Computational complexity

Time complexity

Non-determinism

1. Computational complexity
   - Computational complexity models
   - Asymptotics

2. Time complexity
   - Polynomial time
   - Problems in $P$
   - Example: the Euclidean algorithm

3. Non-determinism
   - Non-deterministic polynomial time
   - Verifiers and certificates
   - $NP$-completeness
   - $3SAT$ is $NP$-complete

Complexity theory

Main question: computability: what is computable?
Complexity: what is computable?

Efficient computation:
- Classify problems according to their computational requirements
  - Time, space, randomness
- Complexity of the problem, and not the complexity of a specific solution/algorithm

Time and space

- Time and space complexity
- How many steps does it take to solve a problem?
- How much memory is required to solve a problem?
- Use the Turing machine model to count resource requirements
Classifying problems

- Recognizable (RE)
  - Decidable (Recursive)
    - Context free
    - Regular

Complexity

- many Turing machine models: single tape, multi-tape, non-deterministic etc.
- a single tape TM simulates all of them, but with an overhead
- develop a model-independent way to measure resource requirements
- and therefore characterize the complexity or difficulty of the problem and not
  - the power of a particular machine
  - efficiency of a specific algorithm
- Compare different problems: easy problems, hard problems

Asymptotics

- asymptotics (big-Oh notation) play a crucial role
- abstract complexity becomes robust and model independent for asymptotic complexity
- sorting $n$ numbers might require
  - $5n \log n + 1270$ steps on a deterministic two tape TM
  - $120n \log n + 110$ on a single tape deterministic TM
  - use $O(n \log n)$
Running time

- Let $M$ be a TM that halts on all inputs
- **(definition 7.1p248)** the running time of $M$ is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n) = \max_{|x|=n} \{\text{no. of steps of } M \text{ on } x\}$
- we say $M$ is a $f(n)$-time TM or the running time of $M$ is $f(n)$

Big Oh

- **(definition 7.2p249)** let $f, g : \mathbb{N} \to \mathbb{R}^+$. $f(n) = O(g(n))$ if positive constants $c, n_0$ exist such that for every integer $n \geq n_0$ $f(n) \leq cg(n)$
- $g(n)$ in an asymptotic upper bound for $f(n)$
- asymptotic: modulo some initial $n$'s and the constant $c$
- remember, we are interested in the (growth of the) function, and not its absolute value
- some examples $O(1), n^{O(1)}, O(n + m), \ldots$

small-oh

- **(definition 7.5p250)** let $f, g : \mathbb{N} \to \mathbb{R}^+$. $f(n) = o(g(n))$ if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$
- $f(n) = o(g(n))$ if for any $c > 0$ there exists $n_0$ such that for every integer $n \geq n_0$ $f(n) < cg(n)$
- $O(g) = \{\text{all functions asymptotically less or equal to } g\}$
- $o(g) = \{\text{all functions asymptotically strictly less than } g\}$
- both notations used: $f \in O(g)$ and $f = O(g)$

Omegas, $\Omega, \omega$

- Oh’s were defined using ‘less than’
- and therefore are used for denoting upper bounds (for example worst case running times)
- corresponding notation is used for lower bounds
- $\Omega(g), \omega(g)$
Analyzing running times

- example: consider a single tape TM accepting the language \( \{0^k1^k\} \)
- we can do this in \( O(n^2) \)
- is this the real complexity of the problem or just a not-so-clever solution?
- how about \( O(n \log n) \)?

**Analyzing running times**

- checking \( \{0^k1^k\} \)
- repeat
  1. check odd-even parity of all 0s and 1s.
     - if odd reject
  2. cross-off every other 0 and every other 1
- until no 0s or no 1s remain
- if no 0s and no 1s remain accept. otherwise reject
- algorithm checks if all parities agree for 0s and 1s

---

**TIME\( (t(n)) \)**

- (definition 7.7p251) for \( t : \mathbb{N} \rightarrow \mathbb{R}^+ \), define \( TIME(t(n)) \) to be the collection of all languages that are decidable by an \( O(t(n)) \)-time TM
- for example the problem of checking if an input has the form \( L = \{0^k1^k\} \) is in \( TIME(n \log n) \)
- a two tape machine can solve \( L \) in \( O(n) \)
- but for a single tape TM \( L \notin TIME(n) \)

---

**Model dependencies**

- the running times and the classes of languages \( TIME(t) \) are model dependent
- \( TIME(n) \) contains different languages for single tape TMs and different for 2 tapes
- define model independent classes of languages
- "robust" complexity classes
**Complexity across models**

- $L = \{0^k1^k\}$ is in
  - $\text{TIME}(n \log n)$ for single tape deterministic TMs
  - $\text{TIME}(n)$ for two tape deterministic TMs
- **(theorem 7.8p254)** for $t(n) \geq n$ any $t(n)$ time multi-tape TM can be simulated by a single tape $t^2(n)$ machine
- proof: use the simulation of multi-tape machines by a single tape as mentioned in theorem 3.13p149
  - every step of the multi-tape machine requires at most $O(t(n))$ steps in the single tape simulation
  - total running time $t^2$

**Polynomial time**

- non-deterministic computation
- recall the notion of a computation tree for non-deterministic computations
- **(theorem 7.9p256)** Every $t(n)$ step non-deterministic single tape computation has an equivalent $2^{O(t(n))}$ deterministic single tape computation
- proof: use simulation from theorem 3.16p150
  - computation tree of height $t$, fan-out $b$
  - explore (simulate) all tree paths breadth first
  - every node can have at most $b$ children ($b$ is determined by the machines transition function)

- number of nodes is less than twice the number of leaves, so $O(b^t(n))$
- to visit a new node, start from the root down the non-deterministic path- at most $t(n)$ steps
- total time $O(tb^t) = 2^{O(t)}$

- defining robust model independent complexity classes
- **(definition 7.12p258)** $P$ is the class of languages that decidable in polynomial time on a single tape deterministic TM
  $$P = \bigcup_k \text{TIME}(n^k)$$
- Robustness: $P$ is invariant for all polynomially equivalent models of computation
- $P$ is a reasonable characterization of problems that we can solve efficiently

---

**University of Sydney**

**COMP3310/3610: Theory of Computation**
**Strong Church Turing thesis**

- All reasonable computation models are polynomial time/space equivalent
- It is possible to simulate a model with a machine from another model in polynomial time/space

**Unreasonable models**

- Unreasonable models may have unrealistic requirements (for our current knowledge)
- Analog computing and infinite precision, unbounded parallel computing
- What about non-determinism?
- What about quantum computing?

**Introduction to Complexity theory**

- Complexity theory: classifying problems
- Resource requirements (time, space)
- Model independent characterizations, when possible
- Classes of difficult problems, easy problems etc

**Polynomial time**

- Defining robust model independent complexity classes
  - **Definition 7.12p258** $P$ is the class of languages that decidible in polynomial time on a single tape deterministic TM
    $$ P = \bigcup_k \text{TIME}(n^k) $$
- Robustness: $P$ is invariant for all polynomially equivalent models of computation
- $P$ is a reasonable characterization of problems that we can solve efficiently
Problems in $P$

- given a graph, is there a path connecting a given pair of its vertices?
  - $\text{PATH} \in P$
- assume graph is given by its adjacency matrix
- on input $G, s, t$ do the following
  1. mark node $s$
  2. repeat the following until no new nodes are marked
  3. scan all edges of $G$. if there is an edge $(a, b)$ where only $a$ is marked then mark $b$ as well
  4. if $t$ is marked accept otherwise reject

Relatively prime numbers

- Checking whether two numbers are relatively prime
- two numbers are called relatively prime if they do not have any common divisors (except 1)
  - $\text{RELPRIME} = \{ <x, y> | x, y \text{ are relatively prime} \}$
  - (theorem 7.15p261) $\text{RELPRIME} \in P$
- proof: use the Euclidean algorithm for the greatest common divisor
- note that a straightforward approach does not yield a polynomial algorithm
- brute-force is exponential: try all smaller numbers to see if there is any non-trivial divisor

The Euclidean algorithm

- The Euclidean algorithm finds the greatest common divisor of two natural numbers
- on input $(x, y)$
  1. repeat until $y = 0$
     1. $x := x \mod y = x - \lfloor \frac{x}{y} \rfloor y$
     2. swap $x$ and $y$
  2. Output $x$
- every loop cuts a number by half (at least)
- at the beginning of each iteration $x > y$
- if $x/2 \geq y$ then $x \mod y < y \leq x/2$
- if $x/2 < y$ then $x \mod y = x - y < x/2$
- let $x = qy + r$ or $r = x - qy$
  - $q$ is the quotient and $r$ the remainder of the division $x \div y$
  - if $d$ is a common divisor of $x, y$ then $x = dz$ and $y = dw$
  - $r = x - qy = (z - qw)d$ which means that any common divisor of $x, y$ also divides the remainder $r$
- this is true for greatest common divisor as well so
  1. if $y = 0$ return $x$
  2. return $\text{gcd}(y, x \mod y)$
The class \( NP \)

- Brute-force search sometimes leads to inefficient algorithms
- Can we always avoid brute-force search?
- \( NP \), a class of problems
- many practical problems of similar complexity do not seem to allow any efficient solution
- (Deterministic) polynomial solutions are not known, but it is also not known whether those problems actually require more than polynomial time

On \( NP \)

- many other problems seem to resist polynomial time solutions
- many of these problems are strongly related: if you solve one you have a solution for all of them
- \( HAMPATH \): two interesting properties
  - no poly-time algorithm known
  - a correct solution can be verified in poly-time
- in other words, the problem seems hard to solve but easy to verify

Hamiltonian paths

- given a directed graph \( G \) and two of its vertices \( s, t \), is there a path connecting \( s \) with \( t \) that visits all vertices exactly once?
- \( HAMPATH = \{(G, s, t) | G \text{ is a directed graph, with a Hamiltonian path from } s \text{ to } t\} \)
- brute-force algorithm will work. check all possible paths from \( s \) to \( t \)
- it is an open problem whether a polynomial solution exists

Solving versus verifying

- is it easier to solve a problem or just verify the solution to problem?
- this question can be formalized in complexity theoretic terms
- verifiers and poly-time verifiable problems
- \( HAMPATH, COMPOSITES \)
- what about primality? is it poly time verifiable?
- primality is now known to be in \( P \)
Verifiers and certificates

- consider a problem $A$ and the associated decision problem $x \in A$?
- sometimes it is possible to present proof of membership or a certificate that proves that a given $x \in A$
- (definition 7.18p265) A verifier for a language $A$ is an algorithm $V$ where
  \[ A = \{ w | V \text{ accepts}(w, c) \text{ for some string } c \} \]
- a polynomial verifier for $A$ runs in time poly. in $w$
- if $A$ has a polynomial verifier, then it is called polynomially verifiable

Verifying certificates

- the verifier uses the information of the certificate $c$
- note that for polynomial verifiers, the certificate must be polynomial in length
- examples of certificates
  - $HAMPATH$: a directed path
  - $COMPOSITES$: a divisor
  - what about primality?

Definitions of $NP$

- (definition 7.19p266) $NP$ is the class of languages that have polynomial time verifiers
  - remember: $NP$ is a polynomial time class
- (theorem 7.20p266) $NP$ is the class of languages that are decided by some non-deterministic poly-time Turing machine
  - $NP$ stands for non-deterministic polynomial
  - $P$: languages with poly-time deciders
  - $NP$: languages with poly-time verifiers

NP and non-determinism

- (theorem 7.20p266) A language is in $NP$ iff it is decided by some non-deterministic poly-time Turing machine
  - proof: show how to convert a certificate to a NTM and vice versa
  - $\implies$: assume the verifier $V$ runs in time $n^k$
  - to decide $w \in A$, on input $w$ (of length $n$)
    - non-deterministically select the certificate $c$ of length at most $n^k$
    - run $V$ on $(w, c)$
    - if $V$ accepts, accept, o.w. reject
**NP and non-determinism**

- *(Theorem 7.20p266)* A language is in *NP* iff it is decided by some non-deterministic poly-time Turing machine.
- **proof**: show how to convert a certificate to a NTM and vice versa.
- $\iff$: assume $N$ is the NTM that decides $A$.
- construct a verifier as follows: on input $(w, c)$
  - simulate $N$ on $(w, c)$. use every symbol of $c$ to pick which non-deterministic choice of $N$ we should follow each time (remove non-determinism).
  - if $c$ leads down a path to an accept, accept; otherwise reject.

---

**Examples of problems in NP**

- **CLIQUE**: given a graph, does it contain a $k$-clique?
- **HAMPATH**: given a graph does it contain a Hamiltonian path?
- **SUBSET – SUM**: given a collection of numbers and a target $t$, does there exist a subset with sum equal to $t$?
  - $\text{SUBSETSUM} = \{(S, t) | S = \{s_1, \ldots, s_k\} \text{ and for some } \{y_1, \ldots, y_l\} \subseteq S \sum y_i = t\}$
- what about the complements of these sets?
- what kind of certificate can you give for something that does not exist?
Boolean Satisfiability

- given a boolean formula $\varphi$ does there exist an assignment of 0,1 to the variables, that makes the entire formula true?
- if a satisfying truth assignment exists for $\varphi$ then $\varphi$ is called satisfiable
- $SAT = \{\varphi|\varphi$ is a satisfiable boolean formula$\}$
- $SAT \in NP$

NP-completeness

- $NP$-completeness was introduced independently by Cook and Levin (70’s)
- the first $NP$-complete problem, $SAT$
- the Cook-Levin theorem: $SAT$ is $NP$-complete (theorem 7.27p267)
- proof: use computation histories: construct a formula such that an accepting computation exists iff the formula is satisfiable
- show how to reduce any $A \in NP$ to a $SAT$ question
- all we know is that $A \in NP$ and therefore the only thing we can use is that a NTM computation exists

3SAT

- (corollary 7.42p282) $3SAT$ is $NP$-complete
- proof: any $SAT$ formula can be converted to a $3SAT$ that is up to polynomially longer
- use the distributive laws of boolean algebra to write the Cook-Levin reduction formula as a cnf
- $\varphi_{cell} \land \varphi_{start} \land \varphi_{move} \land \varphi_{accept}$
- big clauses (more than 3 literals) can be split into smaller ones by introducing new variables