Comparing Problems

Remember the concepts of Problem, Algorithm, and Program.

We've gotten pretty good at comparing algorithms.
How do we compare problems?

- Sorted Array Search
- Sorting
- Integer Factorization
- Integer Multiplication
- Selection
- Maximum Matching
- Minimum Vertex Cover

Computational Complexity

The **difficulty of a problem** is the worst-case cost of the **best possible algorithm** that solves that problem.

Computational complexity is the study and classification of problems according to their inherent difficulty.

Why study this?

- Want to know when an algorithm is as good as possible.
- Sometimes we want problems to be difficult!

How to compare problems

Big-O, big-Θ, and big-Ω are used to compare two functions.

How can we compare two problems?

Example: Sorting vs. Selection

- Forget about any specific algorithms for these problems.
- Instead, develop algorithms to **solve one problem** by using **any algorithm for the other problem**.
- Solving selection using a sorting algorithm:
- Solving sorting using a selection algorithm:
- Conclusion?

Defining tractable and intractable

Cobham-Edmonds thesis:
A problem is tractable only if it can be solved in polynomial time.

What can we say about intractable problems?

- Maybe they're undecidable (e.g., the halting problem)
- Maybe they just seem impossible (e.g., regexp equivalence)
- But not always! (e.g., integer factorization)

Million-dollar question:
Can any problems be verified quickly but not solved quickly?

Fair comparisons: Machine models

Proving lower bounds on problems requires a careful model of computation.

Candidates:

- Turing machine
- Clock cycles on your phone
- MIPS instructions
- "Primitive operations"

Theorem
These models are all polynomial-time equivalent.

Fair comparisons: Bit-length

Input size is our measure of difficulty (n).
It must be measured the same between different problems!

Past examples:

- Factorization $\Theta(\sqrt{n})$ vs. HeapSort $\Theta(n \log n)$
- Karatsuba's $\Theta(n^{1.59})$ vs. Strassen's $\Theta(n^{2.81})$
- Dijkstra's $\Theta(n^2)$ vs Dijkstra's $\Theta((n+m)\log n)$

Only measure for this unit: **length in bits of the input**

Fair comparisons: Decision problems

What about the size of the output? We'll consider **only**:

Definition: Decision Problems
Problems whose output is `YES` or `NO`

Is this a big restriction?

- Selection
- EI Scheduling
- Integer factorization
- Minimum vertex cover

Decision problem comparison

Compare regular factorization with decision problem version:

1. Given **instance** (N, k) of decision problem,
   use computational version to solve it:

2. Given instance N of computational problem,
   use decision problem to solve it:

Formal Problem Definitions

Page 1

`SHORTPATH(G,k)`
Input: Graph $G = (V, E)$, integer k
Output: Does G have a path of length *at most* k?

Input size and encoding:

`LONGPATH(G,k)`
Input: Graph $G = (V, E)$, integer k
Output: Does G have a path of length *at least* k?

Input size and encoding:

Formal Problem Definitions

Page 2

`FACT(N,k)`
Input: Integers N and k
Output: Does N have a prime factor less than k?

Input size and encoding:

`VC(G,k)`
Input: Graph $G = (V, E)$, integer k
Output: Does G have a vertex cover with at most k nodes?

Input size and encoding:

Our first complexity class

Complexity theory is all about classifying problems based on difficulty.

Definition

The complexity class **P** consists of all decision problems that can be solved by an algorithm whose worst-case cost is $O(n^k)$, for some constant k, and where n is the bit-length of the input instance.

This is the "polynomial-time" class. Can you name some members?

Certificates

A *certificate* for a decision problem is some kind of digital "proof" that the answer is `YES`.

The certificate is usually what the output *would be* from the "computational version".

Examples (informally):

- Integer factorization
- Minimum vertex cover
- Shortest path
- Longest path

Nice properties of **P**

When we just worry about polynomial-time, we can be *really lazy* in analysis!

Polynomial-time is closed under:

- **Addition**: $n^k + n^\ell \in O(n^{\max(k,\ell)})$
  In terms of algorithms: one after the other.
- **Multiplication**: $n^k \cdot n^\ell \in O(n^{k+\ell})$
  In terms of algorithms: calls within loops.
- **Composition**: $n^k \circ n^\ell \in O(n^{k\ell})$
  In terms of algorithms: replace every primitive op. with a function call

Verifiers

A *verifier* is an algorithm that takes:

1. Problem instance (input) for some decision problem
2. An alleged certificate that the answer is `YES`

and returns `YES` iff the certificate is legit.

Principle comes from "guess-and-check" algorithms:

- Finding the answer is tough, but
- checking the answer is easy.

We can write fast verifiers for hard problems!

Our second complexity class

Definition

The complexity class **NP** consists of all decision problems that have can be *verified* in polynomial-time in the bit-size of the original problem input.

Steps for an **NP**-proof:

1. Define a notion of certificate
2. Prove that certificates have length $O(n^k)$ for some constant k
3. Come up with a verifier algorithm
4. Prove that the algorithm runs in time $O(n^k)$
   for some (other) constant k

VC is in NP

VC(G,k): "Does G have a vertex cover with at most k vertices?"

1. Certificate:
2. Certificate size:
3. Verifier algorithm:
4. Algorithm cost:

FACT is in NP

FACT(N,k): "Does N have a prime factor less than k?"

1. Certificate:
2. Certificate size:
3. Verifier algorithm:
4. Algorithm cost:

How to get rich

The **BIG** question is: Does **P** equal **NP**?

The Clay Institute offers $1,000,000 for a proof either way.

- What you would need to prove $\mathbf{P} = \mathbf{NP}$:
- What you would need to prove $\mathbf{P} \neq \mathbf{NP}$:

In a nutshell: Is guess-and-check ever the best algorithm?

Alternate meaning of **NP**

Meaning of the name **NP**: "Non-deterministic polynomial time"

Non-deterministic Turing machine

- Turing machine with (possibly) multiple transitions for the same current state and current tape symbol
- Like a computer program with "guesses"
- Connection to randomness?

Why is this equivalent to our definition with certificates and verifiers?

Reductions

Recall that a reduction from problem A to problem B is a way of solving problem A using *any algorithm* for problem A.
Then we know that A is not more difficult than B.

Formally, a reduction from A to B:

1. Takes an *instance* of problem A as input
2. Uses this to create m instances of problem B
3. Uses the solutions to those m problem B's to recover the solution for the original problem A

Example Linear-Time Reduction

Two problems:

- `MMUL(A,B)`: Compute the product of matrices A and B
- `MSQR(A,B)`: Compute the matrix square A^2

Show that the inherent difficulty of `MMUL` and `MSQR` is the same.

Polynomial-Time Reduction

Ingredients for analyzing a reduction:
(All will be functions of n, the input size for problem A)

- Number (m) of problem B instances created
- Maximum *bit-size* of a problem B instance
- Amount of extra work to do the actual reduction.

Polynomial-time reduction: all three ingredients are $O(n^k)$
(Often $m = 1$, sometimes called a "strong reduction".)

We write $A \leq_{\mathbf{P}} B$, meaning
"A is polynomial-time reducible to B".

Formal Problem Definitions

Page 3

Minimum Hitting Set: `HITSET(L,k)`

Input: List L of sets $S_1, S_2, \ldots, S_m$, integer k.
Output: Is there a set H with size at most k such that every $S_i \cap H$ is not empty?

Input size and encoding:

`HAMCYCLE(G)`

Input: Graph $G = (V, E)$
Output: Does G have a cycle that touches every vertex?

Input size and encoding:

`VC` reduces to `HITSET`

HAMCYCLE reduces to LONGPATH

Completeness

Definition

A problem B is **NP**-hard if $A \leq_{\mathbf{P}} B$ for **every** problem $A \in \mathbf{NP}$.

Informally: **NP**-hard means "at least as difficult as every problem in **NP**"

Definition

A problem B is **NP**-complete if B is **NP**-hard and $B \in \mathbf{NP}$.

What is the hardest problem in **NP**?

An easy **NP**-hard proof

Theorem: The halting problem is **NP**-hard.

Proof:

Formal Problem Definitions

Page 4

Circuit Satisfiability: `CIRCUIT-SAT(C)`

Input: Boolean circuit C with AND, OR, and NOT gates,
m inputs, and one output.
Output: Is there a setting of the m inputs that makes the output true?

Input size and encoding:

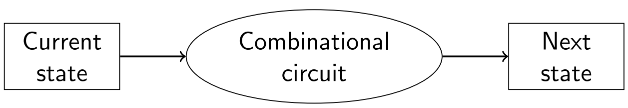
`3-SAT(F)`

Input: Boolean formula F in "conjunctive normal form"
(product of sums), with three literals (terms) in every sum (clause):
$F = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee x_4 \vee x_5) \wedge (x_1 \vee x_2 \vee \neg x_4) \wedge \cdots$
Output: Can we assign T/F to the x_i's to make the formula true?

Input size and encoding:

Modeling programs as circuits

Remember this simple model of a computer?



- **State** contains PC, registers, program, memory
  Size is linear in input size and program runtime
- **Combinational** is a circuit (AND, OR, and NOT gates)
  for ALUs, MUXes, control, shifts, adders, etc.
  Size is polynomial in size of state.

Lemma

Any decision problem with a polynomial-time algorithm can be simulated by a polynomial-size boolean circuit.

`CIRCUIT-SAT` is **NP**-hard

NP-Completeness

Theorem
*CIRCUIT-SAT is **NP**-complete.*

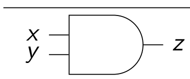
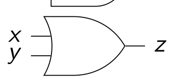
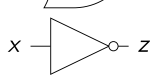
Proof: All that's left is to show `CIRCUIT-SAT` $\in$ **NP**.

- We only have to do this kind of proof once (why?)
- Will this help us prove **P** $\neq$ **NP**?

3-SAT

We want to reduce `CIRCUIT-SAT` to `3-SAT`.

Idea: Every wire in the circuit becomes a variable.

Gate	Formula
	$(\neg x \vee \neg y \vee z) \wedge (x \vee \neg z) \wedge (y \vee \neg z)$
	$(x \vee y \vee \neg z) \wedge (\neg x \vee z) \wedge (\neg y \vee z)$
	$(x \vee z) \wedge (\neg x \vee \neg z)$

- What do these clauses ensure?
- What other clause do we need to add?

VC

Reduce `3-SAT` to `VC`.

Properties of **NP**-Complete Problems

There are many known **NP**-complete problems.

We have seen: `LONGPATH`, `VC`, `HITSET`, `HAMCYCLE`, `CIRCUIT-SAT`, `3-SAT`.

What's needed to prove a new problem is **NP**-complete:

Note: All have *one-sided* verifiers (can't verify `NO` answer!)

What about `FACT`?

Frontiers of Complexity Theory

Big open questions:

- Does **P** = **NP**? (Probably not)
- Is `FACT` **NP**-complete? (Probably not)
- Is `FACT` in **P**? (Hopefully not!)
- Do true one-way functions exist? (Not if **P** = **NP**)
- Can quantum computers solve **NP**-hard problems? (Probably not)
- Where does randomness fit in?

Traveling Salesman Problem

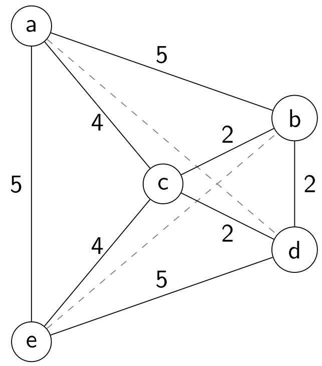
TSP Definition

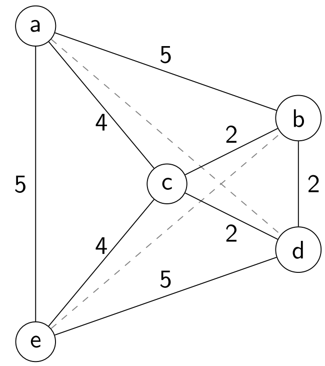
Input: Graph $G = (V, E)$
Output: The shortest cycle that includes every vertex exactly once, or `FAIL` if none exist.

- Classic **NP**-hard problem
- Many important applications
- The worst-case is hard — so what can we do?

MSTs and TSP

Theorem: Length of TSP tour is at least the size of a MST.





Branch and Bound

How to compute the optimal TSP?

1. Pick a starting vertex
2. Explore every path, depth-first
3. Return the least-length Hamiltonian cycle

This is really slow (*of course*!)

Branch and bound idea:

- Define a quick lower bound on remaining subproblem (MST!)
- Stop exploring when the lower bound exceeds the best-so-far

Simplified TSP

Solving the TSP is really hard; some special cases are a bit easier:

Metric TSP

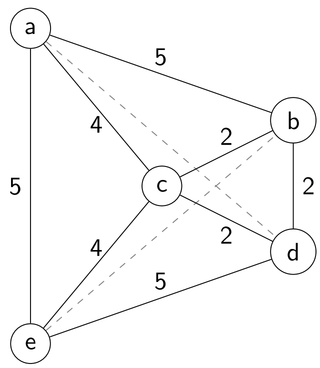
- Edge lengths “obey the triangle inequality”: $w(a,b) + w(b,c) \geq w(a,c) \forall a,b,c \in V$
- What does this mean about the graph?

Euclidean TSP

- Graph can be drawn on a 2-dimensional map.
- Edge weights are just distances!
- (Sub-case of Metric TSP)

Approximating Metric TSP

Idea: Turn any MST into a TSP tour.

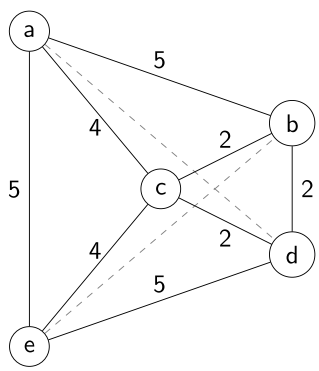


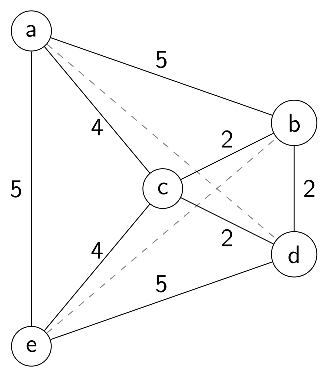
How good is the approximation?

Greedy TSP

Greedy strategies:

- Nearest neighbor
- Smallest "good" edge





Local Refinement

Idea: Take any greedy solution, then make it better.

2-OPT refinement:

- Take a cycle with (a, b) and (c, d)
- Replace with (a, c) and (b, d)