Analysis of algorithms

Misc

- **Any algorithm to solve a problem with an input of size f(n) is very likely to be Ω(f(n))** as it is likely to need to read the majority of the input to be fully correct
  - For a graph with no parallel arcs and n nodes, we are passing in n vertices and up to nC2 edges so f(n) is $O(n^2)$ and Ω(n)
  - Noteable counterexample is binary search but asking for input to be sorted is kind of cheating

Time complexities of graph implementations

- Convention is n=number of nodes, m=number of edges
- We call a graph sparse if m is O(n log n) and dense otherwise
- Adjacency matrix
  - Uses $\Theta(n^2)$ space
  - Check if there is an edge from a given node to another given node is $\Theta(1)$
  - Enumerating all edges is $\Theta(n^2)$
- Adjacency list
  - Uses $\Theta(m+n)$ space, checking if there is an edge with a given source s is $O(\deg(s))$
  - Enumerating all edges takes $\Theta(m+n)$
- If we have a dense graph with no parallel edges, $\Theta(m+n)$ is $\Theta(n^2)$ anyway so adjacency matrix is slightly better
- If we have a dense graph with parallel edges, adjacency matrix is better
- If we have a sparse graph, adjacency list is better

Greedy algorithms

Greediness

- Greedy algorithms sort the input into some specific order then process it linearly in that order, making irrevocable decisions about what to do with the current element to try to optimize some function
  - For the correct choice of ordering you can guarantee that the local optimum found by greedy is a global optimum
    - If this ordering is tractable to find, we have a good algorithm, otherwise we should try a different algorithm design
- When deciding on an ordering to design a greedy algorithm, chose a natural ordering then look for a counterexample and if you can't find one try to prove optimality

Methods for proving optimality: Big picture

- We are trying to prove that all other algorithms (which are not necessarily greedy) would give answers that are no better — be careful what you assume in your proof
- Generally, proofs only allow us to get optimality from the hypothesis of correctness, so don't forget to prove correctness as well
- **One approach: Transform an arbitrary input into an optimal solution and demonstrate we are enforcing the exact properties enforced by our algorithm**
- **Second approach: Transform an arbitrary optimal solution into what our algorithm would produce and demonstrate that the optimality is unaffected**

Methods for proving optimality: Actual methods

- Fixing inversions is sufficient for optimality **(introducing inversions can only make a solution the same or worse) and our solution is inversion free — call (i, j) an inversion if i comes before j in input (after sorting) but i comes after j in output**
  - **Proved in lecture: If there is an inversion, there is an inversion (k, m) s.t. k and m are adjacent in the output**
- **Exchange argument — transform an arbitrary optimal solution into that produced by ours inductively without breaking its correctness or optimality**
- Direct proof
  - E.g. Our algorithm does not enforce anything other than the constraints stated in and implied by the problem description — enforcing more constraints can only make the value of the function worse, enforcing less means correctness cannot be guaranteed

Divide and conquer algorithms

Recurrence trees

- Suppose we have $T(n) = a\,T(n/b) + f(n)$ if $n > 1$, $O(1)$ if $n = 1$
  This gives a tree with root $T(n)$, a^i nodes $T(n/b^i)$ at level i, and $a^{\log_b n}$ leaves $T(1)$
  Total cost of layer $i = a^i f(n/b^i)$ if $i < \log_b n$
  $T(n) =$ sum from $i=0$ to $i=n^{\log_b a}-1$ of $a^i f(n/c^i) + n^{\log_b a}$
  Suppose it is possible to write $a^i f(n/b^i) = g(n)h(i)$, then $T(n) = g(n) *$ sum from $i=0$ to $i=n^{\log_b a}-1$ of $h(i) + n^{\log_b a}$
- **Useful formulae:**
  - $\mathbf{a^{\log_b n} = n^{\log_b a}}$
  - $\mathbf{\Sigma_{i \in \{0, \ldots, k\}} r^i = (r^{k+1} - 1)/(r - 1)}$
  - $\mathbf{\Sigma_{i \in \{1, \ldots, k\}} i = k(k+1)/2}$

Master method

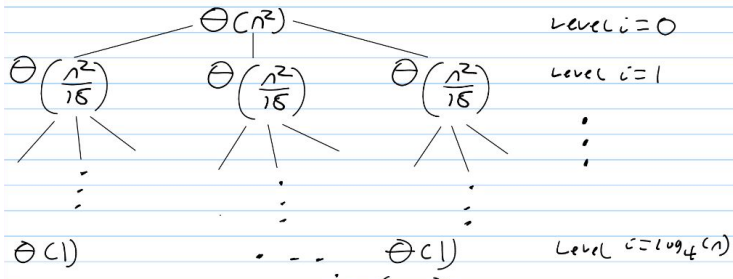
- Master theorem: If $T(n) = aT(n/b) + n^c$, $a \geq 1$, $b \geq 2$, $c \geq 0$, and $T(1) \in \Theta(1)$, then:
  - Case $c > \log_b a$: $T(n) \in \Theta(n^c)$
  - Case $c = \log_b a$: $T(n) \in \Theta(n^c \log n)$
  - Case $c < \log_b a$: $T(n) \in \Theta(n^{\log_b a})$
- Can replace all Θ with O or Ω
- Can replace n/b with $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$
- Can consider only n above some n_0

Derivation(ish) of master theorem

- Suppose we have $T(n) = aT(n/b) + f(n)$ if $n > 1$, $O(1)$ if $n = 1$
  There are a^i subproblems at level i
  There are $1 + \log_b n$ levels $(0, \ldots, \log_b n)$
  Subproblems have size n/b^i at level i
  At level i ($i \neq \log_b n$), total work done = $a^i f(n/b^i)$
  T(n) = sum of work at all levels before leaves + work done at leaves = sum over i of $[a^i/f(b^i)]f(n) + a^{\log_b n} O(1) = f(n) * \text{sum over i of } [a^i/f(b^i)] + n^{\log_a b} = f(n)[(n^{\log_b k}-1)/(k-1)]+n^{\log_b a}$ if there exists k s.t. for all i $k^i = a^i/f(b^i)$ and $k \neq 1$
  If there exists k but k = 1, we instead have $f(n)[\log_b n]+n^{\log_b a}$
- Now suppose $f(n) = n^c$ where c is an integer and let $k=a/b^c$
  Let $C = [1/(\log_b(a)-c-1)]$
  Then, $T(n) = Cn^c[(n^{\log_b (a) - c}-1)+n^{\log_b a}$ if $k \neq 1$
  $= [1/(\log_b(a)-c-1)][n^{\log_b a} - n^c]+n^{\log_b a}$ if $k \neq 1$
  Suppose $c > \log_b a$, then $-C > 0$ and $T(n) = -C[n^c-n^{\log_b a}] + n^{\log_b a} \in \Theta(n^c)$
  Suppose $c < \log_b a$, then $C > 0$ and $T(n) = C[n^{\log_b a}-n^c]+n^{\log_b a} \in \Theta(n^{\log_b a})$
  Whereas if $c = \log_b a$, then k=1, so $T(n) = n^c \log_b n+n^{\log_b a} = n^c[\log_b n + 1] \in \Theta(n^c \log n)$

An example

$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 3T(\frac{n}{4}) + \Theta(n^2) & \text{otherwise} \end{cases}$



At level i work done $= 3^i \Theta\left(\left(\frac{n}{4^i}\right)^2\right)$, $i < \log_4(n)$

$= \Theta\left(\left(\frac{3}{16}\right)^i\right)\Theta(n^2)$

Thus $T(n) = \sum_{i=0}^{\log_4(n)-1}\left(\Theta\left(\left(\frac{3}{16}\right)^i\right)\Theta(n^2)\right) = \Theta(n^2)\sum_{i=0}^{\log_4(n)-1}\left(\Theta\left(\left(\frac{3}{16}\right)^i\right)\right) + \Theta(\log_4(n))$
$+ \log_4(n)\,\Theta(1)$

$= \Theta(n^2)\left[\frac{\left(\frac{3}{16}\right)^{\log_4(n)} - 1}{\frac{3}{16} - 1}\right] + \Theta(\log_4(n))$

$= \Theta(n^2)\left[\frac{1 - n^{\log_4\left(\frac{3}{16}\right)}}{1 - \frac{3}{16}}\right] + \Theta(\log_4(n))$

$\in \Theta(n^2)$ as $\log_4\left(\frac{3}{16}\right) < 0$

Verifying with Master theorem: $2 > \log_4(3)$, so $T(n) \in \Theta(n^2)$

Dynamic programming

Dynamic programming (DP)

- In divide and conquer we aim to create disjoint subproblems as we solve them independently then combine and we don't want to duplicate effort unnecessarily
- Dynamic programming is a special case of divide and conquer where we embrace overlapping of sub-problems (using memoization)
    - Exploits the optimal substructure property — optimal solution can be built from optimal solutions to subproblems
- *By building up a solution inductively, DP avoids having to find a good way of partitioning the data but makes us less able to parallelize than divide and conquer which is very amenable to parallelization*

Doing DP

- **Derive a bellman equation OPT(i) = { max {...} if i ...; ... }**
  - **Can assume we already have values for OPT(1), ..., OPT(i−1) which is very nice**
  - If derivation is fully explained, you've basically already done an inductive proof of correctness?
- **Top-down implementation:**

```
function OPT(i)
    If M[i] is null
        Set M[i] using bellman equation using calls to OPT (this will involve some if blocks)
    return M[i]
```

- **Bottom-up implementation:**

```
function OPT(i)
    M[0] = base case
    For i = 1 to n
        Set M[i] using bellman equation using M (this will involve some if blocks)
    return M[n]
```

- Bottom-up means you are fully responsible for the table being filled out in the right order
- To find which steps to take to achieve the value given by OPT: Store in another table which value of the parameter max is ranging over gave the max and then move through that table from the entry for n to the entry for the entry for n and so on until we reach 1 pushing onto a stack as we go

Graph algorithms

Minimum spanning tree algorithms

- MINIMUM-SPANNING-TREE: Given an undirected weighted graph find a spanning (same node set as the graph) tree (connected acyclic subgraph) with minimal total cost
- Kruskal's algorithm: Start with T empty. while T is not spanning add an arc that does not introduce a cycle to T with minimal weight to T
  - Cycles can be (efficiently) detected using dfs/bfs
- **Prim's algorithm: Start with T as an arbitrary node. While T is not spanning add an arc <u>from a node in the tree to a node outside</u> the tree with minimal weight to T**
  - This is much the same greedy approach as Kruskal but the anti-cycling is implicit this time

Complexity classes

Tractability

- **The complexity of a problem is the time or space efficiency of an optimal algorithm for it**
- A problem is in P (aka PTIME) iff there exists an algorithm that solves it in time $O(n^k)$ for some k
- A problem is tractable if it can be solved in a reasonable amount of time by a computer
    - Cobham's thesis: P = {problem: problem is tractable} (every tractable problem is in P and vice versa)
        - *Problems: Big O hides constants (so a good looking algorithm may actually be slow or a bad looking algorithm may actually be good on instances that are likely to arise in the real world), high powers cannot reasonably be called tractable*
        - *No one has come up with a better idea though so we use it anyway*

Reductions

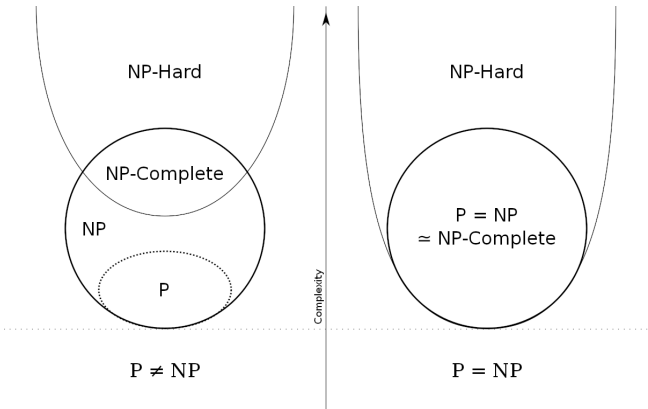
- **A reduction from X to Y is an algorithm for X that uses an algorithm for Y as an oracle (black box)**
- **X polynomial-time reduces to Y ($X \leq_P Y$) iff any instance of X can be solved using poly-time pre and post processing and polynomial (could be 0) calls to an oracle for Y**
  - Note the oracle does not necessarily run in polynomial time!
  - X is at most polynomially harder than Y (even though the notation makes it look like Y is harder than X) and so for the purposes of tractability no harder than Y
  - Theorem: **If $X \leq_P Y$ and Y is in P, then X is in P**
    - Corollary (contrapositive/contradiction): **If $X \leq_P Y$ and X is not in P, then Y is not in P**
  - Note that if $X \leq_P Y$ and X is in P, we can't say whether Y is in P or not
- If X poly-time reduces to Y and Y poly-time reduces to X, then X and Y are poly-time equivalent ($X \equiv_P Y$)

NP

- **A problem X is in NP iff an algorithm A exists s.t. A(i, s) = Yes iff X(i) = Yes <u>and A runs in polynomial time if X(i) = Yes</u>**
  - i is an instance of X, s is a witness for i, A is a verifier for X given s
  - *Can think of this as the verifier must always give the correct answer to the problem (even if the witness is incorrect) and it must give this answer in polynomial time (potentially with the aid of the extra information given by the witness) provided the witness is correct*
  - *K&T say certificate and certifier instead of witness and verifier*
  - s must be of polynomial size in order for A to be able to run in polynomial time
  - We get to pick what s is when designing A
- P is a subset of NP as we can use the polynomial time algorithm as the verifier (we don't need a witness)

NP-Complete

- ***A problem X is NP-hard iff for every problem Y in NP there exists a poly-time reduction <u>from</u> <u>Y to X</u>***
  - *Intuitively, NP-hard is the class of problems that are at least as hard as the hardest problems in NP as an oracle for any one of them can be used to solve every other problem in NP in not much more time than the oracle takes and so they are at least as hard as every problem in NP*
  - *Alternatively (as the universal quantification is hard to prove):* ***A problem X is NP-hard if there exists Y s.t. Y is NP-hard and <u>Y reduces to X</u>*** *as then we can reduce every problem in NP to X by going via Y*
    - *Alternative intuition: As X is at least as hard as Y and Y is NP-hard X must be NP-hard*
- **A problem X is NP-complete iff <u>X is in NP</u> and X is NP-hard**
  - *If P=NP, then (obviously) every NP-complete problem is in P*
  - *If P≠NP, then (obviously) no NP-complete problem is in P*



NP-Complete Problems

- **k-SAT problem: Given a boolean formula in CNF where no clause has more than k literals, is there an assignment of values to variables that makes the formula true?**
- Theorem (Cook-Levin): SAT is NP-complete *by direct proof — because of this we can avoid having to deal with the universal quantification by building a chain of reductions from SAT*
- Theorem (Karp): **3-SAT is NP-complete** by reduction from SAT
- **INDEPENDENT-SET: Does there exist a subset of vertices of size k s.t. no vertices are adjacent?**
- **VERTEX-COVER: Does there exist a subset of vertices of size k s.t. every edge has an endpoint in it?**
- **The complement of a vertex cover is an independent set and vice versa** as for each edge in the graph at most one endpoint is in an independent and at least one endpoint is in a cover
    - **A graph with n vertices has an independent set of size k iff it has a vertex cover of size n−k and vice versa**
        - **INDEPENDENT-SET $\equiv_P$ VERTEX-COVER**
- 3-SAT $\leq_P$ INDEPENDENT-SET and so **INDEPENDENT-SET and VERTEX-COVER are** NP-hard and in fact **NP-complete**
- **SET-COVER: We have a set X and a subset S of 2^X such that the union over S is X. Is there a subset S′ of S such that |S′| = k and the union over S′ is X?**
- VERTEX-COVER $\leq_P$ SET-COVER and so **SET-COVER is NP-complete**

Flow networks

Algorithmic problems on flow networks

- **A flow network is a 5-tuple (V, E, s, t, c) where V is the set of vertices, E is the set of (directed) edges, s is the source (start) vertex, t is the sink (target) vertex and c is the capacity function** (c: $E \mapsto \mathbb{R}_{>0}$ or equivalently but more conveniently c: $V^2 \mapsto \mathbb{R}_{\geq 0}$ where c(x, y) = 0 iff (x, y) $\notin$ E)
- **A flow is a function f: $E \mapsto \mathbb{R}_{\geq 0}$ such that**
  - **Flows do not exceed capacities ($\forall e \in E.\ f(e) \leq c(e)$) and**
  - **Flow is conserved** (at every intermediate node flow in = flow out) **($\forall v \in V \setminus \{s, t\}.\ \sum_{(u, v) \in E} (f(u, v) - f(v, u)) = 0$)**
- **An s-t cut is a partition (A, B) of V s.t. s $\in$ A and t $\in$ B**
  - **Capacity of a given s-t cut (A, B) = c(A, B) = $\sum_{(u, v) \in A \times B} c(u,v)$ = sum of capacities of edges <u>from</u> nodes in A <u>to</u> nodes in B**
- MIN-CUT problem: Given a flow network find a s-t cut with minimal capacity
- **MAX-FLOW problem: Given a flow network find a flow with maximal value**
  - **Value of a flow = $\sum_{v \in V} (f(s, v) - f(v, s))$ = the net flow out of s**
- Net flow of a flow f across an s-t cut (A, B) = sum of the flow values of edges from A to B − sum of the flow values of edges from B to A
- Flow value lemma: The value of any flow is the net flow of it across any s-t cut
- **Max-flow-min-cut theorem: maximum flow in N = minimum capacity over all st cuts in N**
  - **Hence min-cut and max-flow are polytime equivalent**
  - To obtain a min-cut from a max flow: Use reachability in residual graph from start node to create the S partition and place everything else in the T partition

Ford-Fulkerson algorithm

- **Ford-Fulkerson algorithm:**

```
Start with a flow f of 0 everywhere in N
Construct the residual network N' of N, same vertices as N but edges are defined as
for each edge in N a forward edge with the remaining capacity (if non-zero) and a
reverse edge with the used capacity (if non-zero)
While there exists an augmenting path (a path from s to t) p in N'
     Let c be the minimum edge weight on p
     For each forward edge taken in p, increase f at the corresponding edge in N by c
     For each backward edge taken in p, decrease f at the corresponding edge in N by c
     Update N' accordingly
return f
```

- Ford-Fulkerson Theorem: A flow is maximal iff there are no augmenting paths
- Finding the path can be done in O(|E|) (bfs/dfs)
- If capacities are integers, terminates in O(|E|*(sum of capacities)) but capacities are unbounded so although it terminates it could take a very long time depending on the input
- If capacities are not quantized, may not terminate!
    - *Edmonds–Karp algorithm modifies Ford-Fulkerson to remove the arbitrary choice of augmenting path to provide a termination guarantee, by always using the augmenting path with the fewest edges (e.g. bfs) it gets $O(|V||E|^2)$ for any input*

Matchings

- **A matching in an undirected graph is a set of edges with no shared vertices** or equivalently such that each vertex in the graph is connected to a most one edge in the matching
- **MAX-MATCHING: Find a matching with maximal cardinality**
- **Can reduce MAX-MATCHING in a bipartite graph to MAX-FLOW by converting G to a flow network by: adding dummy source and sink nodes, directing all edges towards the second partition, and assigning all edges capacity 1**
- **A perfect matching is a matching that covers all vertices**
- **Hall's marriage theorem: Let G = (V, E) be a bipartite graph with partitions P, Q s.t. |P| = |Q|. Then, G has a perfect matching iff $\forall S \subseteq P$. $|N(S)| \geq S$ where N(S) = neighbourhood of S = $\{y: \exists x \in S: (x, y) \in E\} \subseteq Q$**