CS5112: Algorithms and Data Structures for Applications

Reductions, lower bounds, hashing

Ramin Zabih

Cornell University

CORNELL TECH

Administrivia

- Class average on the duck problem (reductions) was by far the lowest on the prelim
- This is core CS material
  - Dynamic programming, NP completeness, hashing
  - The other hard problem (MST) was not core
- A duck problem will be on the final exam
- Today we're going to go over this again
  - Bring your questions!

Lecture outline

- Reductions to prove hardness

- Lower bounds

- Example problems

- Universal and perfect hashing

- String search with hashing

- Distributed hash tables

Reduction key ideas

- Turning one problem into another is incredibly powerful
- Often to make a problem easy
  - PageRank, intelligent scissors, etc.
  - Most of the 'greatest hits' in CS
- Sometimes to show a problem is hard
  - NP-hardness or lower bounds
  - Utility is not quite as obvious but it's still important

Traveling Salesman Problem

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?
    - The decision version is whether there exists a route whose total distance is no greater than L.

- Seems awfully similar to Hamiltonian cycle
    - They're both graph problems
    - They're both looking for cycles
    - Biggest difference seems to be the edge weights
- Idea: we know Hamiltonian cycle is NP-complete. If we can reduce Hamiltonian cycle to Traveling Salesman Problem, we'll know TSP is NP-complete as well.
    - Intuitively: we know Hamiltonian cycle is hard. If it's easy to take the solution to TSP and determine a solution for Hamiltonian, that must mean TSP is hard too.

Example reduction

- Suppose we can efficiently solve the duck problem by using the rabbit problem as our key subroutine

- What follows from this?
  - What if we knew that the duck problem was NP hard?
 - What if we knew that the duck problem could not be solved in polynomial time?

Going in the other direction

- Suppose we knew that there was a polynomial time duck problem algorithm.
  - We cam do this if we had a polynomial time rabbit algorithm
- What does this tell us about the rabbit problem?
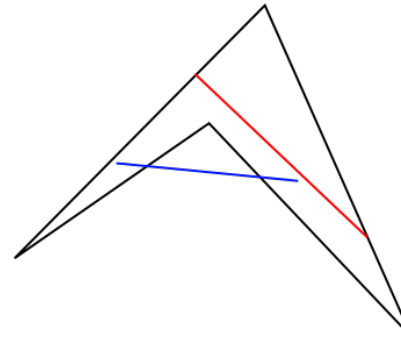- Can you write a sorting algorithm that calls SAT? Or TSP?

Example problems (reductions)

- Assume rabbit is in P. Show duck is in P.

- Assume rabbit is NP-hard. Show duck is NP-hard.

- Assume rabbit is NP-complete. Show duck is NP-complete.
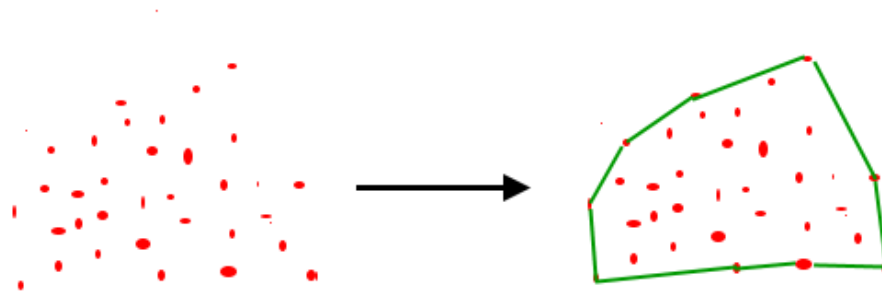
# Convex hull problem
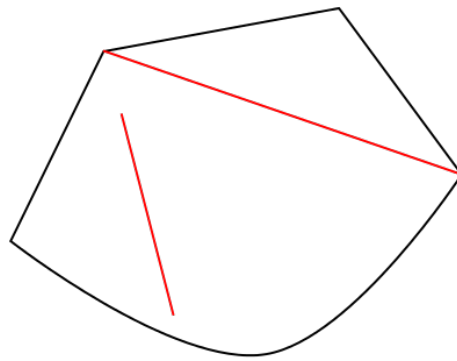
convex

not convex

Convex hull lower bound

- Definition: the convex hull of a set of points is the smallest convex set containing those points
  - More precisely, the algorithm contains the points on the hull in a specific order (assume counter-clockwise)
- How fast can we compute the convex hull?
- We can use convex hull to sort!
  - Suppose you want to sort the positive numbers $\{x_i\}$
  - Compute the convex hull of the points $\{(x_i, x_i^2)\}$
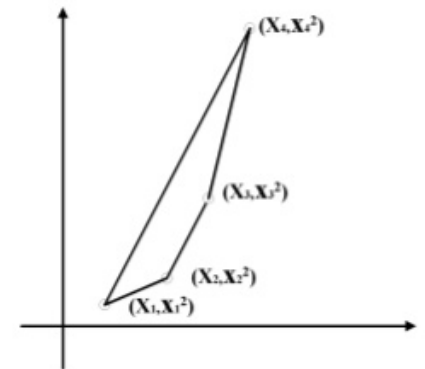
[Figure source](#)

Example problems (lower bounds)

- In the following, what do we know about the duck problem's lower bound, if anything?
  - Duck1 can efficiently solve sorting.
  - Sorting can be used to solve duck2.
  - Duck3 is solvable in polynomial time.
  - Duck4 is solvable in exponential time.
  - Duck5 is NP-hard.

CORNELL
TECH

Universal and perfect hashing

- Back to hashing

- We talked about perfect hashing (no collisions)
  - Minimal: table is exactly full

- Main application is with fixed input data
  - Example: compilers

- How do we generate perfect hash functions for our data?
  - Give up on minimality

Universal hashing

- We can randomly generate (pick) a hash function h
  - This is NOT the same as the hash function being random
  - Hash function is deterministic!
  - Can re-do this if it turns out to have lots of collisions
- Assume input keys of fixed size (e.g., 32 bit numbers)
- Ideally h will spread out the keys uniformly

$$P[h(x) = h(y) \mid x \neq y] \leq \frac{1}{2^{32}}$$

  - Think of this as fixing $x, y \mid x \neq y$ and then picking h randomly
- If we had such an h, the expected number of collisions when we hash N numbers is $\frac{N}{2^{32}}$

Universal hashing by matrix multiplication

- This would be of merely theoretical interest if we could not generate such an h

- There's a simple technique, not efficient enough to be practical
  - More practical versions follow the same idea

- Now assume the inputs/outputs are 4 bit numbers/3 bit numbers respectively, i.e. inputs: 0-15, outputs: 0-7

- We will randomly generate a 3x4 array of bits, and hash by 'multiplying' the input by this array

CORNELL TECH

Universal hashing example

- We multiply using AND, and we add using parity
  - Technically this is mod 2 arithmetic

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Balls into bins

- There is an important underlying idea here
  - Shows up surprisingly often
- Suppose we throw m balls into n bins
  - Where for each ball we pick a bin at random
  - How big should n be so that with probability > ½ there are no collisions?
  - This is the opposite of the birthday paradox
- Answer: need $n \approx m^2$
- So to avoid collisions with probability ½ we need our hash table to be about the square of the number of elements

CORNELL TECH

Perfect hashing from universal hashing

- We can use this to create a perfect hash function

- Generate a random hash function h
  - Technically, from a universal family (like binary matrices)

- Use a "big enough" hash table, from before
  - I.e., size is square of the number of elements

- Then the chance of a collision is < ½

- In expectation we do this twice to get a perfect hash function