ITCS 532:
6. Tractability and p-Time Reduction

Rob Egrot

Computational Complexity Theory

- So far in this course we have only asked whether decision problems are decidable or semidecidable.
- We only care about the existence, or provable non-existence, of algorithms.
- We don't care how fast or slow they are.
- E.g. Turing machines with multiple tapes are 'equivalent' to standard Turing machines.
- In the real world, it is important that algorithms run in a 'reasonable amount of time'.
- Here we start taking the running times of algorithms seriously.
- This is the start of computational complexity theory.

Measuring Run Time - Computation Steps

- We want our measure of the running time of an algorithm to be independent of the hardware we run it on.
- It might take my desktop several hours to run an algorithm that a supercomputer could run in a few seconds.
- To solve this, we think about the number of steps involved in running the algorithm.
- We assume each computational step takes a constant amount of time on each system.
- So in practice run time is a constant multiple of the number of steps (depending on computer power).
- But the number of steps is the important thing for us.
- We can think of these steps as being steps in a Turing machine computation, or, more practically, CPU cycles.

Measuring Run Time - Inputs

- Another thing to consider is that we want our algorithms to run on *inputs*.
- So the time an algorithm takes will depend on the input we give it.
- I.e. Run time is a function of the input (strings over a finite alphabet).
- Usually impossible to describe this function exactly, so we think about the lengths of the inputs.
- Different inputs of the same length may have different run times, so we ask about the *worst case*.
- I.e. what is the slowest possible (halting) run time of this algorithm for an input of length n?
- This will give us a function of n, e.g. $f(n) = 4n^3 - 5n + 6$.
- We ignore the cases where the computation does not halt.
- We cannot usually define this function explicitly, so we think about bounds for it.

Worst Case Analysis

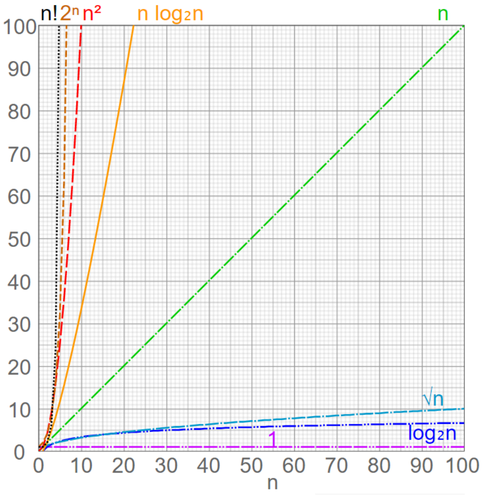
Definition 1 (Big O Notation)

Let f and g be functions from $\mathbb{R}$ to $\mathbb{R}$, and suppose $g(x)$ is strictly positive for large enough values of x (i.e. after some point the values of $g(x)$ are all bigger than zero). We say $f = O(g)$ as $x \to \infty$ if there are $x_0, c \in \mathbb{R}$ such that $|f(x)| \leq cg(x)$ for all $x \geq x_0$.

- We use big O notation to roughly classify the worst case running times of algorithms.
- For example, if $f(n) = 4n^3 - 5n + 6$ then $f = O(n^3)$.
- Intuitively, for large n, $f(n)$ is bounded above by cn^3.
- We are mainly interested in finding algorithms whose big O worst case run time is as small as possible.
- In the real world we are often interested in the *average case*.
- For example, *Quicksort* has a worst case run time of $O(n^2)$, but the algorithm usually runs in $O(n\log(n))$.

Measuring Run Time - Growth Rate Comparison

The following diagram (taken from the Wikipedia page for Big O notation) illustrates the growth rates some common functions used in big O notation.



Algorithm Efficiency vs Computer Power

- We want to understand why designing better algorithms for difficult problems is better than making faster computers (though faster computers are good too!).
- Note that the 'worst case analysis' here only applies to 'large' values of n.
- For small n values the algorithm behaviour may be different.
- We ignore this because small inputs are easy to deal with.
- With inefficient algorithms, there is usually a limit point where input sizes become 'too big'.
- We ask how much increasing computer power increases this limit.

Algorithm Efficiency vs Computer Power - Linear Case

- Suppose we have an algorithm and we have computers C_1 and C_2 running this algorithm.
- C_2 is capable of performing 2^{12} computation steps in the time it takes C_1 to do 1.
- So C_2 is 4096 times faster than C_1.
- In symbols $v_2 = 2^{12} v_1$.
- Suppose first that the algorithm runs in linear time ($O(n)$).
- What is the biggest input size machine C_1 can guarantee to handle in time t (call this n_1)?

Algorithm Efficiency vs Computer Power - Linear Case

- We have $f(n_1) \leq cn_1$ for some c.
- To guarantee that C_1 completes the computation in at most time t, we require that $cn_1 \leq v_1 t$.
- Rearranging the formula, we must have $n_1 \leq \frac{v_1 t}{c}$.
- I.e., $n_1 = \lfloor \frac{v_1 t}{c} \rfloor$.
- What about C_2?
- Remember that $v_2 = 2^{12} v_1$.
- Similar to the case of C_1, we require $cn_2 \leq v_2 t$, which rearranges to give $n_2 \leq \frac{v_2 t}{c}$.
- I.e. $n_2 = \lfloor \frac{2^{12} v_1 t}{c} \rfloor \approx 2^{12} n_1$.
- This is 2^{12} times bigger than for C_1, so is a huge improvement.

Quadratic Case

- Suppose now the algorithm runs in quadratic time.
- I.e. $f(n)$ is $O(n^2)$, so $f(n) \leq cn^2$ for some constant c.
- Let v_1 and v_2 be as before.
- To guarantee completion in at most time t by C_1 on an input of length n_1, we must have $cn_1^2 \leq v_1 t$.
- I.e. $n_1 \leq \sqrt{\frac{v_1 t}{c}}$.
- Similarly, for C_2, we must have input length

$$n_2 \leq \sqrt{\frac{v_2 t}{c}} = \sqrt{\frac{2^{12} v_1 t}{c}} = 2^6 \sqrt{\frac{v_1 t}{c}}.$$

- In other words, $n_2 \approx 2^6 n_1$.

Exponential Case

- Suppose now the algorithm that runs in exponential time.
- E.g. f is $O(2^n)$.
- Then we have $2^{n_1} \leq \frac{v_1 t}{c}$ and $2^{n_2} \leq \frac{v_2 t}{c}$ for some constant c.
- Taking logs base 2:
  - $n_1 \leq \log(v_1) + \log(t) - \log(c)$.
  - $n_2 \leq \log(v_2) + \log(t) - \log(c)$.
- Since $v_2 = 2^{12} v_1$, the second inequality can be rewritten as

$$\begin{aligned} n_2 &\leq \log(2^{12} v_1) + \log(t) - \log(c) \\ &= \log(2^{12}) + \log(v_1) + \log(t) - \log(c) \\ &= 12 + \log(v_1) + \log(t) - \log(c). \end{aligned}$$

- I.e. $n_2 \approx n_1 + 12$.
- A very small improvement.

Tractability and Intractability

- Complexity theory divides algorithms into rough complexity classes based on their use of resources.
- The classes we discuss here are based on worst case run time.
- The difference between $O(n^2)$ and $O(n^3)$ can be very important in applications.
- For example, in some applications an $O(n^3)$ algorithm might be much too slow, but an $O(n^2)$ algorithm might be manageable.
- For some applications, e.g. in Big Data, $O(n^2)$ might be much too slow.
- Despite this, theorists often make a sharp distinction between 'efficient' and 'inefficient' algorithms, and between 'easy' and 'hard' decision problems, based on the following definitions.

Tractability and Intractability

Definition 2 (polynomial time)

An algorithm is polynomial time (p-time) if it is $O(n^k)$ for some $k \in \mathbb{N}$.

Definition 3 (tractable)

A decision problem is tractable if there is a polynomially time algorithm that decides it. It is *intractable* otherwise.

- As an approximation, complexity theorists consider tractable problems to be ‘easy’, and intractable ones to be ‘hard’.
- In practice, a polynomial time algorithm might be much too slow for practical use, as mentioned before.
- Of course, finding a polynomial time algorithm for a problem, thus demonstrating its ‘easiness’, may not be easy at all!

The simple PRIME Algorithm

Consider the following algorithm:

```
Prime(n).
answer = 'true'
for i = 2 to n-1
if i divides n then answer = 'false'
return answer.
```

The Run Time of PRIME

- Say we have an $O(n^2)$ algorithm to test if i divides n.
- Worst case run time of 'Prime' is $O(n^3)$.
- But to run this on a TM we need to encode natural numbers using a finite alphabet.
- Length of the input is the number of symbols used.
- E.g. a binary number of length n can represent a natural number up to $2^n - 1$.
- So considered as a function of binary input length this algorithm has a worst case run time of $O((2^n)^3) = O(2^{3n})$.
- But if we use unary notation then the algorithm does run in p-time in length of input.
- So there is a relationship between how we encode problems and the running times of algorithms we can use to solve them.
- We can make algorithms look more efficient by encoding the problem in an inefficient way.

Polynomial Time Reduction

- Previously we saw how we can order decision problems in terms of their decidability using reduction.
- That is, $A \leq B$ if there's an algorithm that turns 'yes' instances of A into 'yes' instances of B, and 'no' instances of A into 'no' instances of B.
- I.e. problem B is 'at least as hard' as problem A.
- We can extend the idea of reduction to order decision problems in terms of their polynomial time solvability.
- We write $A \leq_p B$ if there is a p-time algorithm that reduces A to B.
- Then, if we had a p-time algorithm for solving B we could combine it with the p-time conversion algorithm to get a p-time algorithm for solving A.

Polynomial Time Reduction

- Actually there is a complication here.
- Our first algorithm converts instances of A to instances of B in p-time, and the second algorithm solves B in p-time.
- But the second algorithm runs in p-time on the size of its input, and the algorithm that converts instances of A to instances of B does not necessarily preserve their sizes.
- Could the converted input be more than polynomial in the length of the original?
- No, because the fact that the conversion algorithm is p-time bounds the size of the converted instance.
- I.e. if the conversion algorithm runs in time n^k and the original A instance has size m, then the converted instance can be at most size cm^k (constant c).
- If the algorithm for solving B is $O(n^l)$, the worst case run time for the combined algorithm is $O((n^k)^l) = O(n^{kl})$.

Properties of $\leq_p$

Theorem 4

$\leq_p$ is transitive.

Proof.

- Suppose $A \leq_p B$ and $B \leq_p C$.
- If I is an instance of A let $f(I)$ be the result of applying the reduction of A to B to I.
- If J is an instance of B let $g(J)$ be the result of applying the reduction of B to C to J.
- $g(f(I))$ is a reduction of A to C.
- We need to check $g \circ f$ is p-time.
- Suppose the reduction of A to B is $O(n^k)$ for some k, and the reduction from B to C is $O(n^l)$ for some l.
- Then the maximum size of $f(I)$ is $O(n^k)$, so $g \circ f$ is $O((n^k)^l) = O(n^{kl})$.

$\square$

Properties of $\leq_p$

Exercise 5

Why is it obvious that $\leq_p$ is reflexive?

Theorem 6

Let A and B be decision problems. Then, if there is a p-time algorithm for solving A, and if B has at least one 'yes' instance and at least one 'no' instance, then $A \leq_p B$.

Proof.

- There's a p-time algorithm for solving A, by assumption.
- Just use this algorithm to check if they are 'yes' or 'no' instances of A then convert them to either the 'yes' or 'no' instance of B appropriately.
- This 'conversion' is constant time, as we always convert to either the fixed 'yes' instance, or the fixed 'no' instance.

□

P-Time Equivalence

- We know that $\leq_p$ is not symmetric.
- Because if $\leq_p$ were symmetric it would follow from theorem 6 that every decision problem would have a p-time solution.
- Since there are problems that are not even decidable this is impossible.
- However, we can use $\leq_p$ to define a relation between decision problems that is reflexive, transitive, and symmetric (i.e. an equivalence relation).

Definition 7 ($\equiv_p$)

Decision problems A and B are *p-time equivalent* if $A \leq_p B$ and $B \leq_p A$.

The Hamiltonian Circuit Problem

Definition 8 (Hamiltonian circuit)

Given a finite simple graph G with n vertices, a Hamiltonian circuit is a sequence $c = (v_1, v_2, \ldots, v_n)$ such that

1. if v_i and v_j occur consecutively in the sequence then there is an edge from v_i to v_j in G,
2. every vertex of G occurs in c, and
3. there is an edge from v_n to v_1.

Informally a Hamiltonian circuit is a path in G that passes through every vertex exactly once before returning to its origin.

Definition 9 (HCP)

The Hamiltonian circuit problem (HCP) is whether a given finite undirected simple graph has a Hamiltonian circuit.

The Travelling Salesman Decision Problem

Definition 10 (weighted graph)

A weighted graph is a graph where every edge is associated with a number (usually a non-negative integer). This number can be thought of as the 'cost' of traveling along that edge.

Definition 11 ($TSDP$)

The traveling salesman decision problem ($TSDP$) has instances (G, d), where G is a finite complete undirected weighted simple graph with non-negative integer weights, and d is a non-negative integer. The question is whether there is a Hamiltonian circuit in G such that the sum of the weights of the edges used in the circuit is less than or equal to d.

$HCP \leq_p TSDP$

Theorem 12

$HCP \leq_p TSDP$, (p-time in the number of vertices plus the number of edges of the graph G).

Proof

- An instance of HCP is a finite undirected simple graph G with n vertices and e edges.
- We will convert G into a pair (G', d) where G' is a finite complete undirected weighted simple graph, and d is a non-negative integer.
- This is an instance of $TSDP$.
- We then show that 'yes' instance of HCP become 'yes' instances of $TSDP$, and similar for 'no' instances.
- Finally we check that the conversion algorithm runs in p-time as a function of $n + e$.

$HCP \leq_p TSDP$ - The Reduction

- Let G' have the same vertices as G.
- To be an instance of $TSDP$ G' must be complete, so we add edges for each pair of vertices.
- In G', edges from G get weight 0, new edges have weight 1.
- We set $d = 0$.
- So G' has a Hamiltonian circuit with total weight 0 if and only if there is a Hamiltonian circuit in G.

$HCP \leq_p TSDP$ - How Fast is the Reduction?

- The reduction is correct, but how fast is it?.
- We have to clone the vertices of G. This is $O(|V|)$.
- Then we add edges for pairs of vertices. There are $\binom{n}{2}$ pairs of vertices of G, so this is $O(|V|^2)$.
- Setting the weights involves checking every edge of G' to see if it corresponds to an edge of G.
- Can check every edge of G to see if it is an edge between v and v' for every pair $v, v' \in V$.
- This is $O(|E| \times |V|^2)$.
- Setting $d = 0$ is a constant time operation, say c. So the whole process takes $O(|V|) + O(|V|^2) + O(|E| \times |V|^2) + c$, which is $O((|V| + |E|)^3)$ at most.

$TSDP \leq_p HCP$

- The converse to the theorem we just proved is also true.
- If you are feeling ambitious you can try to prove it directly using p-time reduction.
- This is a lot harder than the direction proved in the theorem.
- We will prove it using some powerful general theory at the end of the course.