CS5112: Algorithms and Data Structures for Applications

Reductions, lower bounds, hashing

Ramin Zabih

Some content from: Wikipedia;
J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

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?

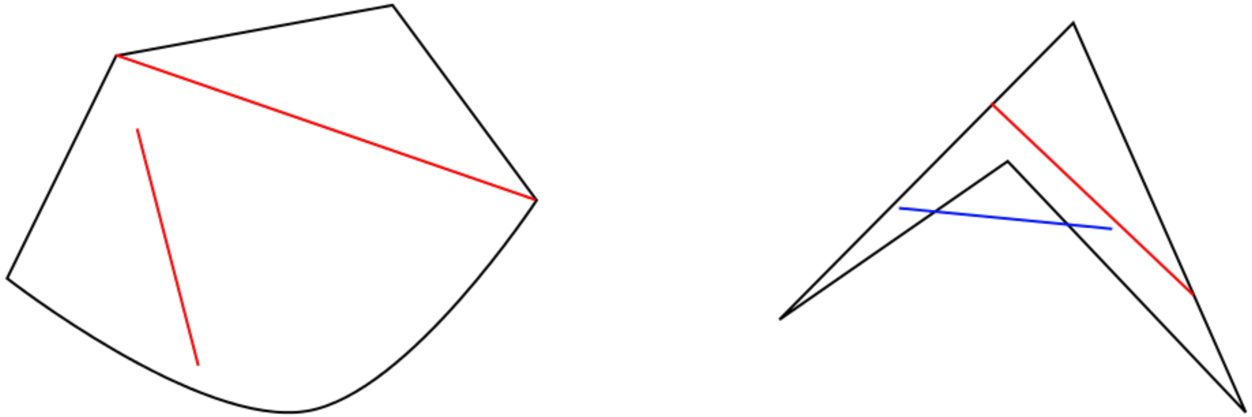
Going in the other direction

- 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

Figure source

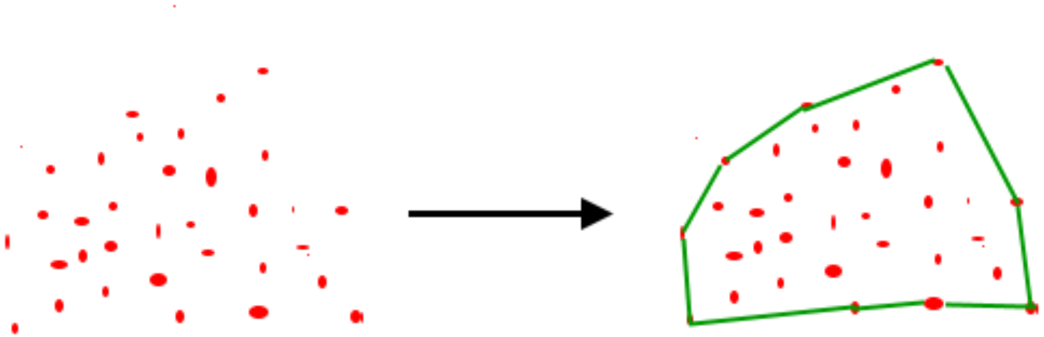


Figure source

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 $\{(x_i, x_i^2)\}$

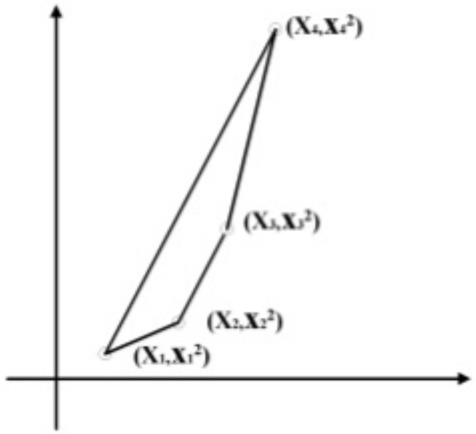


Figure source

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.

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

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

  - Think of this as fixing $x, y | x \neq y$ 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

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

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

Rabin-Karp string search

- Find one string (“pattern”) in another
  - Naively we repeatedly shift the pattern
  - Example: To find “greg” in “richardandgreg” we compare greg against “rich”, “icha”, “char”, etc. (‘shingles’ at the word level)
- Instead let’s use a hash function h
- We first compare h(“greg”) with h(“rich”), then h(“icha”), etc.
- Only if the hash values are equal do we look at the string
  - Because $x = y \Rightarrow h(x) = h(y)$ (but not $\Leftarrow$ of course!)

Rolling hash functions

- To make this computationally efficient we need a special kind of hash function h
- As we go through “richardandgreg” looking for “greg” we will be computing h on consecutive strings of the same length
- There are clever ways to do this, but to get the flavor of them here is a naïve way that mostly works
  - Take the ASCII values of all the characters and multiply them
  - Reduce this modulo something reasonable

Distributed hash tables (DHT)

- BitTorrent, etc.
- Given a file name and its data, store/retrieve it in a network
- Compute the hash of the file name
- This maps to a particular server, which holds the file
- Sounds good! Until the file you want is on a machine that is not responding…
  - But is this a real issue? Aren't computers pretty reliable?

Google datacenter numbers (2008)

- *In each cluster's first year, it's typical that:*
  - *1,000 individual machine failures will occur;*
  - *thousands of hard drive failures will occur;*
  - *one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours;*
  - *20 racks will fail, each time causing 40 to 80 machines to vanish from the network;*
  - *5 racks will "go wonky," with half their network packets missing in action;*
  - *The cluster will have to be rewired once, affecting 5 percent of the machines at any given moment over a 2-day span.*
  - *About a 50 percent chance that the cluster will overheat, taking down most of the servers in less than 5 minutes and taking 1 to 2 days to recover.*
- Jeff Dean, “Google spotlights data center inner workings”, CNET May 2008

From filename to processor

- Typically the result of a hash function is a large number
  - SHA-1 produces 160 bits (not secure!)
- Map into servers with modular arithmetic
  - Reminder: 4 + 7 = 1 (mod 10)
  - Mod with powers of 2 is just the low-order bits
- How do we handle a server crashing or rejoining??

Consistent hashing

- Effectively the hash table itself is resized
  - Note that this is an important operation in general!
- With naïve hash functions, resizing is a disaster
  - Everything needs to be shuffled between buckets/servers
- Key idea is to give add **state**
  - Traditional hash functions are stateless/functional

Hashing into the circle

- Let’s convert the output of our hash function into a circle
  - – For example, using the low-order 8 bits of SHA-1
- We map both servers and data onto the circle
  - – For a server, hash of IP address or something similar
- Data is stored in the “next” server on the circle
  - – By convention we will move clockwise

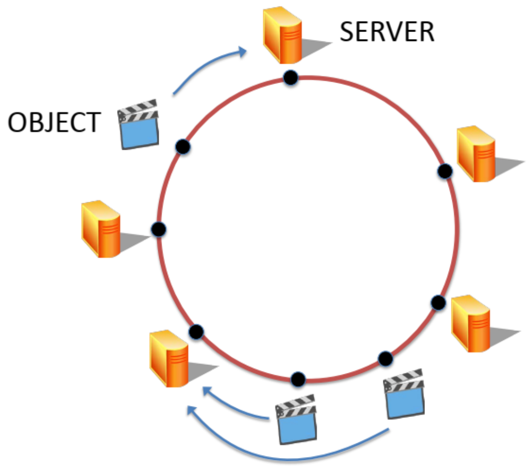


Figure from Maggs, Bruce M.; Sitaraman, Ramesh K. (July 2015), "Algorithmic nuggets in content delivery" (PDF), *SIGCOMM Computer Communication Review*, New York, NY, USA,**45** (3): 52–66

Example of consistent hashing

- Data 1,2,3,4 stored on computers A,B
- Servers->data (good quiz/exam question):

  A->1,4

  B->2

  C->3
- If C crashes, we just move 3 to A

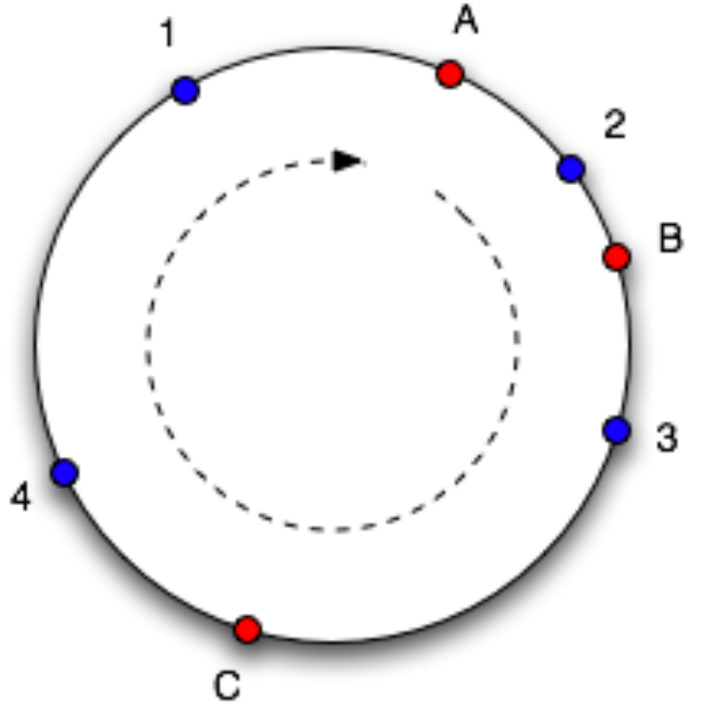


Diagram taken from Tom White based on original article

Gracefully adding/removing a server

- Add server D after C crashes
  - Takes 3,4 from A
- Servers->data:

  A->1

  B->2

  D->3,4
- This is a lot faster!
  - Naively, going from 3 to 4 servers moves 75% of data
  - With consistent hashing we move 25% of data
  - Advantage gets even larger for more servers

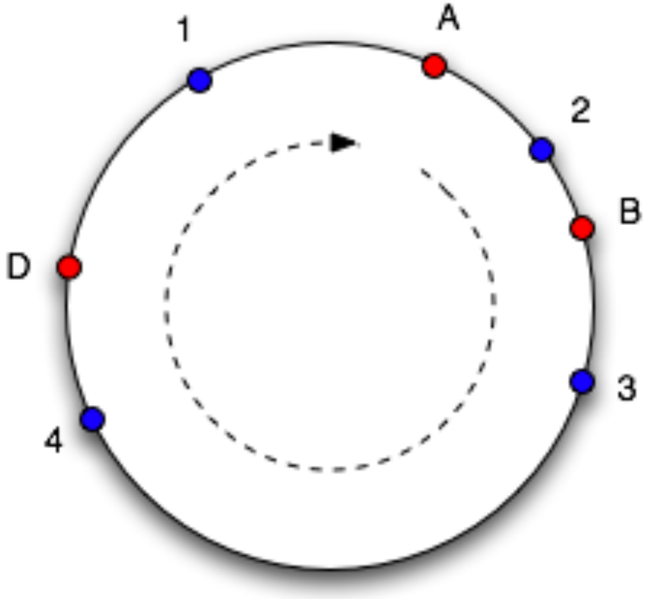


Diagram taken from Tom White based on original article

Improving consistent hashing

- Need a uniform hash function, lots of them aren’t
- Typically make replicas of servers for load balancing
  - About $\log m$ replicas from m servers for theoretical reasons
  - Can also replicate data items if they are popular
- Typically store a list of nearby nodes for redundancy
- Note that the data still needs to move after a crash
- Store the servers in a BST to efficiently find successor
  - This requires global knowledge about the servers

Handling popular objects

- Each object can have its own hash function
- Basically, it’s view of the unit circle
- Ensures that you are very unlikely to have 2 popular objects share the same server