Theory of Algorithms

Tyler Chang
April 30, 2018

Big O Notation

$$f(n) = \mathcal{O}\left(g(n)\right) \iff f(n) \le C_0 g(n) \text{ for all } n \ge N \text{ and some } C_0.$$
$$f(n) = \Omega\left(g(n)\right) \iff f(n) \ge C_0 g(n) \text{ for all } n \ge N \text{ and some } C_0.$$
$$f(n) = \theta\left(g(n)\right) \iff f(n) = \mathcal{O}\left(g(n)\right) \text{ and } f(n) = \Omega\left(g(n)\right).$$

A problem X of size n is said to be solvable in polynomial time if there exists an algorithm A that computes the *exact* solution to X, and A runs in time (as measured by counting FLOPs) bounded by $\mathcal{O}(p(n))$, where $p(n)$ is a polynomial in n.

Recurrence Relations

The runtime of a problem X for an input of size n is given by

$$T(n) = \alpha\, T\left(\frac{n}{\beta}\right) + f(n)$$

where α, β are constants and f is a function of n. I.e., the time for computing a problem of size n is given in terms of the time to compute some smaller problem (as in a recursion algorithm).

Such relations can be resolved by hand (using a table), proved by induction, or (in special cases) by using **Master's Theorem**:

1. If $f(n) = \theta(n^c)$ with $c < \log_\beta \alpha$, then: $T(n) = \theta\left(n^{\log_\beta \alpha}\right)$.
2. If (for some $k \ge 0$) $f(n) = \theta(n^c \log^k n)$ with $c = \log_\beta \alpha$, then: $T(n) = \theta\left(n^c \log^{k+1} n\right)$.
3. If $f(n) = \theta(n^c)$ with $c > \log_\beta \alpha$, then: $T(n) = \theta\left(f(n)\right)$.

Dynamic Programming and Greedy Algorithms

Consider the class of problems characterized by finding the optimal (shortest, longest, largest, smallest, etc.) solution to some problem subject to some constraints. An optimization problem X with solution J is said to have *optimal substructure* if every subsolution of J is itself the optimal solution to a subproblem of X.

The following are conditions for solving an optimization problem via *dynamic programming*:

1. The problem has optimal substructure;
2. There are exists a polynomial number of subproblems, one of which is the solution;
3. There exists a natural ordering for solving this sequence of subproblems (with the final problem giving the solution to X); and
4. The optimal value for the first problem is easy to compute, and the optimal solution to each subsequent problem is obvious given the previous solution.

Such a problem can by solved in polynomial time using *dynamic programming* as follows: Solve the first problem, then the rest in order. Since there are polynomially many problems, each with a trivial solution given the previous, this can be done in polynomial time. The final solution is the solution to X.

For example, Dijkstra's Algorithm, BFS, and DFS all compute the shortest path between two nodes s and t in a graph G by starting at s then finding the shortest path to each of its neighborhing nodes $\{s_i\}$ (simply by travelling straight there). Then, exploiting the optimal substructure, the shortest path to each unvisted node x_j neighboring some s_i is given by the path to s_i union with the connection to x_j: $s \to s_i \to x_j$. Continuing in this fashion, if a path from s to t exists, eventually the shortest path will be found.

A problem X has *greedy substructure* if taking the "greedy choice" always ultimately leads to the optimal solution. A problem with greedy substructure can be solved by a *greedy algorithm.* For example, Prim's algorithm exploits the greedy substructure of the *s*-*t* connectivity problem above. In a graph with weighted edges, Prim's algorithm finds the minimum weight path from s to t by always taking the minimum weight path from a node x to one of its neighbors.

Output Sensitive Algorithms

An algorithm that computes a large solution (of size k) cannot run faster than $\mathcal{O}(k)$ since it must *at least* do one computation to report each solution. If k grows with the input n, then the algorithm is said to be *output sensitive.* Since we can't do better that $\mathcal{O}(k)$, we instead hope to compute a solution that is polynomial when the size of the output permits: I.e., the time complexity should be $\mathcal{O}\left(k + p(n)\right)$ where $p(n)$ is a polynomial of the input size n.

Many computational geometry problems such as range searching, Delaunay triangulations, Voronoi diagrams, and convex hulls are output sensitive. Often in these scenarios, storing the output in minimal space is also of interest, and *space complexity* can be analyzed similarly as time complexity.

Dual Problems

Given a problem X, if there exists a problem Y such that solving Y (provably) admits the solution to X, then Y is the *dual* of X. For example, the Hungarian Algorithm solves a minimum cost bipartite matching by converting it into a linear program, which admits a known polynomial time solution.

Randomization and Backward Analysis

Often it is easier to analyze how an algorithm will perform in expectation rather than how it will perform in the worst case. For such problems, one can often avoid pathologically bad input cases by randomly shuffling inputs. Such algorithms are analyzed via *backward analysis*, whereby the expected complexity is reverse engineered by playing the algorithm backward in time.

P vs. NP

A decision problem is a problem whose answer is either YES or NO. Any optimization problem can be easily transformed into a decision problem by asking: *Is there a solution such that the cost is less than (or greater than) some value?*. Decision problems are easier to compare since their solution spaces are equivalent, therefore they are preferred in algorithm analysis.

P is the class of decision problems that can be solved in polynomial time. Given an oracle (that always tells us the best solution), NP is the class of problems for which the oracle's answer could be checked to see if the conditions for a YES answer were met. Clearly, $P \subset NP$ but is it in fact true that $P = NP$?

To answer this, define a one-to-one map M between inputs to problem X and problem Y such that $M(y \in Y) \rightarrow YES$ in X if and only if y is a YES instance of Y. We say such a map is answer preserving.

If an answer preserving map M exists between X and Y and can be computed in polynomial time, we say X *reduces to* Y in polynomial time. Clearly, if we can solve Y in polynomial time, then we can also solve any X that reduces to Y in polynomial time using the map M.

Note that every decision problem in computer science can be reduced to a circuit satisfiability (C-SAT) problem since computers are circuits. Intuitively, this means that (up to a polynomial time difference) circuit satisfiability is at least as hard as (if not harder than) every problem that can be verified in polynomial time. Therefore, we say that C-SAT is NP-hard. Also, as a decision problem, C-SAT can be checked in polynomial time. Therefore, C-SAT is also in NP, and we say that it is NP-complete.

Therefore, any other problem that can be reduced to circuit satisfiability (or another NP-hard problem) must also be in the NP-hard class. And any problem that is NP-hard and is also in NP, must also be NP-complete. There are many problems that are known to be NP-complete, but it is still unknown whether $P = NP$. To prove $P \neq NP$, it would suffice to show that one problem in NP cannot be solved in polynomial time. To prove that $P = NP$, it would suffice to show that one NP-complete problem admits a polynomial time solution.