Complexity & Fine-Grained Complexity

Sam Ruggerio

Fine Grained Complexity

- What is Computational Complexity?
- What are reductions?
- What is FGC?
- What do people research in FGC?

Complexity Primer

- How do we determine the *difficulty* of a problem?
- There are many intuitive ways - but we're computer scientists!
- Determine difficulty based on the runtime of an algorithm that solves the problem.
- But what is our ground truth of computation? How do we know we have the best algorithm?

Complexity Primer

- Sometimes we don't know the best algorithm!
- We can still prove difficulty - just via other means.
- However, we will sometimes need to choose a *model of computation* as well.

Runtime Symbols

- $O(f(n))$, some function $g(n) \in O(f(n))$ if $g(n)$ grows as much or no faster than $f(n)$
  - $O(n)$ grows linearly w.r.t n. $O(n^2)$ grows quadratically.
  - $n\log(n) \in O(n^2)$ because it does not grow faster than $O(n^2)$. It is not in $O(n)$.
- $\Theta(f(n))$, ... "grows as-fast-as" ... ("Tight")
- $\Omega(f(n))$, ... "grows at-least as fast as" ("Lower Bound")

Runtime Symbols

- $o(f(n))$, some function $g(n) \in o(f(n))$ if $g(n)$ grows strictly less than $f(n)$
  - $n^2 \in O(n^2)$, but $n^2 \notin o(n^2)$
  - This is useful because we can say $n^{1.99999999} \in o(n^2)$ without needing specifics.
- $\omega(f(n))$, ... "grows strictly faster than" ... ("Lower Bound")

Runtime Summary

- You can view these Runtime Notations as (roughly) inequalities:
  - $g(n) \in O(f(n)) \iff g(n) \leq f(n)$
  - $g(n) \in o(f(n)) \iff g(n) < f(n)$
  - $g(n) \in \Theta(f(n)) \iff g(n) = f(n)$
  - $g(n) \in \omega(f(n)) \iff g(n) > f(n)$
  - $g(n) \in \Omega(f(n)) \iff g(n) \geq f(n)$
- But remember that these still abstract away constants and lower-order terms.

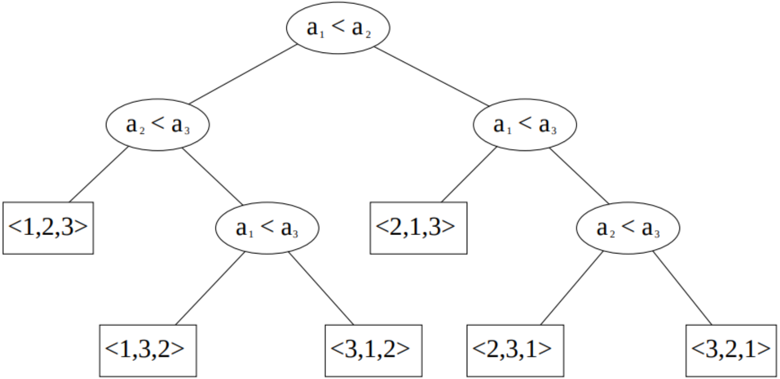
Lower Bound Example 1

- How do we know that taking the min of n elements needs to take at least $\Omega(n)$ time?
- We proceed on contradiction:
  - Assume we have an $min()$ algorithm which runs in $o(n)$
  - Then there must be some element we didn't observe...
  - Thus, adversarialy that element can be the minimum, so no such algorithm exists!
- So $min() \in \Omega(n)$

Lower Bound Example 2

- How do we know that sorting is $\Omega(n \log n)$?
- In this case, we change our model of computation to a model of *decision trees*
- The only operation we have available is to compare two elements and make new decisions.
- This is very different from how we program, or even think about Turing machines.

Lower Bound Example 2



Lower Bound Example 2

- There are $n!$ possible permutations, thus $n!$ possible leaves
- A tree of height h has at most 2^h leaves
- We find an h such that $2^h \geq n!$
- $\log\left(2^h\right) \geq \log(n!) \implies h \geq n\log(n)$
- Thus sorting n elements is $\Omega(n\log(n))$

P, NP, & More

- The first week we talked about P vs NP
- P are problems *decidable* in polynomial time
- NP are problems *decidable* in non-deterministic, polynomial time
    - You can view non-determinism as always-perfect guessing or infinite multithreading
    - The equivalent definition is problems whose solutions can be checked in polynomial time

P, NP, & More

- We know, via the Cook-Levin Theorem that Boolean Satisfiability is NP-Complete.
- This means it is representatively hard for all problems in NP
- These problems are also exponentially hard, i.e. runtimes on the order of $O(2^n)$.
- This is not very useful to try and compute for any reasonable size

P, NP, & More

- We don't know if these problems have lower bounds that are not exponential (P vs NP)
- If we have a hunch a problem is hard, how do we prove it?
- Cook-Levin is a given, but its proof is tedious and complex to replicate for other problem types.
- We can bypass this tediousness with reductions!

Reductions

- We can show a problem is *just as hard as* another problem by reducing a known hard problem to our target problem.
- Let A be our NP-Hard problem and B be our target problem.
- We're given an instance of A, e.g. if we had an algorithm for A, this would be the input.
  - Boolean Formula, Graph, etc.
- We take that instance of A and design an algorithm to convert it to an instance of B
- We this conversion to ensure that if we were to answer A one way, we answer B the same way.

Reductions

- Now given this conversion, we assume if we had a *fast* algorithm for B, then this could imply a *fast* algorithm for A
- As long as the runtime of our converter algorithm is polynomial (for NP-Hard reductions), this is a valid claim.
- Since A was NP-Hard, we now know that B must be NP-Hard

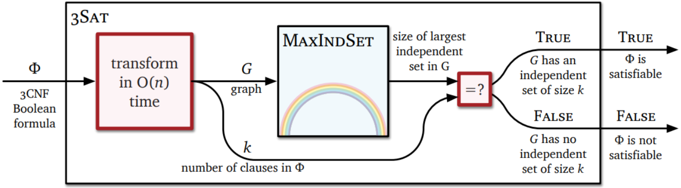


Figure 12.7. A polynomial-time reduction from 3SAT to MAXINDSET.

Looking at Fine Grained Complexity

- P vs NP is cool and all - but trodded ground
- It's unlikely anyone will make meaningful progress other than showing problems are hard
- But what about problems we know how to solve fast?
- How fast can we solve these problems?

Looking at Fine Grained Complexity

- FGC looks at complexity at a function level: Problems that are $O(n^2)$ and reducing between them.
- The techniques of reductions still apply, but now if we're trying to show something is $O(n^2)$-hard, our reduction can't be slower than $O(n^2)$
- $O(n^2)$ is just an example, reductions between problems in P is the idea.

Leetcode Haunts Us

- TwoSum: Given an array of n numbers, find 2 numbers that sum to t
- How do we solve it?
- Iterate through, storing $t - i$ in a BST or Hashmap. $O(n \log n)$ or $O(n)$ expected.

Leetcode Haunts Us

- ThreeSum: Given an array of n numbers, find 3 numbers that sum to 0
- How do we solve it?
- Iterate through, Calling TwoSum with the value at the given index as t, and the rest of the array
- $O(n)$ calls to TwoSum, $O(n^2)$.

3SUM

- Can we do better?
- We... don't know!
- Using the same decision tree analysis as we did with sorting, Kane, Lovett and Moran showed 3SUM has a $O(n \log^2(n))$ decision complexity
- No-one has been able to come up with a working sub-quadratic algorithm for the purely general case.

3SUM

- This constant searching has led to the ***3SUM Conjecture:***
  - There is no algorithm to solve 3SUM in $O(n^{2-\epsilon})$ for $\epsilon > 0$
- With this has led to a massive field of research, reductions, and discoveries

Geombase

- We turn to another problem, Geombase:

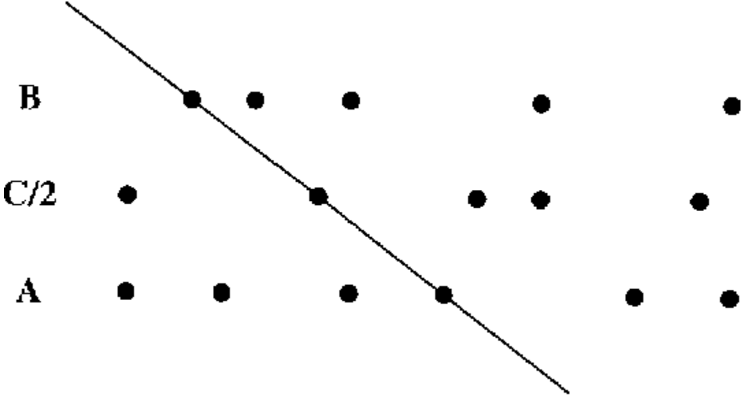


Fig. 1 An example of GEOMBASE

Geombase

- Given n points on 3 parallel lines, is there 3 points that are co-linear across the 3 lines?
- We can show that this problem is 3SUM hard!

3SUM to Geombase

- Given n points on 3 parallel lines, is there 3 points that are co-linear across the 3 lines?
- We can show that this problem is 3SUM hard!

3SUM to Geombase

- Equivalent 3SUM Problem: Given 3 arrays A, B, C find one element in each such that $a + b + c = 0$
- We take each element $a \in A$ and make a point $(2a, 0)$, $b \in B$, $(2b, 2)$, and $c \in C$, $(-c, 1)$
- We see that a solution is finding three points $(x_i, 0), (x_j, 1), (x_k, 2)$ where $x_i + x_k = 2x_j$
- With the points we made, $2a + 2b = -2c \implies a + b + c = 0$
- This reduction took $O(n)$ time, thus Geombase is $3SUM$-Hard.

Improvements to 3SUM

- People still try to find better algorithms, because they are tractable!
- In 2018 Timothy M Chan (Professor Here!) found a $O(n^2(\log\log n)^{O(1)}/log^2(n))$ solution to 3SUM
- With certain assumptions, 3SUM can be solved relatively fast as well (such as a bounded universe).

Other Conjectures

- All Pairs Shortest Path (APSP)
  - Find the shortest path between all pairs of vertices
  - We can reduce to matrix multiplication... $O(n^{\omega})$ for $\omega = 2.371522$
  - Is there a *combinatorial* algorithm $O(n^{3-\epsilon})$?
- Orthogonal Vectors (OV)
  - Given two sets of bitvectors of size d, find a pair for which $a \cdot b = \mathbf{0}$
  - $O(dn^2)$, $O(4^d + n)$ (when d is small).
  - Conjecture: No $O(n^{2-\epsilon})$ when $d >> \log(n)$

Connections to NP

- Orthogonal Vectors (OV) has a reduction from k-SAT!
  - Suppose a $O(d^{O(1)}n^{2-\epsilon})$ algorithm existed for OV...
  - Through some care, we can take a k-CNF Boolean Formula and turn it into two sets of size $2^{(n/2)}$.
  - Then we can solve k-SAT in $O^*(2^{(1-\epsilon/2)n})$ time!
  - $O^*(\cdot)$ hides polynomial factors.

Questions?