9. INTRACTABILITY

Raveen de Silva, r.desilva@unsw.edu.au
office: K17 202
Course Admin: Anahita Namvar, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 2, 2022

Table of Contents

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.

This means that there exists a natural number k (independent of the input) so that the algorithm terminates in $T(n) = O(n^k)$ many steps, where n is the size of the input.

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.

- As we will see, 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.

- This can only change the constants involved in the expression $T(n) = O(n^k)$ but not the asymptotic bound.

Length of Input: Weighted Graphs

- If the input is a weighted graph G, 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.

- Alternatively, we can represent G with its adjacency matrix.
- If the input graphs are all sparse, this can unnecessarily increase the length of the representation of the graph.

- However, since we are interested only in whether the algorithm runs in polynomial time and not in the particular degree of the polynomial bounding such a run time, this does not matter.
- In fact, every precise description 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 **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 exists 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 a *certificate* for x.

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: Vertex Cover

Vertex Cover

Instance: a graph G and an integer k.

Problem: "There exists a subset U consisting of at most k vertices of G (called a vertex cover of G) such that each edge has at least one end belonging to U."

- Clearly, given a subset of vertices U we can determine in polynomial time whether U is a vertex cover of G with at most k elements.
- So Vertex Cover is in class **NP**.

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?

- The existence of such an algorithm would mean that finding out whether a propositional formula evaluates true in any of the 2^n cases for its variables' values is actually not much harder than simply checking one of these cases.

- Intuitively, this should not be the case; determining if such a case exists should be a harder problem than simply checking a particular case.

P vs NP

- 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

Polynomial Reductions

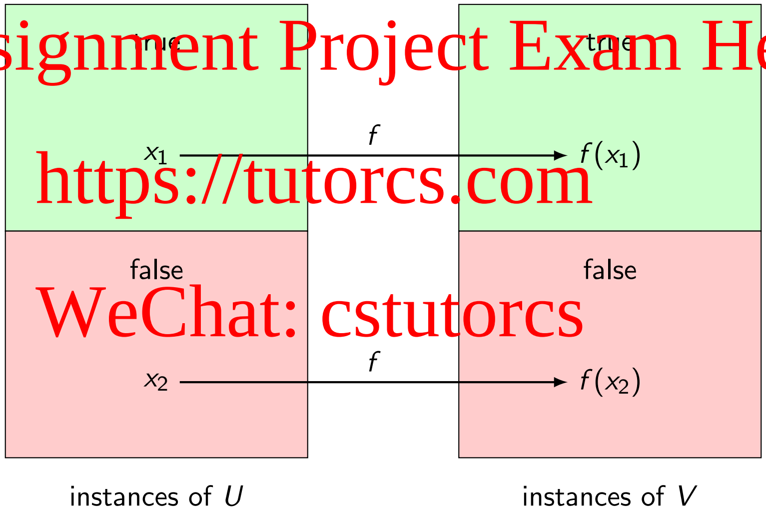
Definition

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



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 Algorithms lecture" is logically equivalent to "Students who dread the end of each Algorithms lecture don't enjoy puzzles".

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 $U(x)$ there exists a polynomial time computable function $f(x)$ such that:

1. for every instance x of U, $f(x)$ produces a propositional formula Φ_x;
2. $U(x)$ is true if and only if Φ_x is satisfiable.

NP-complete

Definition

An NP decision problem U is NP-*complete* ($U \in \mathbf{NP\text{-}C}$) if every other NP problem is polynomially reducible to U.

- Thus, Cook's Theorem says that SAT is NP-complete.
- NP-complete problems are in a sense universal: if we had an algorithm which solves any NP-complete problem U, then we could also solve every other NP problem as follows.
- A solution of an instance x of any other NP problem V could simply be obtained by:
  1. computing in polynomial time the reduction $f(x)$ of V to U,
  2. then running the algorithm that solves U on instance $f(x)$.

Why do we care about NP-complete problems?

- So NP-complete problems are the hardest NP problems – a polynomial time algorithm for solving an NP-complete problem would make every other NP problem 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!

Traveling 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: it 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 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?

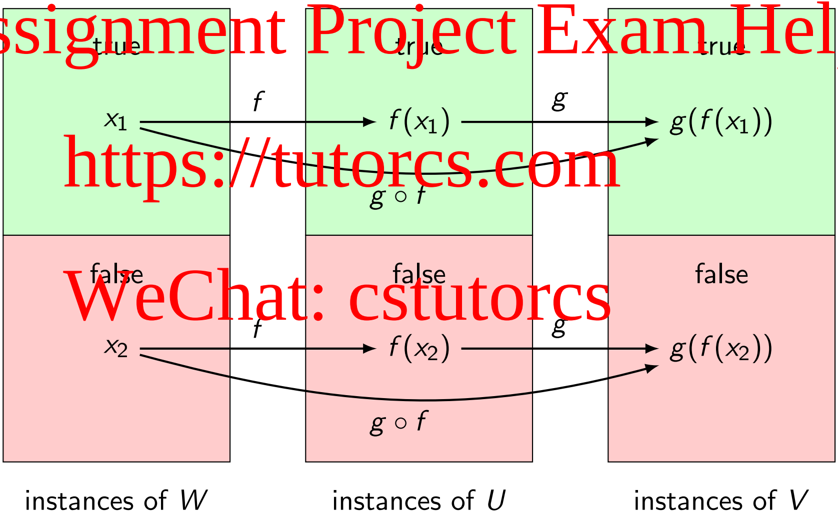
Theorem

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

Proof

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

Polynomial Reductions



Proving NP-completeness

Proof (continued)

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

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

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

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

Proving NP-completeness

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 W to V.

- But W could be any NP problem!

- We have now proven that any NP problem is polynomially reducible to the NP problem V, i.e. V 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.

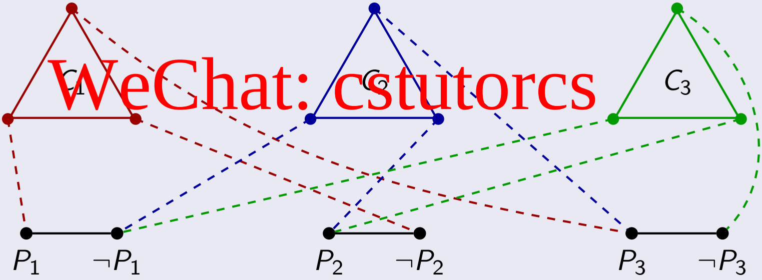
Example: Reducing 3SAT to VC

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.

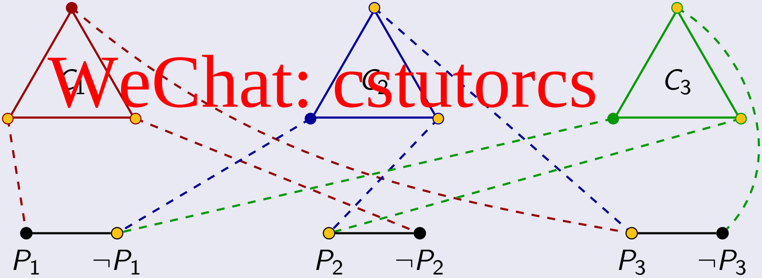
Reducing 3SAT to VC

Proof (continued)

Recall the example:

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

and the corresponding VC instance.



Reducing 3SAT to VC

Proof (continued)

- Set each propositional letter P_i to true if P_i end of the segment corresponding to P_i 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

NP-hard problems

- Let A be a problem and suppose we have a "black box" device which for every input x instantaneously computes $A(x)$.
- We consider algorithms which are *polynomial time in A*. This means algorithms which run in polynomial time in the length of the input and which, besides the usual computational steps, can also use the above mentioned "black box".

Definition

We say that a problem A is *NP-hard* ($A \in \textbf{NP-H}$) if every NP problem is polynomial time in A, i.e., if we can solve every NP problem U using a polynomial time algorithm which can also use a black box to solve any instance of A.

Note that we do NOT require that A be an NP problem; we even do not require it to be a decision problem - it can also be an optimisation problem.

An example of an NP-hard problem

Traveling Salesman 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 mailman who has to deliver mail to several addresses and then return to the post office. How can he do it while traveling the minimum total distance?

NP-hard problems

- The Traveling Salesman Optimisation Problem is clearly NP-hard: using a "black box" for solving it, we can solve the Traveling Salesman Decision problem.
- That is, given a weighted graph G and a number L we can determine if there is a tour containing all vertices of the graph and whose length is at most L.
- We simply invoke the black box for the Traveling Salesman Optimisation Problem, which gives us the length of a shortest tour, and compare this length with L.
- Since the Traveling Salesman Decision Problem is NP-complete, all other NP problems are polynomial time reducible to it.
- Therefore every other NP problem is solvable using a "black box" for the Traveling Salesman Optimisation Problem.

The significance of NP-hard problems

- It is important to be able to figure out if a problem at hand is NP-hard 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 NP-hard problem?
- If this problem is an optimisation problem, we can try to solve it in an approximate sense by finding a solution which might not be optimal, but it is reasonably close to an optimal solution.

- For example, in the case of the Traveling Salesman Optimisation Problem we might look for a tour which is at most twice the length of the shortest possible tour.

How to make progress on NP-hard problems

- Thus, for a practical problem which appears to be infeasible, the strategy would be:

  - prove that the problem is indeed NP-hard, to justify not trying solving the problem exactly;

  - look for an approximation algorithm which provides a feasible sub-optimal solution that it is not too far from optimal.

Example: 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.

Example: 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.

Example: Metric Traveling Salesman Problem (MTSP)

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

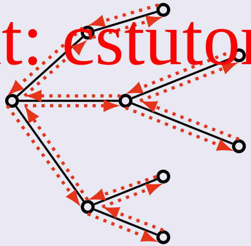
MTSP has an approximation algorithm producing a tour of total length at most twice the length of the optimal (i.e. minimal) length tour, which we will denote by *opt*.

Example: Metric Traveling Salesman Problem (MTSP)

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 $w(T) \leq opt - w(e) \leq opt$.

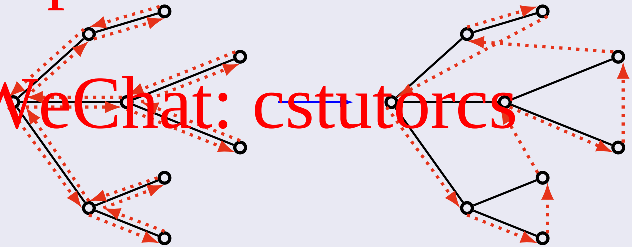
If we do a depth first traversal of the tree, we will travel a total distance of $2w(T) \leq 2opt$.



Example: Metric Traveling Salesman Problem (MTSP)

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 Traveling Salesman Problem does not allow any approximate solution at all: if $\mathbf{P} \neq \mathbf{NP}$, 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

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!!