Lecture Note 5

CSCI 6470 Algorithms (Fall 2024)

Liming Cai

School of Computing UGA

October 31, 2024

Chapter 5. Lower bound and Intractability

Topics to be discussed:

- Time complexity Lower bound
- Exhaustive search for intractable problems
- NP-completeness theory

1. Time complexity lower bound

- Discussions have been focused on

  (1) "at most" (upper bound) time usage by algorithms;

  (2) "sufficient" (upper bound) time to solve a problem,

  Both are denoted with big-O: $T(n) = O(f(n)) \Leftrightarrow T(n) \leq cf(n)$

  e.g.,

  (1) `Mergesort` uses $O(n \log_2 n)$ time; **so**

  (2) Time $O(n \log_2 n)$ is sufficient to solve **Sorting** problem;

- We are now interested in

  (1) "at least" (lower bound) time usage by algorithms;

  (2) "necessary" (lower bound) time to solve a problem,

1. Time complexity lower bound

Lower bounds of algorithms

- `Mergesort` runs at most $cn \log_2 n$ time on the worst case instances or simply with $O(n \log n)$ time;
- `Mergesort` runs **at least** $cn \log n$ time on the worst case instances, for all $n \geq n_0$ for some constants $c > 0$ and $n_0 \geq 0$

  Proof: (Taken-home exercise)

  Called time complexity **lower bound** of `Mergesort`, denoted with $\Omega(n \log_2 n)$
- Definition of big-Ω:

$$T(n) = \Omega(f(n)) \Leftrightarrow T(n) \geq cf(n)$$

holds for all $n \geq n_0$ for some $c > 0$ and $n_0 \geq 0$.

1. Time complexity lower bound

Lower bounds of algorithms

- More big-Ω results:

  `Insertion-Sort` and `Selection-Sort` have complexity $\Omega(n^2)$;

  `Binary Search` has time complexity $\Omega(\log_2 n)$;

  `Recursive-Fib` has time complexity $\Omega(\sqrt{2}^n)$;

- $\Theta \Leftrightarrow O + \Omega$;

  exact bound $=$ both upper bound and lower bound ;

  e.g., `Binary Search` has time complexity $\Theta(\log_2 n)$;

  What about `Recursive-Fib` ?

We have so far only discussed time lower bound for **for algorithms**!

1. Time complexity lower bound

Lower bounds of problems

Def: a **lower bound** for a problem is the least (i.e., necessary) amount of time required to solve the worst cases of the problem.

Why lower bound is interesting:

- Fastest algorithms for **Sorting** have time $O(n \log_2 n)$, e.g., `Mergesort`;

  Question: is there faster algorithm, say of $O(n)$, for **Sorting**?

Can we derive a lower bound of Π from time complexity of A?

- Let Π be a problem and A be an algorithm solving Π in time $T(n)$.

  (1) If $T(n) = O(n^2)$, is $\Omega(n^2)$ a lower bound for Π?
  (2) If $T(n) = \Omega(n^2)$, is $\Omega(n^2)$ a lower bound for Π?

  consider Π is **Sorting**, and A is `Insertion Sort`

1. Time complexity lower bound

Lower bounds of problems (cont.)

- Conclusion: Algorithms do not help prove lower bounds of problems;
- Instead, need to look into the problem itself;

  Ideally, the problem can be modeled in a certain way to reveal the necessity of certain amount of time complexity;
- Example 1: **Finding max** can be solved using $n-1$ comparisons on n elements;

  (how to achieve this?)

  Does **Finding max** require $n-1$ comparisons in the worst case?

1. Time complexity lower bound

Example 1: **Finding max**

Observation 1: to produce the max element, elements participate in element-element comparisons;

Observation 2. some comparisons can be directed or indirected, but all elements have to participate in comparisons;

$\Rightarrow$ Any algorithm for **Finding max** is *modeled as a graph* on the n elements, where

(1) comparisons as edges;
(2) graph is connected;
(3) the least number of comparisons = least number of edges;

Fact: Least number of edges in a connected graph is at least $n-1$.

Theorem 1: Problem **Finding max** requires $n-1$ element-element comparisons.

1. Time complexity lower bound

Example 2: **Sorting**

- Any algorithm for **Sorting** is modeled as a decision tree; where

  (1) Nodes are comparison between two elements;

  (2) Every parent-child edge is an outcome of the comparison represented by the parent;

  (3) Every path from the root to a leaf is some sorting computation;

  (4) Every leaf represents a sorting outcome (based on the input);

  (5) The longest path from root to a leaf is the worst case time;

  (6) The lower bound question for **Sorting** essentially becomes to know the minimum height of such a tree;

Note: The tree model is an arbitrary tree representing an arbitrary algorithm; we should not assume any specific ordering of comparison nodes in the tree.

1. Time complexity lower bound

Example 2: **Sorting**

- The height h of a binary tree is related to its number l of leaves as: $h \geq \log_2 l$;
- The algorithm needs to accommodate all possible scenarios of the input with different sort outcomes;
  (1) The number of scenarios of the input is $n!$ on n elements (of distinct values)
  (2) The number of outcomes of the algorithm $\geq n!$, accordingly;
- Number of leaves $\geq n!$; and height $\geq \log_2(n!) = \Omega(n \log_2 n)$;

Theorem 2: **Sorting** requires $\Omega(n \log_2 n)$ element-element comparisons.

2. Intractable problems and exhaustive search

Def. If a problem admits algorithms of time $O(n^c)$ for some constant c, it is *tractable*; otherwise, it is *intractable*.

- We have investigated quite a few tractable problems: **Sorting, Shortest paths, SCC, Edit Distance, Fractional Knapsack, MST**, etc.

- Many other problems are likely intractable: **TSP, SAT, Max Independent Set, Knapsack**,

Usually for intractable problems, in order to find exact (optimal, accurate) solutions, exhaustive search is resorted to.

But not all exhaustive searches are naïve.

2. Intractable problems and exhaustive search

How to do an exhaustive search?

Enumerate all potential solutions then check each solution to identify the optimal/correct one.

- Most problems have solutions that can be easily encoded and enumerated in certain order.
- But non-naïve exhaustive searches are often based on additional insights to the problem.

Some intractable problems:

- **TSP**
  Input: an edge-weighted graph $G(V, E)$;
  Output: a circular path on G of the minimum total edge weight.

- **SAT**
  Input: a boolean $\phi(x_1, \ldots, x_n)$ with boolean variables x_i;
  Output: "yes" if and only if there is a truth value assignment to variables x_i's to satisfy ϕ.

- **Max Ind Set**
  Input: A graph $G = (V, E)$;
  Output: a subset of $I \subseteq V$, such that for all $u, v \in I$, $(u, v) \notin E$; and cardinality $|I|$ is the maximum.

  I is called an *independent set*.

2. Intractable problems and exhaustive search

(1) Naïve exhaustive search:

- encoding of solutions, e.g.,

  TSP: a circular path is encoded as a permutation of n vertices;

  SAT: an assignment is encoded as an n-bits binary string;

  Max Ind Set: an independent set can be encoded as ?

- Checking

  Routinely applying every encoded solution;

  Checking if the solution is correct/optimal;

In classroom exercise
write a recursive exhaustive algorithm for **SAT**.

Taken-home exercise
write a recursive exhaustive algorithm for **Max Ind Set**.

2. Intractable problems and exhaustive search

(2) Non-naïve exhaustive search:

- Take advantage of some characteristics of the problem;
- Search through the solution space in a savvy way;
- Derive a non-trivial time complexity (still exponential though);

A non-naïve exhaustive (recursive) algorithm for **Max Ind Set**:

- Consider any vertex u, there are two options (try both)
  (a) discard u; then find m.i.s. on reduced graph $G - \{u\}$;
  (b) choose u into I; then continue to include more vertices to I from graph $G - \{u\} - N(u)$, where $N(u)$ are neighbors of u;
- Time complexity derivation, assume $m = \min_u |N(u)|$

$$T(n) = T(n-1) + T(n-1-m)$$

2. Intractable problems and exhaustive search

Time complexity derivation, assume $m = \min_u |N(u)|$

$$T(n) = \begin{cases} T(n-1) + T(n-1-m) + O(n) & n \geq 1 \\ c & n = 0 \end{cases}$$

- $m = 0$, i.e., there are some "isolated vertices";

  $$T(n) = T(n-1) + T(n-1) = 2T(n-1) \Longrightarrow T(n) = \Theta(2^n)$$

  same as the naïve exhaustive search!

- $m = 1$, after removing all isolated vertices, how?

  $$T(n) = T(n-1) + T(n-2) \Longrightarrow T(n) = O(1.62^n)$$

- Can we do better? removing all vertices of degree 1, how?, if we can, then

  $$T(n) = T(n-1) + T(n-3) \Longrightarrow T(n) = O(1.5^n), \text{ why?}$$

Taken-home exercises V(A)

1. What is a complexity lower bound for an algorithm? What is a complexity lower bound for a problem? Are they related in anyway?
2. Understand the definitions of big-Ω and big-Θ. Prove that `Insertion Sort` has time complexity $\Theta(n^2)$.
3. Can we say **Find Max** uses exactly $n-1$ element-element comparisons on the worst cases? Explain.
4. Can we say **Sorting** has time complexity $\Theta(n \log_2 n)$? Explain.
5. Use the decision tree model to prove $\Omega(n \log_2 n)$ lower bound for **Sorting** on the required number of element-element comparisons. Write the proof following the logic given in the class but using your own language.
6. Use the decision tree modeling of algorithms to prove that searching for a given key from any input list of n elements requires at least $\log_2 n$ element-element comparisons.
7. Use the decision tree modeling of algorithms to prove that searching for a given key from any input list of n elements requires at least $\log_2 n$ element-element comparisons.

Taken-home exercises V(A) cont.

8. Let **Bit-Sorting** be the problem to sort an input of n binary bits (yes, each element is a single binary bit 0 or 1). What is the least number of bit-bit comparisons required by **Bit-Sorting**? Explain.
9. Write a naïve exhaustive search recursive algorithm for **SAT**.
10. Write a naïve exhaustive search non-recursive algorithm for **Max Ind Set**.
11. Write the non-a naïve exhaustive search algorithm for **Max Ind Set** that has been discussed in the class. Write it as a recursive algorithm.
12. Show that if problem **Max Ind Set** can be solved in time $O(n^c)$ for some constant c, so can problem **Ind Set**.
13. Show that if problem **Ind Set** can be solved in time $O(n^c)$ for some constant c, then problem **Max Ind Set** can be solved in time $O(n^{c+1})$.