Stable matchings: Gale-Shapley

What’s representative?

Helper properties
unmatched college: has not offered to some student
student options get better during a run

Invariants (for loops)
once student matched, stays matched

Non-determinism: different possible runs
here: with same result: same stable matching best for college,
worst for student among all stable matchings

These issues appear in many other algorithms

Algorithmic Complexity

\[ f(n) = O(g(n)) \] (and \( \Omega \), \( \Theta \) are relations between functions)

Can also see \( O(g(n)) \) as a class of functions that grow asymptotically not faster than \( g \)

\[ f(n) = O(g(n)) \] means
there exist \( c > 0 \) and \( n_0 \) s.t. \( f(n) \leq cg(n) \) \( \forall n \geq n_0 \)

Can choose \( c \) and \( n_0 \) as needed (arbitrarily large)

\[ f(n) = \Omega(g(n)) \] (lower bound) equivalent to \( g(n) = O(f(n)) \)

\[ f(n) = \Theta(g(n)) \] equivalent to \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \)

Graph Searches

Need to distinguish directed from undirected graphs

Undirected graphs
DFS has tree edges and back edges (at least 2 levels up)
BFS has tree edges and non-tree edges (as most \( \pm 1 \) difference)

Directed graphs
DFS has tree, back, cross and forward edges
BFS non-tree edges:
go at most 1 level down, same level, or any level up

Cycle detection: DFS, only back edges
Detect for directed graphs: mark nodes unvisited/open/closed

Directed Acyclic Graphs

DFS has no back edges (only tree, cross and forward edges)

Topological Ordering / Sorting
in linear time: \( O(V + E) \)

Some algorithms more efficient
e.g. find longest path (dynamic programming)

Amortized Analysis

Often, useful to count total work rather than work per iteration

naive analysis of BFS and DFS: \( O(V) \), actual bound is \( O(V + E) \)
more complex: Union-Find, negative cycle detection

Minor data structure changes can improve runtime bound
e.g., updating indegree for topological sorting
Greedy

Make local choice that seems best now
- earliest deadline for jobs
- shortest edge for Kruskal, Prim
- closest node for Dijkstra

For problems with \textit{optimal substructure} property

\textbf{Correctness Arguments}

Greedy stays ahead
Exchange argument (compare to purported optimum)
careful if several optimal solutions

Divide and Conquer

Divide problem into several parts
- Solve each instance
- Combine solutions to solve original problem

\textbf{Recurrences}

Unroll (draw recursion tree)
Guess solution ($f(n) \leq c \cdot g(n)$), prove by strong induction
Use Master Theorem

Recurrences: Master Theorem

Let $T(n) = aT(n/b) + f(n)$, with $a \geq 1$, $b > 1$. Then:

1. $T(n) = \Theta(n \log_a b)$ when $f(n) = O(n^{\log_a b - \epsilon})$ for some $\epsilon > 0$ and $f(n)$ grows polynomially slower than $n^{\log_a b}$
2. $T(n) = \Theta(n \log_a b \log \log n)$ when $f(n) = \Theta(n \log_a b)$ (border case)
   $T(n) = \Theta(n^{\log_a b \log \log n})$ when $f(n) = \Theta(n^{\log_a b \log \log n})$
3. $T(n) = \Theta(f(n))$ when $f(n) = \Omega(n^{\log_a b + \epsilon})$ for some $\epsilon > 0$ and $af(n/b) < cf(n)$ for some $c < 1$ when $n$ sufficiently large
   $f(n)$ grows polynomially faster than $n^{\log_a b}$

Does not cover everything: gaps between 1 and 2, and 2 and 3
Guess and prove by induction for other cases

Strengthening Assumptions

Solve more than was asked for
- sort-and-count for counting inversions
Return more than was asked for
- tree problems: balanced trees, well-ordered nodes
Avoid recomputations!

Dynamic Programming

Overlapping subproblems: avoid recomputing common partial results

- Often: computing optimum: \textit{optimal substructure} but evaluates multiple choices, unlike greedy
- Binary choice (choose or don’t choose an item)
- $n$-ary choice (multiple options): rod cutting
- Adding one more dimension (subset sum, knapsack)

Pseudopolynomial cases: proportional to one of input values actually exponential in number of bits for that input value

Space-Time Tradeoff

Use more time to save some space

- Sometimes, same asymptotic time (more rarely)
  - Hirschberg sequence alignment, $T(n) = 2T(n/2) + O(n^2) \Rightarrow O(n^2)$
- More often: higher time complexity for smaller space
  - coin game: $T(n) = T(n-1) + O(n^2) \Rightarrow O(n^3)$
Network Flows: Ford-Fulkerson

Flow networks directed, source-sink, edge capacities
Maximum flow = minimum cut.
Residual graph for max flow disconnects s from t (cut).
Max flow: forward edges saturated, backward edges have no flow.
Complexity: \(O(mnC_{\text{max}})\) (Ford-Fulkerson), \(O(m^2n)\) (Edmonds-Karp), \(O(mn^2)\) (Dinitz)
Solve: node capacities, node-disjoint paths, edge-disjoint paths, etc.

Max flow: forward edges saturated, backward edges have no flow.

Maximum bipartite matching: \(O(mn)\) time

Polynomial-Time Reduction

- \(Y \leq_P X\)
  - \(\text{solve}_Y(y\text{input})\)
  - \(\text{Construct } x\text{input} \quad // \text{poly-time}\)
  - \(\text{foo }= \text{solve}_X(x\text{input}) \quad // \text{poly # of calls}\)
  - \(\text{return yes/no based on foo } \quad // \text{poly-time}\)

- Statement about relative hardness
  1. If \(Y \leq_P X\) and \(X \in P\), then \(Y \in P\)
  2. If \(Y \leq_P X\) and \(Y \notin P\) then \(X \notin P\)

To prove NP-Completeness, must reduce from NP-complete problem (reduce NP-complete problem to the one considered)

P and NP / Solver vs. Certifier

- \(P\): Decision problems with a polynomial time algorithm.
- \(NP\): Decision problems with a polynomial time certifier.

Intuition: A correct solution can be certified in polynomial time.

Let \(X\) be a decision problem and \(s\) be problem instance (e.g., \(s = (G,k)\) for \text{Independent Set})

Poly-time solver. Algorithm \(A(s)\) such that \(A(s) = \text{YES} \iff \text{correct answer is YES}\), and running time polynomial in \(|s|\)

Poly-time certifier. Algorithm \(C(s, t)\) such that for every instance \(s\), there is some \(t\) such that \(C(s, t) = \text{YES} \iff \text{correct answer is YES}\), and running time is polynomial in \(|s|\).

- \(t\) is the “certificate” or hint. Must also be polynomial-size in \(|s|\)

Finding Reductions

Problems are very close (map to one another)
- \text{SetCover} and \text{HittingSet}

Problems may be are duals:
- \text{VertexCover} and \text{IndependentSet}

Sometimes we construct gadgets
- \text{3-SAT} to \text{IndependentSet}

Approximation Algorithms

- \(\rho\)-approximation algorithm
  - Runs in polynomial time
  - Solves arbitrary instance of the problem
  - Guaranteed to find a solution within ratio \(\rho\) of optimum:
    \[
    \frac{\text{value of our solution}}{\text{value of optimum solution}} \leq \rho
    \]

- Sometimes non-obvious (spanning tree to get cycle in TSP), both greedy and non-greedy (choose both nodes for vertex cover).

Examples:
- 1.5-approximation for Load Balancing
- 2-approximation for Clustering
- 2-approximation for Vertex Cover

Randomized Algorithms

- Efficient in expectation
- Optimal with high probability
- Break (undesired) symmetry
- Show some solution exists, or derive bound on number

Types of randomized algorithms:
- Fail with some small probability (Monte Carlo)
- Always succeed, but running time is random (Las Vegas)

Techniques used in proof:
- expected value, union bound, write sum in two ways