COMP 3761: Algorithm Analysis and Design

Shidong Shan

BCIT

Limitations of Algorithm Power

- Lower bounds arguments
  1. Trivial lower bounds
  2. Information-theoretic arguments
  3. Adversary arguments
  4. Problem reduction
- Decision trees
- P, NP, and NP-complete problems

Limitations of algorithm power

- The power of algorithms is **not unlimited**
- Some problems cannot be solved by any known algorithm
- Other problems can be solved algorithmically but not in polynomial time
- For problems can be solved in polynomial time by some algorithms, they are usually lower bounds on their efficiency.

Lower bounds

- **lower bounds**: an estimate of minimum amount of work needed to solve a problem
- Examples:
  - number of comparisons needed to find the largest element in a set of n numbers
  - number of comparisons needed to sort an array of size n
  - number of comparisons necessary for searching in a sorted array
  - number of multiplications needed to multiply two n-by-n matrices

Comparisons of algorithms

- Asymptotic efficiency class gives some information about the algorithms
- For example: Selection sort: $O(n^2)$
  Algorithm for the Tower of Hanoi $O(2^n)$
- It may appear that selection sort is much faster than the algorithm for the Tower of Hanoi.
- But the two algorithms are for two significant different problems!

Alternative comparison of algorithms

- How efficient is a **particular** algorithm compared with other algorithms for the same problem?
- What is the **best** possible efficiency that **any** algorithm solving the problem may have?
- e.g., compare selection sort $O(n^2)$ with heapsort $O(n \log n)$.

Lower bounds

- Lower bound can be an exact count or an efficiency class (Ω)
- **Tight** lower bound: there exists a known algorithm with the same efficiency as the lower bound
- Lower bounds can tell how much improvement we can hope for a better algorithm for the given problem.

Problem	Lower bound	Tightness
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n \log n)$	yes
n-digit integer multiplication	$\Omega(n)$	unknown
multiplication of n-by-n matrices	$\Omega(n^2)$	unknown

Establishing lower bounds

- relatively easy to determine the efficiency of a **particular** algorithm
- difficult to establish a limit on the efficiency of **any** algorithm, known or unknown
- very difficult to obtain a nontrivial lower bound, even for a simple-sounding problem

Common Methods of Lower Bound Arguments

Lower-bound arguments:

1. trivial lower bounds
2. information-theoretic arguments
3. adversary arguments
4. problem reduction

Trivial lower bounds

- based on counting the number of items that must be processed in input and generated as output
- Any algorithm must at least "read" all the input and "write" all its output.
- Examples:
  1. finding max element
  2. polynomial evaluation
  3. sorting
  4. element uniqueness
- Comments:
  - trivial lower bounds may or may not be useful

Information theoretic lower bounds

- Establish lower bounds based on the amount of information it has to produce
- information theoretic lower bounds are very useful for many comparison-based problems
- For examples, sorting and searching
- Use decision trees

Example of information-theoretic lower bounds

Game: Guessing a number between 1 and n with yes/no questions.

Question: What is the minimum number of questions that any algorithm can determine the output in the worst case?

- The number of bits needed to specify a particular number between 1 and n is $\lceil \log_2 n \rceil$
- An answer to each question yields at most one bit of information about the selected number
- Any algorithm solving this problem has to take at least $\lceil \log_2 n \rceil$ in the worst case.

Adversary arguments

- Prove a lower bound by playing the role of adversary
- Adversary can adjust the input values to force algorithm to work the hardest
- Adversary must stay consistent with the choices already made
- Measure the amount of work needed to shrink a set of potential inputs to a single input along the most time-consuming path

Examples of Adversary Strategy

Example 1: Guessing a number between 1 and n with yes/no questions.

- consider each number between 1 and n as being potentially selected
- Strategy: after each question, give an answer with largest set of numbers consistent with all previously given answers
- The strategy leaves the adversary at least one half of the numbers he had before his last answer
- If an algorithm stops before the size of the set is reduced to one, the adversary can exhibit a legitimate input number that an algorithm failed to identify
- Lower bound: at least $\lceil \log_2 n \rceil$ steps needed to shrink an n-element set to a one-element set.

Examples of Adversary Strategy

Example 2: Merging two sorted lists of size n

$$a_1 < a_2 < \ldots < a_n \quad \text{and} \quad b_1 < b_2 < \ldots < b_n.$$

- The number of key comparisons for merging in the worst case is $2n - 1$ (mergesort)
- Question: Is there any comparison-based algorithm that can do merging faster?
- Adversary strategy: answer true to $a_i < b_j$ if and only if $i < j$

1. force any algorithm to produce the only combined list

$$b_1 < a_1 < b_2 < a_2 < \ldots < b_n < a_n$$

2. any correct algorithm will have to explicitly make $2n - 1$ comparisons of adjacent elements of the combined list
3. if one of these comparisons has not been made, transpose these keys to get a different order, which is consistent with all the comparisons made but incorrect to the given list configuration.

Problem reduction

- Idea: if problem P is at least as hard as problem Q, then a lower bound for Q is also a lower bound for P

- Find problem Q with a known lower bound that can be reduced to problem P in question
- Any algorithm solving P would solve Q, so a lower bound for Q will be a lower bound for P
- Reduce problem Q with a known lower bound to problem P.

Problem reduction example

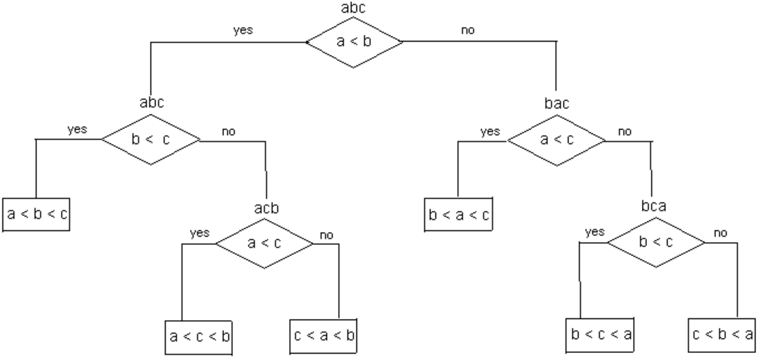
- Euclidean MST problem (Problem P): Given n points in the Cartesian plane, construct a tree of minimum total length whose vertices are the given points.
- Element uniqueness problem (Problem Q): determine whether there are duplicates among n given numbers, the known lower bound is $\Omega(n \log n)$
- Transform any set $x_1, x_2, \ldots, x_n$ of n real numbers into a set of n points in the Cartesian plan by simply adding 0 as the points' y coordinates: $(x_1, 0), (x_2, 0), \ldots, (x_n, 0)$
- Let T be an Euclidean MST for this set of points
- Checking whether T contains a zero-length edge will answer the question about uniqueness of the given numbers.
- Thus, $\Omega(n \log n)$ is a lower bound for the Euclidean MST problem.

Decision tree

- A convenient model of algorithms involving comparisons:
  internal nodes represent comparisons
  leaves represent outcomes
- Binary tree with ℓ leaves and height h: $h \geq \lceil \log_2 \ell \rceil$
- The largest number of leaves in a binary tree is 2^h, $\ell \leq 2^h$
- Ternary tree: An ordered tree in which each node has at most three children
- The height h of ternary trees with ℓ leaves: $h \geq \lceil \log_3 \ell \rceil$.

Decision tree example

Decision tree for the 3-element insertion sort



- Number of leaves: $3! = 6$
- Minimum height of the tree is $\lceil \log_2 6 \rceil = 3$.

Decision trees for comparison-based algorithms

- Lower bound of the worst case number of comparisons made by any comparison-based algorithm is the lower bound on the heights of its binary decision trees.
- Number of leaves $\geq$ the number of possible outcomes
- Note: same outcome may be arrived at through a different chain of comparisons
- An algorithm's work on a particular input can be traced by a path from the root to a leaf in its decision tree
- The number of comparisons made by the algorithm on an input is the number of edges in its path.

Decision trees for sorting algorithms

- Number of outcomes is the number of permutations of the n element set: $n!$
- Number of tree leaves $\geq n!$
- Height of binary tree with $n!$ leaves: $h \geq \lceil \log_2 n! \rceil$
- Sterling's formula: $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$ as $n \to \infty$, then $\lceil \log_2 n! \rceil \approx n \log_2 n$
- Minimum number of comparisons in the worst case $\geq \lceil n \log_2 n \rceil$
- This lower bound is tight (Mergesort)

Ternary decision trees for Binary Search

- Binary search: searching a K in a sorted array $A[0..n-1]$
- Use three-way comparisons: $k < A[i], \quad K = A[i], \quad K > A[i]$
- Can represent the algorithm with a ternary decision tree
- Example: Draw a ternary decision tree for binary search in a four-element array $A[0..3]$.

Ternary decision trees for Binary Search

- Ternary decision tree for binary search has $2n+1$ leaves
- Possible outcomes:
  n for successful searches and $n+1$ for unsuccessful ones
- Height $h \geq \lceil \log_3(2n+1) \rceil$
- Binary Search Algorithm: number of comparisons in the worst case is $\lceil \log_2(n+1) \rceil$
- $\lceil \log_3(2n+1) \rceil \leq \lceil \log_2(n+1) \rceil$ for every positive integer n
- Questions:
  Can we prove a better lower bound?
  Is binary search optimal?

Binary decision trees for searching a sorted array

Use binary decision tree:

- Internal nodes correspond to three-way comparisons as in ternary decision trees
- Internal nodes also serve as terminal nodes for successful searches
- Leaves represent only unsuccessful searches with a total of $n+1$
- The height of binary decision tree $h \geq \lceil \log_2(n+1) \rceil$
- This lower bound is tight (binary search)

Classifying problem complexity

- An algorithm solves a problem in **polynomial** time if its worst-case time efficiency is $O(n^k)$ for an input size of n.
- **tractable** problems: **can** be solved in polynomial time
- **intractable** problems: **cannot** be solved in polynomial time; cannot solve large instances in a reasonable amount of time
- **Computational complexity**: seeks to classify problems according to their inherent difficulty
- A problem's intractability remains the same for all principle models of computations and all reasonable input-encoding schemes

Is a problem tractable?

Possible answers:

- Yes (give examples)
- No, because:
  1. It's been proved that no algorithm exists at all
  2. It's been proved that any algorithm takes exponential time to solve it
     Example: generating all the subsets of a n element set ($T(n) = \Omega(2^n)$)
  3. It is an **undecidable problem**: the problem cannot be solved by any algorithm
     Example: Alan Turing's halting problem
     Given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.
- Unknown

Problem types

- **Optimization problem**: find a solution that maximizes or minimizes some objective function
- **Decision problem**: answer yes/no to a question
- Many problems have decision and optimization versions.
- Decision problems are more convenient for formal investigation of their complexity

Class P: Polynomial

- A class of decision problems that can be solved in polynomial time
- There exists deterministic algorithms to solve the problems.
- Examples:
  - searching
  - sorting
  - element uniqueness
  - graph connectivity
  - graph acyclicity

Nondeterministic Polynomial Algorithm

Definition: A **nondeterministic polynomial algorithm** is a two-stage procedure

1. The non-deterministic (guessing) stage:
   - generate a random string s as the "proposed solution"
   - each time the algorithm is run the string may differ
2. The deterministic (verification) stage:
   A deterministic algorithm takes the input of the problem and the proposed solution, and it returns value true or false in polynomial time

Class NP

- Class of decision problems whose proposed solutions can be verified in polynomial time
- A given proposed solution for a given input can be checked in polynomial time to see if it really is a solution.
- NP problems can be solved by nondeterministic polynomial algorithms.

Class NP Problems

Many important problems belong to class NP:

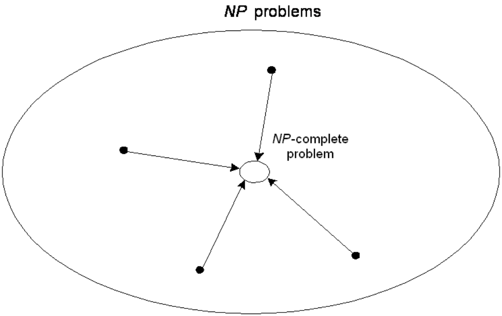
- Decision versions of TSP, knapsack problem, and many other combinatorial optimization problems.
- With a few exceptions:
  e.g, MST and shortest paths can be solved in polynomial time.
- Partition problem: Is it possible to partition a set of n integers into two disjoint subsets with the same sum?

Theorem

- **Theorem**: $P \subseteq NP$
- All the problems in P can also be solved in the verification-stage of a nondeterministic algorithm
- Simply ignore the nondeterministic guessing stage.
- Big question: Is $P = NP$?
- $P = NP$ implies that each of those difficult combinatorial decision problems can be solved by a polynomial time algorithm!

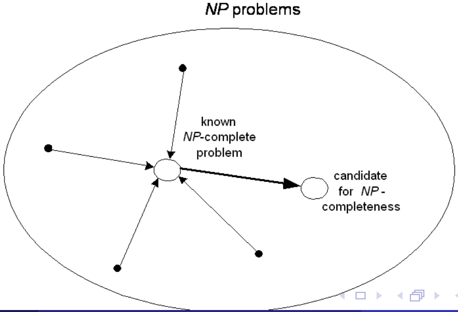
NP-Complete problems

- A decision problem D is NP-complete if it's as hard as any problem in NP
- i.e. it is in NP and it is NP-hard.
- A problem D is NP-hard if every problem in NP is polynomially reducible to D.



Problem Reduction

- Problem reduction is often used to classify problems according to their complexity.
- Cook's theorem (1971): CNF-sat is NP-complete.
- Other *NP*-complete problems obtained through polynomial-time reductions from a known *NP*-complete problem
- Well-known *NP*-Complete problems: TSP, knapsack, partition, graph-coloring and many others.



$P = NP$? Dilemma

- $P = NP$ would imply that every problem in NP, including all NP-complete problems, could be solved in polynomial time
- If a polynomial-time algorithm for just one NP-complete problem is discovered, then every problem in NP can be solved in polynomial time, i.e., $P = NP$
- Most but not all researchers believe that $P \neq NP$, i.e. P is a proper subset of NP ($P \subset NP$)