The Wider World of Algorithms

Algorithms: Design and Analysis, Part II

Matchings, Flows, and Beyond

Stable Matchings

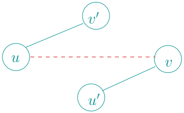
Consider two node sets U and V ("men" and "women")

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 <u>not</u> matched, then either u likes its mate v' better than v, or v likes its mate u' better than u.

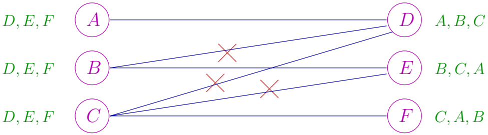


Gale-Shapley Proposal Algorithm

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]

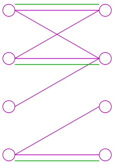
(3) Terminates with a stable matching. Why? Consider some u, v not matched to each other.
Case 1: u never proposed to v.
$\Rightarrow$ u matched to someone he prefers to v.
Case 2: u proposed to v.
$\Rightarrow$ v got a better offer, ends up matched to someone she prefers to u. QED!

Bipartite Matching

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

Goal: Compute 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.



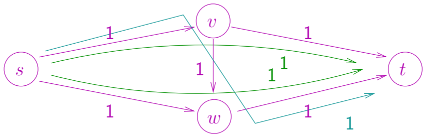
max matching
size = 3

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.

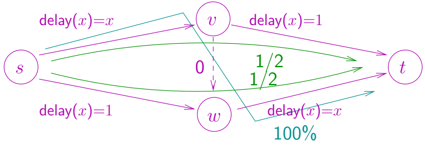


max flow value $= 2$

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]



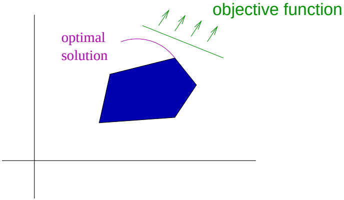
Steady state: With a 50/50 split, commute time = 1.5 hours

Braess's Paradox ('68): After adding a teleported 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.

$\Rightarrow$ Generalizes maximum flow plus tons of other problems



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

$\Rightarrow$ 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.
- Gerometric 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