Module 6: Intractable Problems

Raveen de Silva (K17 202)
Course Admins: cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 1, 2024

Module Learning Outcomes

- Identify intractable problems and explain the relationships between them
- Solve intractable problems exactly and approximately using a variety of algorithm design techniques
- Communicate algorithmic ideas at different abstraction levels
- Evaluate the efficiency of algorithms and justify their correctness
- Apply the LaTeX typesetting system to produce high-quality technical documents

Table of Contents

1. Decision Problems

1.1 Feasibility of Algorithms

1.2 Polynomial Reductions

1.3 Intractable Decision Problems

2. Optimisation Problems

2.1 Aside: Linear Programming

2.2 Intractable Optimisation Problems

2.3 Superpolynomial Algorithms

2.4 Approximation Algorithms

3. Puzzle

Table of Contents

1. Decision Problems

1.1 Feasibility of Algorithms

1.2 Polynomial Reductions

1.3 Intractable Decision Problems

2. Optimisation Problems

2.1 Aside: Linear Programming

2.2 Intractable Optimisation Problems

2.3 Superpolynomial Algorithms

2.4 Approximation Algorithms

3. Puzzle

Polynomial Time Algorithms

Definition

A (sequential) algorithm is said to be *polynomial time* if for every input it terminates in polynomially many steps in the length of the input.

- Supposing the length of the input is denoted by n, this means that the worst case complexity satisfies $T(n) = O(n^k)$ for some integer k.
- Examples:
  - merge sort requires about $n \log n$ comparisons, and since $n \log n = O(n^2)$ it runs in polynomial time, but
  - the brute force algorithm for longest increasing subsequence (for each of the 2^n subsets of activities, check whether it is increasing, and update the max length) takes at least $n\,2^n$ operations, so it is not $O(n^k)$ for any integer k.

Length of Input

Question

What is the *length* of an input?

Answer

It is the *number of symbols* needed to describe the input precisely.

Length of Input: Integers

- For example, if input x is an integer, then $|x|$ can be taken to be the number of bits in the binary representation of x, i.e. $\lceil \log_2 x \rceil$.
  - As usual, rounding doesn't affect asymptotics so we will usually just write $\log_2 x$.
- The definition of polynomial time computability is quite robust with respect to how we represent inputs.
  - For example, we could instead define the length of an integer x as the number of digits in the *decimal* representation of x, i.e. $\log_{10} x$.
  - This can only change the constants involved in the expression $T(n) = O(n^k)$ (where n is the number of symbols $\log_? x$) but not the asymptotic bound.

Length of Input: Arrays

- If the input is an array A, then each entry $A[i]$ contributes $\log_2 A[i]$ bits, for a total of

$$\log_2 A[1] + \ldots + \log_2 A[n].$$

- If the array entries are guaranteed to be each at most M, then this can be abbreviated to $n \log_2 M$.
- A polynomial time algorithm therefore runs in time polynomial in n and $\log M$.
- Example:
    - In Module 5, we saw that the 0-1 Knapsack and Integer Knapsack problems have input size $n \log C$, but the standard DP algorithm runs in $O(nC)$.
    - Since $C = 2^{\log_2 C}$, we know that C isn't bounded by any power of $\log_2 C$ and hence this is *not* a polynomial time algorithm.
    - Such algorithms are said to be *pseudopolynomial*.

Length of Input: Weighted Graphs

- If the input is a (simple) weighted graph G with edge weights up to W, then G can be described using:
  - its adjacency list: for each vertex v_i, store a list of edges incident to v_i together with their (integer) weights represented in binary, or
  - its adjacency matrix: for each vertex v_i, for each vertex v_j, store the weight of the edge from v_i to v_j (or some default value to indicate a non-edge).
- The adjacency list takes $O(E \log W)$ space, whereas the adjacency matrix takes $O(V^2 \log W)$.
  - If the graph has to be sparse ($E \ll V^2$), then the adjacency list might be much more concise.
  - However, since we are interested only in whether the algorithm runs in polynomial time and not in the particular degree of that polynomial, this does not matter.

Length of Input: Weighted Graphs

- Example:
  - In Module 4, we saw the maximum flow problem. There are E edges, each with capacity up to C, so the size of the input is $O(E \log C)$.
  - The Ford-Fulkerson algorithm runs in $O(E|f|)$. Since the value of the maximum flow could be up to EC, this is *not* a polynomial time algorithm.
  - The Edmonds-Karp algorithm runs in $O(VE^2)$. Since $V \leq E - 1$, this is a polynomial time algorithm.
- In general, every precise description of the input without artificial redundancies will do.

Decision Problems and Class **P**

Definition

A *decision problem* is a problem with a YES or NO answer.

Examples include:

- "*Input number n is a prime number.*"
- "*Input graph G is connected.*"
- "*Input graph G has a cycle containing all vertices of G.*"

Definition

A decision problem $A(x)$ is in class $\mathbf{P}$ (*polynomial time*, denoted $A \in \mathbf{P}$) if there exists a polynomial time algorithm which solves it (i.e. produces the correct output for each input x).

Class **NP**

Definition

A decision problem $A(x)$ is in class **NP** (*non-deterministic polynomial time*, denoted $A \in \mathbf{NP}$) if there is a problem $B(x, y)$ such that

1. for every input x, $A(x)$ is true if and only if there is some y for which $B(x, y)$ is true, and
2. the truth of $B(x, y)$ can be verified by an algorithm running in polynomial time in the length of x *only*.

- We call y the *certificate*: it provides evidence for why x is a YES instance of problem A.
- We call B the *certifier*: it checks that evidence in polynomial time.

Example: Primality Testing

Question

Consider the decision problem $A(x) =$ “integer x is not prime”. Is $A \in \mathbf{NP}$?

Answer

- We need to find a problem $B(x, y)$ such that $A(x)$ is true if and only if there is some y for which $B(x, y)$ is true.
- The natural choice is $B(x, y) =$ “x is divisible by y”.
- $B(x, y)$ can indeed be verified by an algorithm running in time polynomial in the length of x only.

Example: Primality Testing

Question

Is $A \in \mathbf{P}$?

Answer

- Also yes! But this is not at all straightforward. This is a famous and unexpected result, proved in 2002 by the Indian computer scientists Agrawal, Kayal and Saxena.
- The AKS algorithm provides a *deterministic*, *polynomial time* procedure for testing whether an integer x is prime.

Example: Primality Testing

The length of the input for primality testing is $O(\log x)$.

Comparing some well-known algorithms for primality testing:

- The naïve algorithm tests all possible factors up to the square root, so it runs in $O(\sqrt{x})$ and is therefore not a polynomial time algorithm.

- The Miller-Rabin algorithm runs in time proportional to $O(\log^3 x)$ but determines only *probable primes*.

- The original AKS algorithm runs in $\tilde{O}(\log^{12} x)$, and newer versions run in $\tilde{O}(\log^6 x)$.

- However, the AKS algorithm is rarely used in practice; tests using elliptic curves are much faster.

Example: Satisfiability

Satisfiability

Instance: a propositional formula in the CNF form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each clause C_i is a disjunction of propositional variables or their negations, for example

$$(P_1 \vee \neg P_2 \vee P_3 \vee \neg P_5) \wedge (P_2 \vee P_3 \vee \neg P_5 \vee \neg P_6) \wedge (\neg P_3 \vee \neg P_4 \vee P_5)$$

Problem: "There exists an evaluation of the propositional variables which makes the formula true."

- Clearly, given an evaluation of the propositional variables one can determine in polynomial time whether the formula is true for such an evaluation.
- So Satisfiability (SAT) is in class **NP**.

Example: Satisfiability

Question

Is SAT in class **P**?

(Partial) Answer

If each clause C_i involves exactly two variables (2SAT), then yes!

In this case, it can be solved in linear time using strongly connected components and topological sort.

Another special case of interest is when each clause involves exactly *three* variables (3SAT). This will be fundamental in our study of NP-complete and NP-hard problems.

P vs **NP**

Question

Is it the case that *every* problem in **NP** is also in **P**?

- For example, is there a polynomial time algorithm to solve the general SAT problem?
- If so, given a propositional formula with n variables, then solving SAT, i.e.

  > *finding out whether any of the 2^n assignments of true/false to the variables satisfies the formula*

  would be not much harder than verifying a solution to SAT, i.e.

  > *checking whether a particular assignment of true/false to the variables satisfies the formula.*

P vs **NP**

- Your intuition may vary:
  - solving 2SAT is actually not much harder than verifying a solution to 2SAT, but
  - most people think that this does *not* apply to the general SAT problem.
- However, so far no one has been able to prove (or disprove) this, despite decades of effort by very many very famous people!
- The conjecture that **NP** is a strictly larger class of decision problems than **P** is known as the "$\mathbf{P} \neq \mathbf{NP}$" hypothesis, and it is widely considered to be one of the hardest open problems in mathematics.

Table of Contents

1. Decision Problems
1.1 Feasibility of Algorithms
1.2 Polynomial Reductions
1.3 Intractable Decision Problems

2. Optimisation Problems
2.1 Aside: Linear Programming
2.2 Intractable Optimisation Problems
2.3 Superpolynomial Algorithms
2.4 Approximation Algorithms

3. Puzzle

Reductions

- You have already used the technique of *problem solving by reduction*.

- This is when we map a new problem back to an old problem that we already know how to solve. For example:

    - instances of Balanced Partition can be mapped to instances of 0-1 Knapsack, and

    - instances of Budget Holiday I can be mapped to instances of Single Source Shortest Path Without Negative Edges.

- This gives an algorithm for the new problem, by applying the mapping function followed by the known algorithm for the old problem.

Reductions

Note

A mapping between two problems doesn't necessarily mean that the problems are equivalent.

Indeed, the mapping used in a reduction need not be surjective, i.e. we might only map to specific kinds of instances of the old problem.

Example

Instances of Balanced Partition map to instances of 0-1 Knapsack where all items have value equal to their weight.

The full generality of 0-1 Knapsack instances allows values unrelated to the weights.

Even so, any algorithm solving 0-1 Knapsack is also able to solve these special cases where $v_i = w_i$ for all i, from which we can infer an algorithm for Balanced Partition.

Polynomial Reductions

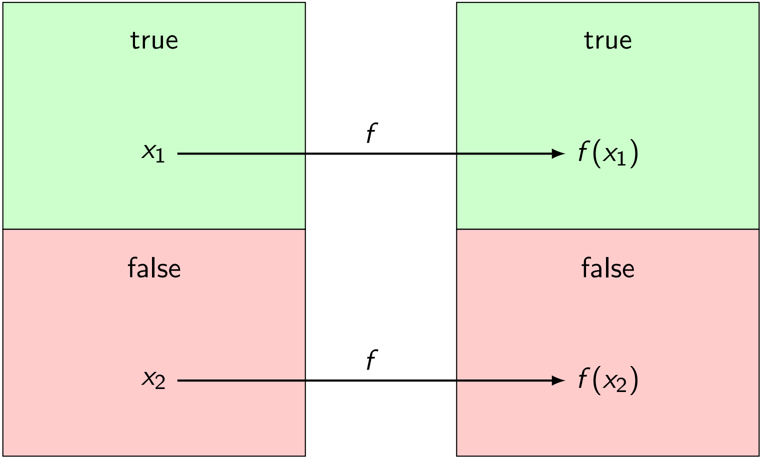
Definition

Let U and V be two decision problems. We say that U is polynomially reducible to V if and only if there exists a function $f(x)$ such that:

1. $f(x)$ maps instances of U into instances of V.
2. f maps YES instances of U to YES instances of V and NO instances of U to NO instances of V,

   i.e. $U(x)$ is YES if and only if $V(f(x))$ is YES.
3. $f(x)$ is computable by a polynomial time algorithm.

Polynomial Reductions



true

$x_1 \xrightarrow{f} f(x_1)$

false

$x_2 \xrightarrow{f} f(x_2)$

instances of U

instances of V

Polynomial Reductions

- If there is a polynomial reduction from U to V, we can conclude that U is *no harder than* V.
- If you could solve problem V in polynomial time, then problem U would also have a polynomial time solution.
  - For an instance x of U, you can instead solve the instance $f(x)$ of V, which has the same answer.
- The contrapositive is also true: if there is no (known) polynomial time algorithm for U, then there also can't be a (known) polynomial time algorithm for V.

Note

A reduction does not need to be *surjective*, that is, we might only map to specific kinds of instances of V.

Contrapositive

Definition

The *contrapositive* of the implication $p \implies q$ is $\neg q \implies \neg p$.

Example

"Students who enjoy puzzles look forward to the end of each module of Algorithms lectures" is logically equivalent to "Students who dread the end of each module of Algorithms lectures don't enjoy puzzles".

Note that these are not equivalent to "Students who don't enjoy puzzles dread the end of each module"! One might not care about the puzzle but have other reasons to look forward to the end of a module.

Contrapositive

Note

Instead of proving that if x is a NO instance, then $f(x)$ is a NO instance, we often prove the equivalent statement that if $f(x)$ is a YES instance, it must have been mapped from a YES instance x.

Example: Reduction of SAT to 3SAT

Claim

Every instance of SAT is polynomially reducible to an instance of 3SAT.

Proof Outline

We introduce more propositional variables and replace every clause by a conjunction of several clauses.

Example: Reduction of SAT to 3SAT

For example, we replace the clause

$$\underbrace{P_1 \vee \neg P_2} \vee \underbrace{\neg P_3} \vee \underbrace{P_4} \vee \underbrace{\neg P_5 \vee P_6} \tag{1}$$

with the following conjunction of "chained" 3-clauses with new propositional variables Q_1, Q_2, Q_3:

$$\begin{aligned} &(\underbrace{P_1 \vee \neg P_2} \vee Q_1) \wedge (\neg Q_1 \vee \underbrace{\neg P_3} \vee Q_2) \\ \wedge &(\neg Q_2 \vee \underbrace{P_4} \vee Q_3) \wedge (\neg Q_3 \vee \underbrace{\neg P_5 \vee P_6}) \end{aligned} \tag{2}$$

Easy to verify that if an evaluation of the P_i makes (1) true, then the corresponding evaluation of the Q_j also makes (2) true and vice versa: every evaluation which makes (2) true also makes (1) true. Clearly, (2) can be obtained from (1) using a simple polynomial time algorithm.

Cook's Theorem

Theorem

Every NP problem is polynomially reducible to the SAT problem.

- This means that for every NP decision problem U there is a polynomial time computable function f such that:
  1. for every instance x of U, $f(x)$ is a valid propositional formula for the SAT problem, and
  2. $U(x)$ is true if and only if the formula $f(x)$ is satisfiable.
- Why is this true? It's complicated! Proof is beyond the scope of this course.

NP-hard

Definition

A decision problem V is NP-*hard* ($V \in \mathbf{NP\text{-}H}$) if every other NP problem is polynomially reducible to V.

- Informal meaning: V is NP-hard if *it is at least as hard as everything in class* NP.
- Cook's Theorem says that SAT is NP-hard.

Misconception

An NP-hard problem might not be in class **NP** itself! The famous *halting problem* is one example.

NP-complete

Definition

A decision problem is NP-*complete* ($U \in \mathbf{NP\text{-}C}$) if it is in class **NP** and class **NP-H**.

- We saw earlier that SAT is in **NP**, and from Cook's theorem it is also NP-hard, so it is NP-complete.
- NP-complete problems are in a sense universal. If we had an algorithm which solves any NP-complete problem V, then we could also solve every other NP problem U by reduction:
  1. compute in polynomial time the reduction $f(x)$ of U to V,
  2. then running the algorithm that solves U on instance $f(x)$.
- A polynomial time algorithm for *any* NP-complete problem would imply that $\mathbf{P} = \mathbf{NP}$.

Complexity

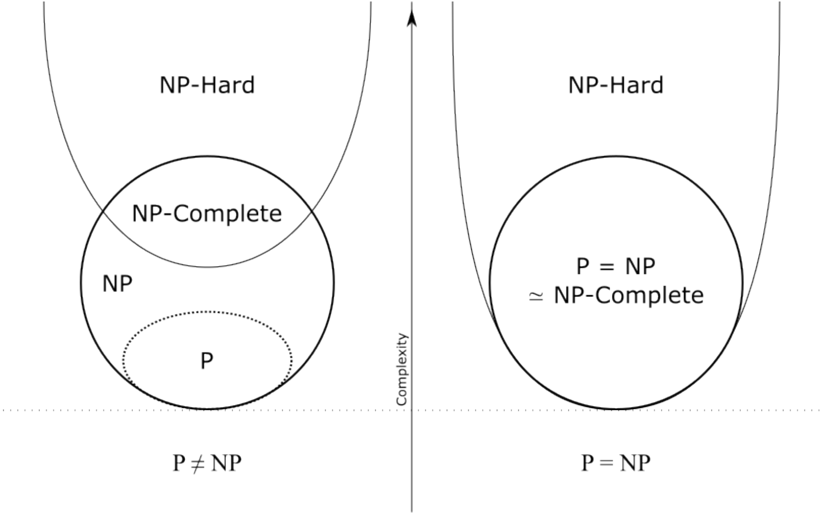

NP-Hard
NP-Complete
NP
P
P ≠ NP
Complexity
NP-Hard
P = NP
≃ NP-Complete
P = NP

Table of Contents

1. Decision Problems

1.1 Feasibility of Algorithms

1.2 Polynomial Reductions

1.3 Intractable Decision Problems

2. Optimisation Problems

2.1 Aside: Linear Programming

2.2 Intractable Optimisation Problems

2.3 Superpolynomial Algorithms

2.4 Approximation Algorithms

3. Puzzle

Why do we care about NP-complete problems?

- So NP-complete problems are the hardest problems in **NP** - a polynomial time algorithm for solving an NP-complete problem would make every other problem in **NP** also solvable in polynomial time.

- But if $\mathbf{P} \neq \mathbf{NP}$ (as is commonly hypothesised), then there cannot be any polynomial time algorithms for solving an NP-complete problem, not even an algorithm that runs in time $O(n^{1,000,000})$.

- So why bother with them?

Why do we care about NP-complete problems?

- Maybe SAT is not that important - why should we care about satisfiability of propositional formulas?

- Maybe NP-complete problems only have theoretical significance and no practical relevance?

- Unfortunately, this could not be further from the truth!

- A vast number of practically important decision problems are NP-complete!

Examples of NP-complete problems

Travelling Salesman Problem

Instance:

1. a map, i.e., a weighted directed graph with:
   - vertices representing locations
   - edges representing roads between pairs of locations
   - edge weights representing the lengths of these roads;
2. a number L.

Problem: Is there a tour along the edges which visits each location (i.e., vertex) exactly once and returns to the starting location, with total length at most L?

Think of a mailman who has to deliver mail to several addresses and then return to the post office. Can he do it while traveling less than L kilometres in total?

Examples of NP-complete problems

Register Allocation Problem

Instance:

1. an undirected unweighted graph G with:
   - vertices representing program variables
   - edges representing pairs of variables which are both needed at the same step of program execution;
2. the number of registers K of the processor.

Problem: is it possible to assign variables to registers so that no edge has both vertices assigned to the same register?

In graph theoretic terms: is it possible to color the vertices of a graph G with at most K colors so that no edge has both vertices of the same color?

Examples of NP-complete problems

Vertex Cover Problem

Instance:

1. an undirected unweighted graph G with vertices and edges;
2. a number k.

Task: is it possible to choose k vertices so that every edge is incident to at least one of the chosen vertices?

Examples of NP-complete problems

Set Cover Problem

Instance:

1. a number of items n;
2. a number of bundles m such that
   - each bundle contains of a subset of the items
   - each item appears in at least one bundle;
3. a number k.

Task: it it possible to choose k bundles which together contain all n items?

This problem can be extended by assigning a price to each bundle, and asking whether satisfactory bundles can be chosen within a budget b.

NP-complete problems

- We will see that many other practically important problems are also NP-complete.

- Be careful though: sometimes the distinction between a problem in **P** and a problem in **NP-C** can be subtle!

Examples in **P** vs examples in **NP-C**

Problems in **P**:

- Given a graph G and two vertices s and t, is there a simple path from s to t of length *at most* K?
- Given a propositional formula in CNF form such that every clause has at most *two* propositional variables, does the formula have a satisfying assignment? (2SAT)
- Given a graph G, does G have a tour where every *edge* is traversed exactly once? (Euler tour)

Problems in **NP-C**:

- Given a graph G and two vertices s and t, is there a simple path from s to t of length *at least* K?
- Given a propositional formula in CNF form such that every clause has at most *three* propositional variables, does the formula have a satisfying assignment? (3SAT)
- Given a graph G, does G have a tour where every *vertex* is visited exactly once? (Hamiltonian cycle)

Proving NP-completeness

Taking for granted that SAT is NP-complete, how do we prove NP-completeness of another NP problem?

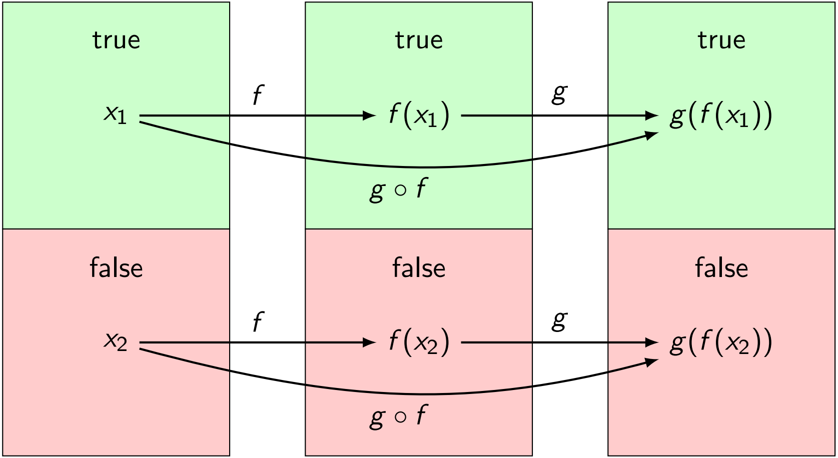
Theorem

Let V be an NP-complete problem, and let W be another NP problem. If V is polynomially reducible to W, then W is also NP-complete.

Proof Outline

- Let $g(x)$ be a polynomial reduction of V to W, and let U be any other NP problem.
- Since V is NP-complete, there is a polynomial reduction $f(x)$ of U to V.
- Then $(g \circ f)(x)$ is a polynomial reduction of V to W.

Polynomial Reductions



true

x_1 $\xrightarrow{f}$ $f(x_1)$ $\xrightarrow{g}$ $g(f(x_1))$

$g \circ f$

false

x_2 $\xrightarrow{f}$ $f(x_2)$ $\xrightarrow{g}$ $g(f(x_2))$

$g \circ f$

instances of U

instances of V

instances of W

Proving NP-completeness

Proof

We first claim that $(g \circ f)(x)$ is a reduction of U to W.

1. Since f is a reduction of U to V, $U(x)$ is true iff $V(f(x))$ is true.

2. Since g is a reduction of V to W, $V(f(x))$ is true iff $W(g(f(x)))$ is true.

Thus $U(x)$ is true iff $W(g(f(x)))$ is true, i.e., $(g \circ f)(x)$ is a reduction of U to W.

Proof (continued)

- Since $f(x)$ is the output of a polynomial time computable function, the length $|f(x)|$ of the output $f(x)$ can be at most a polynomial in $|x|$, i.e., for some polynomial (with positive coefficients) P we have $|f(x)| \leq P(|x|)$.
- Since $g(y)$ is polynomial time computable as well, there exists a polynomial Q such that for every input y, computation of $g(y)$ terminates after at most $Q(|y|)$ many steps.
- Thus, the computation of $(g \circ f)(x)$ terminates in at most:
  - $P(|x|)$ many steps, for the computation of $f(x)$, plus
  - $Q(|f(x)|) \leq Q(P(|x|))$ many steps, for the computation of $g(y)$ (where $y = f(x)$).
- In total, the computation of $(g \circ f)(x)$ terminates in at most $P(|x|) + Q(P(|x|))$ many steps, which is polynomial in $|x|$.

Proving NP-completeness

Proof (continued)

- Therefore $(g \circ f)(x)$ is a polynomial reduction of U to W.
- But U could be any NP problem!
- We have now proven that any NP problem is polynomially reducible to the NP problem W, i.e. W is NP-complete.

Example: Reducing 3SAT to VC

Problem

Prove that Vertex Cover (VC) is NP-complete by finding a polynomial time reduction from 3SAT to VC.

Outline

We will map each instance Φ of 3SAT to a corresponding instance $f(\Phi) = (G, k)$ of VC in polynomial time, and prove that:

1. if Φ is a YES instance of 3SAT, then $f(\Phi)$ is a YES instance of VC, and
2. if $f(\Phi)$ is a YES instance of VC, then Φ is a YES instance of 3SAT.

Note that this uses the earlier mentioned contrapositive.

Example: Reducing 3SAT to VC

Construction

Given an instance of 3SAT:

1. for each clause C_i, draw a triangle with three vertices v_1^i, v_2^i, v_3^i and three edges connecting these vertices;
2. for each propositional variable P_k draw a segment with vertices labeled as P_k and $\neg P_k$;
3. for each clause C_i connect the three corresponding vertices v_1^i, v_2^i, v_3^i with one end of the three segments corresponding to the variable appearing in C_i;
    - if the variable appears with a negation sign, connect it with the end labeled with the negation of that letter;
    - otherwise connect it with the end labeled with that letter.

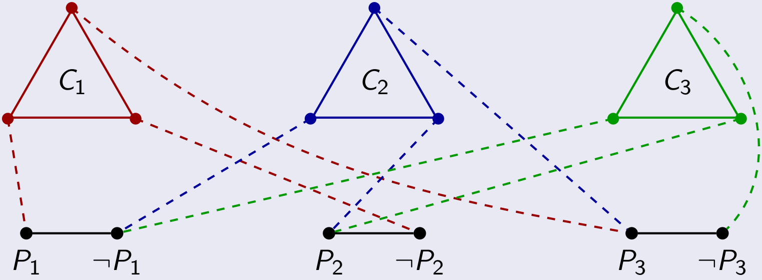
Example: Reducing 3SAT to VC

Example

Consider the propositional formula

$$(P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3).$$

The corresponding instance of Vertex Cover is:



Example: Reducing 3SAT to VC

Claim

An instance of 3SAT consisting of M clauses and N propositional variables is satisfiable if and only if the corresponding graph has a vertex cover of size at most $2M + N$.

Proof

Assume there is a vertex cover with at most $2M + N$ vertices chosen. Then

1. each triangle must have at least two vertices chosen, and
2. each segment must have at least one of its ends chosen.

This is in total $2M + N$ points; thus each triangle must have *exactly* two vertices chosen and each segment must have *exactly* one of its ends chosen.

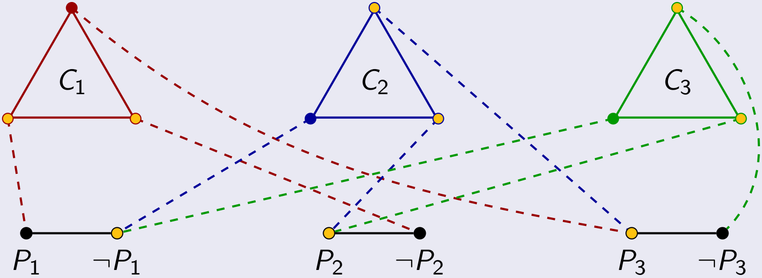
Reducing 3SAT to VC

Proof (continued)

Recall the example

$$(P_1 \lor \neg P_2 \lor P_3) \land (\neg P_1 \lor P_2 \lor P_3) \land (\neg P_1 \lor P_2 \lor \neg P_3)$$

and the corresponding VC instance.



Reducing 3SAT to VC

Proof (continued)

- Set each propositional letter P_i to true if the vertex labelled P_i (at the 'true' end of the ith segment) is covered.
- Otherwise, set a propositional letter P_i to false if $\neg P_i$ is covered.
- In a vertex cover of such a graph, every uncovered vertex of each triangle must be connected to a covered end of a segment, which guarantees that the clause corresponding to each triangle is true.

Reducing 3SAT to VC

Proof (continued)

- For the reverse direction, assume that the formula has an assignment of the variables which makes it true.
- For each segment, if it corresponds to a propositional letter P_i which such a satisfying evaluation sets to true cover its P_i end.
- Otherwise, if a propositional letter P_i is set to to false by the satisfying evaluation, cover its $\neg P_i$ end.
- For each triangle corresponding to a clause at least one vertex must be connected to a covered end of a segment, namely to the segment corresponding to the variable which makes that clause true; cover the remaining two vertices of the triangle.
- In this way we cover exactly $2M + N$ vertices of the graph and clearly every edge between a segment and a triangle has at least one end covered.

Table of Contents

Table of Contents

1. Decision Problems

1.1 Feasibility of Algorithms

1.2 Polynomial Reductions

1.3 Intractable Decision Problems

2. Optimisation Problems

2.1 Aside: Linear Programming

2.2 Intractable Optimisation Problems

2.3 Superpolynomial Algorithms

2.4 Approximation Algorithms

3. Puzzle

Salad

Problem

Instance: You want to make a salad of lettuce, tomato and onion. You know that:

- 100g of lettuce costs \$1.00, and gives 6 points of texture,
- 100g of tomato costs \$0.80, and gives 4 points of flavour, and
- 100g of onion costs \$0.60, and gives 1 point of flavour and 2 points of texture.

Task: Find an amount of each vegetable so that your salad has:

- weight exactly 300g,
- flavour at least 7, and
- texture at least 7,

at the cheapest possible cost.

Salad

- If we take x_L, x_T and x_O times 100g of each vegetable, we need
  - weight $x_L + x_T + x_O = 3$
  - flavour $4x_T + x_O \geq 7$, and
  - texture $6x_L + 2x_O \geq 7$.
- Subject to these restrictions, we want to minimise the cost $1\,x_L + 0.8\,x_T + 0.6\,x_O$.
- It turns out that the best salad has 110g lettuce, 170g tomato and 20g onion, but we will concentrate on the structure of such problems more than the process of solving them.

Linear Programming

- The restrictions (*constraints*) and cost (*objective function*) deal only with *linear* functions of the variables.

- This is an example of linear programming!

  - Linear programming problems are everywhere in industry.

  - Skills: recognise LP, formulate problem as LP.

- There are polynomial time algorithms for linear programming, such as the ellipsoid algorithm.

- The most popular algorithm for linear programming is the SIMPLEX algorithm, which is worst case exponential but has good average case performance.

Linear Programming

- Well optimised solvers available on the Web and in libraries for various programming languages.

- The variables are typically assumed to be non-negative.[1]

[1] An unconstrained variable x can be simulated by the difference between two non-negative variables x' and x^*.

Linear Programming

- $\geq$ constraints can be negated to make $\leq$ constraints, e.g.

$$4x_T + x_O \geq 7 \quad \rightarrow \quad -4x_T - x_O \leq -7.$$

- Equality constraints can be represented by a pair of inequality constraints, e.g.

$$x_L + x_T + x_O = 3 \quad \rightarrow \quad \begin{cases} x_L + x_T + x_O \leq 3 \\ -x_L - x_T - x_O \leq -3 \end{cases}$$

- Minimisation problems can be converted to maximisation by negating the objective function, e.g.

$$\min\left[1\,x_L + 0.8\,x_T + 0.6\,x_O\right] \rightarrow \max\left[-1\,x_L - 0.8\,x_T - 0.6\,x_O\right].$$

Linear Programming

- We have already seen an example of linear programming in the maximum flow problem.
- The variables are the amounts of flow in each edge.
- There are $|E|$ inequality constraints arising from the capacity constraint.
- There are $|V|$ equality constraints arising from flow conservation, each represented as a pair of inequalities.
- The objective is to maximise the sum of flows on edges leaving the source.
- The particular structure of these constraints permit efficient algorithms such as Edmonds-Karp.

Salad II

- Suppose now that the vegetables come in pre-portioned 100g packets, and to avoid waste you have decided to use the entire contents of any packet that you open.

- How does this change the problem?

- Now the variables x_L, x_T and x_O are *integer*-valued rather than real-valued!

- This is an *integer* linear programming problem.

Integer Linear Programming

Question

Does the change from real-valued variables to integer-valued variables make the problem easier or harder (or neither)?

- It might seem to make the problem easier, as the search space now consists of only the points with integer coordinates, not all the points in between.
- However, integer linear programming is actually *much harder* than linear programming with only real-valued variables!
- There is no known polynomial-time algorithm for integer linear programming.

Integer Linear Programming

- We have already seen an example of integer linear programming in the integer knapsack problem.
- The variables are the number of times each item is taken.
- There is one inequality constraint arising from the knapsack capacity.
- The objective is to maximise the total value.

Exercise

What additional constraints must be included to model 0-1 Knapsack as an integer linear programming problem?

Table of Contents

1. Decision Problems

1.1 Feasibility of Algorithms

1.2 Polynomial Reductions

1.3 Intractable Decision Problems

2. Optimisation Problems

2.1 Aside: Linear Programming

2.2 Intractable Optimisation Problems

2.3 Superpolynomial Algorithms

2.4 Approximation Algorithms

3. Puzzle

Intractable optimisation problems

- We have already seen many *decision problems* for which we don't know of a polynomial time solution.
- Not every problem is a decision problem; often we need to maximise or minimise some quantity.
- Example:
  - decision problem: "Is there a path of length $\leq L$?"
  - optimisation problem: "What is the length of the shortest path?"
- We saw in Module 2 that the optimisation problem is no easier than the corresponding decision problem, and sometimes much harder (as there is an extra variable).

Example: Travelling Salesman Problem (Optimisation)

Problem

Instance:

1. a map, i.e., a weighted graph with:
   - vertices representing locations
   - edges representing roads between pairs of locations
   - edge weights representing the lengths of these roads;

Problem: find a tour along the edges which visits each location (i.e., vertex) exactly once and returns to the starting location, with *minimal* total length?

Think of a courier who has to deliver mail to several addresses and then return to the post office. How can they do it while traveling the minimum total distance?

Intractable optimisation problems

- The Traveling Salesman Optimisation Problem is clearly no easier than the corresponding decision problem.

- If we could solve the optimisation version, we could then solve the decision version in constant extra time by comparing the length of the shortest path to the length L being tested.

- We don't know of any polynomial time algorithm for TSP (Decision), so we certainly don't know of any polynomial time algorithm for TSP (Optimisation).

The significance of intractable optimisation problems

- It is important to be able to figure out if a problem at hand is intractable in order to know that one has to abandon trying to come up with a feasible (i.e., polynomial time) solution.

- So what do we do when we encounter an intractable problem?

- If this problem is an optimisation problem, we can try to:

    - solve it exactly with a slower-than-polynomial time algorithm (which might still be substantially better than brute force), or

    - solve it approximately with an efficient (often greedy) algorithm (so long as the approximation is reasonably close to optimal).

How to make progress on intractable optimisation problems

- Thus, for a practical problem which appears to be intractable, the strategy would be:
    - confirm that the problem is indeed intractable, and
    - look for an exact algorithm which provides an answer in slower than polynomial time, or
    - look for an approximation algorithm which provides an answer that is always within an acceptable factor of the optimum.

Table of Contents

1. Decision Problems
1.1 Feasibility of Algorithms
1.2 Polynomial Reductions
1.3 Intractable Decision Problems

2. Optimisation Problems
2.1 Aside: Linear Programming
2.2 Intractable Optimisation Problems
2.3 Superpolynomial Algorithms
2.4 Approximation Algorithms

3. Puzzle

Superpolynomial algorithms

- Even if a problem is intractable, we don't necessarily have to abandon our usual techniques for algorithm design.
- In a practical scenario, there are often degrees to the intractability of a problem.
  - The brute force might run for an unreasonably long time for all but the smallest instances, whereas a faster (but still slower than polynomial) algorithm might still be viable for moderately small instances.
- Even if we can't find a polynomial time algorithm, there are sometimes significant benefits to:
  - reducing the index of an exponential term, or
  - changing from factorial time to exponential time.

.

Subset Sum

Problem

Instance: an array of n positive integers, and a positive integer S.

Task: find the maximum sum of a subset with sum $\leq S$.

Example

If the array is $[2, 5, 6, 8]$ and the target is 12, then the answer is 11 (from $5 + 6$).

Subset Sum

- A brute force algorithm for this problem would try every possible subset.
- There are two choices for each number (include it or exclude it in the subset), for a total of 2^n subsets.
- We then have to add up each subset, for a total time complexity of $O(2^n n)$.
- Early exit doesn't help in the worst case.

Question

Can we do better? Smaller exponent perhaps?

Subset Sum

- Some redundancy!
  - Each set of choices from the first half of the array is represented in $2^{n/2}$ different subsets.
  - After a small change (or indeed any change) in the first half, we will again go through $2^{n/2}$ corresponding subsets for the various selections from the second half.
  - Solve each half separately?
- We can try adding up every subset of the first half, and separately every subset of the second half, and look for a pair which matches.
  - Given two arrays, find a pair which adds to as close to S as possible.
  - We've seen similar problems before!
- This technique is called *meet in the middle*.
  - It might remind you of divide and conquer, but note that we don't recurse any deeper.

Subset Sum

Solution

1. Compute all the sums of subsets of the first half and the second half of the array by brute force, and store them in arrays A and B.

2. Sort array B using mergesort.

3. For every entry of array A, say $A[i]$, find the lower bound for $S - A[i]$ in array B,[2] using binary search.

4. Update the tentative answer to $A[i] + B[j]$.

[2] Recall this is the largest entry $B[j]$ so that $B[j] \leq S - A[i]$, i.e. $A[i] + B[j] \leq S$.

Subset Sum

- Clearly correct; every subset of the array combines a subset of each half.

- Time complexity:

    - Computing arrays A and B takes $O(2^{n/2}\,n)$.

    - Array B has $2^{n/2}$ entries, so mergesort runs in $O(2^{n/2}\,n)$.

    - For each of $2^{n/2}$ entries of A, we run a binary search in $O(\log 2^{n/2}) = O(n)$ time.

- Therefore the total time complexity is $O(2^{n/2}\,n)$.

Subset Sum

- Our new complexity $O(2^{n/2}\,n)$ is still exponential, but about the square root of the brute force complexity $O(2^n n)$.
- For example, assuming 10^9 operations per second, and that the constant factors are not enormous,
  - the brute force will take *hours* to solve an instance where $n = 40$, whereas
  - our new algorithm will take less than a second!
- Still doesn't scale well: number of operations is *squared* when we go up to $n = 80$.
- Trade-off: we need to store all the sums from each half of the array, using $O(2^{n/2})$ memory instead of the brute force's $O(n)$.

Travelling Salesman Problem

Problem

Instance: a weighted graph $G = (V, E, w)$, where $|V| = n$ and $|E| = m$.

Task: find the minimum weight of a tour which visits each vertex exactly once and returns to the starting vertex.

Travelling Salesman Problem

- Brute force would try all orders in which we could visit the vertices.
  - n choices for the first vertex, $n-1$ for the second, $n-2$ for the third and so on.
  - $n!$ orders.
  - For each order, we also need to sum up the edge weights in $O(n)$ time.
  - Total time complexity is $O(n!\,n)$.

Question

Can we do better? Exponential time perhaps?

Travelling Salesman Problem

- We solved some shortest path problems using dynamic programming. Let's try that again here.
- Subproblems are shortest *partial paths*.
- For a partial path $[v_1, v_2, \ldots, v_{k-1}, v_k]$ to be valid, we need to ensure that the new vertex v_k isn't one that we've seen already, i.e.

$$v_k \notin \{v_1, v_2, \ldots, v_{k-1}\}.$$

- Do we really need the whole *sequence* in which those vertices were visited?
    - Maybe we just need to know which vertices were visited, not the order.

Travelling Salesman Problem

Notation

Let $s \in V$ be a designated vertex. We will insist that TSP tours start and end at s.

Solution

Subproblems: for $S \subseteq V$ containing s, and $v \in S$, let $P(S, v)$ be the problem of determining $\mathrm{opt}(S, v)$, the length of the shortest path starting at s, visiting the vertices of S once each and finishing at vertex v.

Recurrence: for $v \neq s$, we have

$$\mathrm{opt}(S, v) = \min_{\substack{u \in S \\ (u,v) \in E}} \left[\mathrm{opt}(S \setminus \{v\}, u) + w(u, v)\right].$$

Travelling Salesman Problem

Solution

Base case:

$$\text{opt}(S, s) = \begin{cases} 0 & \text{if } S = \{s\} \\ \infty & \text{otherwise.} \end{cases}$$

Order of computation: as $P(S, v)$ depends only on $P(S', u)$ where $|S'| = |S| - 1$, we can solve the subproblems in increasing order of $|S|$ to ensure all dependencies are respected.

Overall answer: considering all choices of the penultimate vertex plus the edge that completes the tour, we have

$$\min_{v \neq s} \left[\text{opt}(V, v) + w(v, s)\right].$$

Travelling Salesman Problem

Solution

Time complexity: $O(2^n\, n)$ subproblems each taking $O(n)$ time, for a total of $O(2^n\, n^2)$.

We can slightly improve this to $O(2^n\, m)$ by more carefully counting how many times each edge is used.

Travelling Salesman Problem

- Our new complexity $O(2^n\, m)$ is exponential, whereas the brute force took factorial time $O(n!\, n)$.
- For example, assuming 10^9 operations per second, and that the constant factors are not enormous,
    - the brute force will take *centuries* to solve an instance where $n = 20$, whereas
    - our new algorithm will take less than a second!
- Still doesn't scale well: number of operations is *squared* when we go up to $n = 40$.
- Trade-off: we need to store the DP table, using $O(2^n\, n)$ memory instead of the brute force's $O(n)$.

Table of Contents

1. Decision Problems

1.1 Feasibility of Algorithms

1.2 Polynomial Reductions

1.3 Intractable Decision Problems

2. Optimisation Problems

2.1 Aside: Linear Programming

2.2 Intractable Optimisation Problems

2.3 Superpolynomial Algorithms

2.4 Approximation Algorithms

3. Puzzle

Minimum Vertex Cover

Problem

Instance: an undirected unweighted graph $G = (V, E)$.

Task: find the smallest number of vertices so that every edge is incident to at least one of the chosen vertices.

Approximate Minimum Vertex Cover

Algorithm

1. Pick an arbitrary edge and cover *both* of its ends.
2. Remove all the edges whose one end is now covered. In this way you are left only with edges which have either both ends covered or no end covered.
3. Continue picking edges with both ends uncovered until no edges are left.

Approximate Minimum Vertex Cover

- This certainly produces a vertex cover, because edges are removed only if one of their ends is covered, and we perform this procedure until no edges are left.

- The number of vertices covered is equal to twice the number of edges with both ends covered.

- But the minimal vertex cover must cover at least one vertex of each such edge.

- Thus we have produced a vertex cover of size at most twice the size of the minimal vertex cover.

Metric Travelling Salesman Problem

Problem

Instance: a complete weighted graph G with weights $d(i,j)$ of edges (to be interpreted as distances) satisfying the “triangle inequality”: for any three vertices i,j,k, we have

$$d(i,j) + d(j,k) \geq d(i,k).$$

Claim

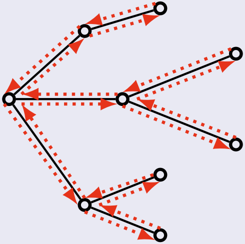
Metric TSP has an approximation algorithm producing a tour of total length at most twice the length of the optimal (i.e. minimal) length tour, which we will denote by *opt*.

Approximate Metric TSP

Algorithm

Find a minimum spanning tree T of G. Since the optimal tour with one of its edges e removed represents a spanning tree, we have that the total weight of T satisfies $w(T) \leq opt - w(e) \leq opt$.

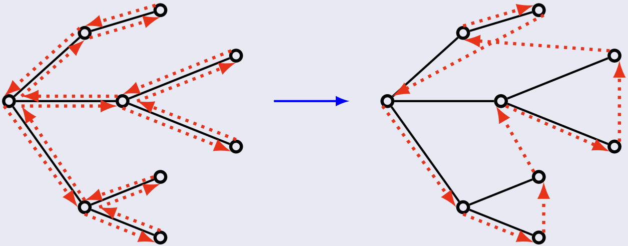
If we do a depth first traversal of the tree, we will travel a total distance of $2w(T) \leq 2opt$.



Approximate Metric TSP

Algorithm (continued)

We now take shortcuts to avoid visiting vertices more than once; because of the triangle inequality, this operation does not increase the length of the tour.



Approximating NP-hard optimisation problems

- All NP-complete problems are equally difficult, because any of them is polynomially reducible to any other.
- However, the related optimisation problems can be very different!
- For example, we have seen that some of these optimisation problems allow us to get within a constant factor of the optimal answer.
  - Vertex Cover permits an approximation which produces a cover at most twice as large as the minimum vertex cover.
  - Metric TSP permits an approximation which produces a tour at most twice as long as the shortest tour.

Approximating the general TSP

- On the other hand, the most general Travelling Salesman Problem does not allow any approximate solution at all: if **P** $\neq$ **NP**, then for no $K > 1$ can there be a polynomial time algorithm which for every instance produces a tour which is at most K times the length of the shortest tour!
- To prove this, we show that if for some $K > 0$ there was indeed a polynomial time algorithm producing a tour which is at most K times the length of the shortest tour, then we could obtain a polynomial time algorithm which solves the Hamiltonian Cycle problem.
- This is the problem of determining for a graph G whether G contains a cycle visiting all vertices exactly once. It is known to be NP-complete.

Approximating the general TSP

- Let G be an arbitrary unweighted graph with n vertices.

- We turn this graph into a complete weighted graph G^* by setting the weights of all existing edges to 1, and then adding edges of weight $K \cdot n$ between the remaining pairs of vertices.

- If an approximation algorithm for TSP exists, it produces a tour of all vertices with total length at most $K \cdot opt$, where opt is the length of the optimal tour through G^*.

Approximating the general TSP

- If the original graph G has a Hamiltonian cycle, then G^* has a tour consisting of edges already in G and of weights equal to 1, so such a tour has length of exactly n.

- Otherwise, if G does not have a Hamiltonian cycle, then the optimal tour through G^* must contain at least one added edge of length $K \cdot n$, so

$$opt \geq (K \cdot n) + (n - 1) \cdot 1 > K \cdot n.$$

Approximating the general TSP

- Thus, our approximate TSP algorithm either returns:
  - a tour of length at most $K \cdot n$, indicating that G has a Hamiltonian cycle, or
  - a tour of length greater than $K \cdot n$, indicating that G does not have a Hamiltonian cycle.
- If this approximation algorithm runs in polynomial time, we now have a polynomial time decision procedure for determining whether G has a Hamiltonian cycle!
- This can only be the case if $\mathbf{P} = \mathbf{NP}$.

Table of Contents

1. Decision Problems

1.1 Feasibility of Algorithms

1.2 Polynomial Reductions

1.3 Intractable Decision Problems

2. Optimisation Problems

2.1 Aside: Linear Programming

2.2 Intractable Optimisation Problems

2.3 Superpolynomial Algorithms

2.4 Approximation Algorithms

3. Puzzle

Puzzle

Problem

You are given a coin, but you are not guaranteed that it is a fair coin. It may be biased towards either heads or tails.
Use this coin to simulate a fair coin.

Hint

Try tossing the biased coin more than once!

That's All, Folks!!