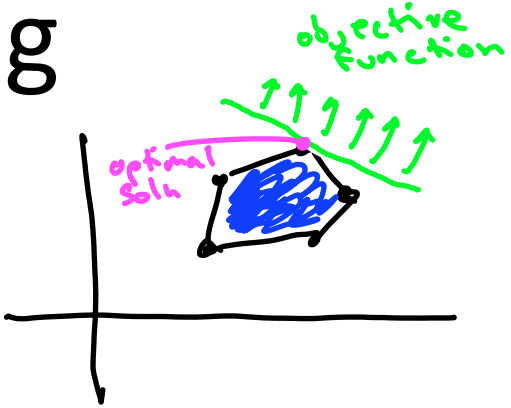
- flow network
- 1 unit of "selfish" traffic
- each edge has a delay function
  [travel time as function of edge load]



Steady state: with a 50/50 split, commute time = 1.5 hours.

Braess's Paradox ('68): after adding a teleporter from v to w, commute time of selfish traffic degrades to 2 hours!

Linear Programming



The general problem: optimize a linear function over the intersection of halfspaces.

⇒ generalizes maximum flow plus tons of other problems

Fact: Can solve linear programs efficiently (in theory and in practice).

⇒ very powerful "black-box" subroutine

Extensions: convex programming, integer programming.

polynomial-time solvable under mild conditions

NP-hard in general

Other Topics and Models

- deeper study of data structures, graph algorithms, approximation algorithms, etc.
- geometric algorithms
  - low-dimensional (e.g., convex hull) 
  - high-dimensional (e.g., nearest neighbors in information retrieval)
- algorithms that run forever (usually in real time)
  [e.g., caching, routing]
- bounded memory ("streaming algorithms")
  [e.g., maintain statistics at a network router]
- exploiting parallelism (e.g., via Map-Reduce/Hadoop)

Epilogue