

1 Problems in P and NP

All of the algorithms we have considered so far run in time that grows polynomially with the size of the input. Formally, these algorithms are all $O(n^c)$ for some constant c , where n is the size of the input. From a general point of view, running in polynomial time is a minimal requirement for calling an algorithm “efficient,” and the smaller we make c , the more efficient we say the algorithm is.

Not all problems can be solved in polynomial time, however. For instance, some problems require at least exponential time, running in $O(c^n)$ time, for some constant c , and some problems require only poly-logarithmic time, running in $O(\log^c n)$ time. One important class of problems are those that are **NP-hard**, which are problems that *we believe cannot be solved* in polynomial time, even though no one can prove a super-polynomial lower bound.

In order to make these ideas concrete, we will only consider different types of *decision problems*. This kind of problem is one whose output is simply a boolean variable, YES or NO.¹ Here are three important types of such decision problems:

- **P**: the set of decision problems that can be solved in polynomial time.
- **NP**: the set of decision problems such that if the answer is YES, then there exists a *witness* (succinctly: a proof) of that fact that can be checked in polynomial time.
- **co-NP**: the set of decision problems such that if the answer is NO, then there exists a witness of that fact that can be checked in polynomial time.

Every decision problem that is in P is also in NP, because for every problem in P, we can verify YES answers by simply computing the answer from scratch, in polynomial time. And, for the same reason, every problem in P is also in co-NP.

What does the “N” in NP stand for? It stands for “non-deterministic,” which is a mathematical term meaning that our theoretical computer will non-deterministically² arrive at a solution, by making an arbitrary sequence of choices. In this case, that solution is the witness.

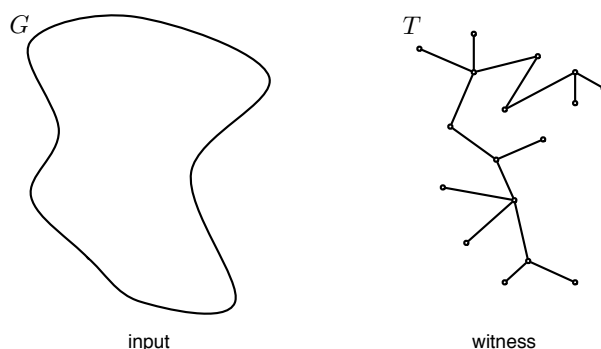
For example. All of the problems we have seen in this class can be phrased as decision problems. The following table gives several examples from the lectures, listing the generic problem name, its input, its YES-NO decision question, and the form of the witness. Strictly speaking, verifying that a particular witness w is correct also requires verifying that w is a witness to our particular input. For all of the problems listed below, although time consuming, this step takes time only polynomial in n .

¹This is in contrast to an *optimization problem*, in which we seek to return the solution, among all possible solutions, that minimizes or maximizes some criteria, e.g., maximum flow, minimum spanning tree, etc.

²Not to be confused with “not deterministic,” which refers randomized algorithms.

Problem	Input	Decision	Witness
Sorting	an array of elements A	is there a permutation of A with no more than k out-of-order elements?	a permutation of A
Interval Scheduling	a set of requests R and starting/ending functions s and f	is there a compatible set of requests S of size at least k ?	a compatible set S
String Encoding	set of input-string symbol frequencies F	is there an encoding C of F such that the encoded string has length $\ell \leq k$?	a codebook C
Knapsack	set of items B , capacity W , value function $v : b \rightarrow \mathbb{R}$ and weight function $w : b \rightarrow \mathbb{R}$	is there a set of items S with weight $w(S) \leq W$ and value $v(S) \geq k$?	a set of items S
String Alignment	strings A and B of lengths n and m	is there an alignment T of A and B with cost(T) $\leq k$?	an alignment T
Short Paths	directed graph $G = (V, E)$, vertex $s \in V$, and weight function $w : e \rightarrow \mathbb{R}$	is there a spanning tree T rooted at s with weight $w(T) \leq k$?	a spanning tree T
Weighted Spanning Tree	undirected graph $G = (V, E)$ and weight function $w : e \rightarrow \mathbb{R}$	is there a spanning tree T with weight $w(T) \leq k$?	a spanning tree T
Flow	directed graph $G = (V, E)$ and capacity function $c : V \times V \rightarrow \mathbb{R}_{\geq 0}$	is there a feasible flow f with value $ f \geq k$?	a flow f
Bipartite Matching	bipartite graph $G = (A \cup B, E)$ for $E : A \times B$, and weight function $w : e \rightarrow \mathbb{R}$	is there a matching S with weight $w(S) \geq k$?	a matching S
Stable Matching	sets M and W of individuals, where $ W = M = n$, and preferences lists $p_{m \in M}$, $p_{w \in W}$ which are total orderings of the sets W and M respectively	is there a matching of M to W with fewer than k unstable pairs?	a matching S

To be concrete, let us take the specific problem of finding a weighted spanning tree. Here, the input is an undirected graph $G = (V, E)$ and a weight function on the edges $w : e \rightarrow \mathbb{R}$. The decision question is whether there exists a spanning tree T with weight $w(T) \leq k$, where k is a constant. Suppose we (non-deterministically) identify a candidate spanning tree T . How can we verify that $w(T) \leq k$?



Simple. Let T be stored as an adjacency list. For each edge $e \in T$, we add its weight $w(e)$ to an accumulator to compute $w(T) = \sum_{e \in T} w(e)$. By assumption, a call to $w(\cdot)$ takes $O(1)$ time, and we make $|T| = |V| - 1 = n - 1$ calls to compute $w(T)$. Comparing this value with our specified constant k also takes $O(1)$ time. (Finally, we may wish to check that each edge $e \in T$ is also some edge $e \in E$. Even if we are inefficient about how we do this, it takes at worst $|T| \times |E| = O(n^3)$ time. Do you see how we could be more efficient in this step?) Thus, verifying that T is a YES takes time polynomial in the size of the input.

1.1 P vs. NP, and other questions

The crucial difference between P and NP is the difference between solving a problem yourself versus checking that someone else's solution is correct. Perhaps the single most important open question in theoretical computer science, and possibly all of computer science, is whether the classes P and NP are actually different.

An answer to this “P vs. NP” question, i.e., whether $P=NP$ or $P \neq NP$, would have major implications to other branches of science, as well. But, no one knows whether they are the same or different.³ The Clay Mathematics Institute lists P vs. NP as the first of its Millennium Prize Problems, each of which comes with a \$1,000,000 prize for a correct answer. If you could prove that $P=NP$ and produce an algorithm for solving the hard problems in NP, however, more than half of the remaining Millennium Prize Problems would become immediately solvable. Although we do

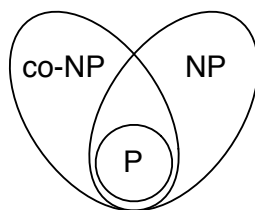
³And many have tried. History is littered with false claims one way or another, and new claims are made every year. In fact, it has been proved that deciding whether $P=NP$ is itself a hard question.

not have a proof of the fact, it is generally believed that $P \neq NP$, but nobody knows how to prove it (any many many have tried).

Intuitively, it certainly *seems* that P and NP are different. Solving a problem from scratch seems and often is difficult. Verifying that someone else's solution is correct seems easier. But the fact that we do not know if $P=NP$ means that we do not, in fact, know whether these two tasks, which seem so different, actually are.

A related, and perhaps more subtle open question is whether $NP=co-NP$. That is, even if we can quickly verify that YES answers are correct, it does not follow that we can also quickly verify that NO answers are correct. This may seem maddening, but it is an accurate statement: as far as we know, there is no short (polynomial in length) proof or witness that a problem like Traveling Salesman is not satisfiable on a given graph. As a simple analogy, if the YES answer is a needle hiding in a haystack, then I can prove to you that the answer is YES by showing you the needle. But, to prove that the haystack contains NO needles, I have to check all the hay, which could take exponential time.

As with P vs. NP , it is generally believed that $NP \neq co-NP$, but, like P vs. NP , no one knows how to prove it. Here is what we think are the relationships between P , NP , and $co-NP$, although we do not know the proofs:

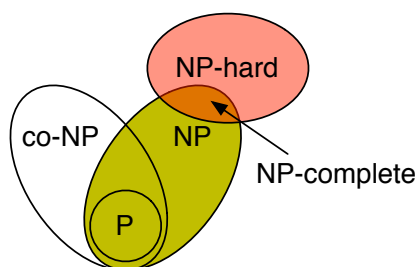


NP-hard and NP-complete problems. Let us define a problem Q to be **NP-hard** if a polynomial-time algorithm for Q would imply a polynomial-time algorithm for all problems in NP . In this case, a polynomial-time algorithm for Q would imply $P=NP$. Thus, NP-hard problems are at least as hard as any problem in NP .

A corollary is that the NP-hard set of problems may include some problems that are not, themselves in NP . But, if a problem is NP-hard *and* is a member of NP , then we call it **NP-complete**, which makes it part of the set of hardest problems in NP . That is, a polynomial-time solution to even one NP-complete problem would imply a polynomial-time algorithm to every NP-complete problem. At the time of this writing, thousands of real and interesting problems have been shown

to be NP-complete,⁴ including several related to the Clay Mathematics Institute’s Millennium Prize Problems. Crucially, a polynomial-time solution any one of these problems would yield a polynomial-time solution to all of them,⁵ which makes them the most important problems in NP.

Here is another picture of what we think are the relationships between these classes of problems.



How do we know that any problem is in the NP-complete set? Because we have at least one example of a problem that has been proved to be in the NP-complete set:

The Cook-Levin Theorem: Boolean Satisfiability is NP-complete.

To be clear, Boolean Satisfiability is the problem of deciding if a string of boolean variables, chained together with ANDs, ORs and NOTs, has any setting that yields a YES when the operations are carried out.⁶ The fact that Boolean Satisfiability (or simply SAT) is NP-complete means that all problems in the NP-complete set can be *reduced*, in polynomial time,⁷ to SAT. Thus, if we want to show that some new problem Q is NP-complete, we do not need to prove it directly to be in the set. Instead, we may simply show how to reduce Q to SAT.

All of these statements can be formalized, but we omit the details here.⁸

⁴An excellent, if a little dated, reference for NP-complete problems is Garey and Johnson’s excellent book *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W. H. Freeman and Company, 1979). Examples of problems you may have heard of that are NP-complete, when they are appropriately generalized, include Tetris (reduction from 3Partition), Starcraft (and other RTS, reduction from Hamiltonian Path), Minesweeper (reduction from CircuitSat), and Pac-Man (reduction from Hamiltonian Cycle).

⁵There is, in fact, an entire hierarchy of these “computational complexity” classes, extending far above and far below P and NP. The complexity hierarchy includes classes of problems that assume access to a quantum computer, that provide no more than a logarithmic amount of space, etc. An excellent, and modern reference is the Complexity Zoo wiki: https://complexityzoo.uwaterloo.ca/Complexity_Zoo.

⁶Technically, these are called propositional logic formulas.

⁷Actually, we usually require a logarithmic-space reduction. These are always also polynomial-time reductions, but the reverse is, as far as we know, not true. Nearly all of the standard reductions that we say are polynomial in time are also logarithmic in space, so the difference is usually moot.

⁸Interested readers should refer to one many excellent texts. Gary and Johnson’s book—see footnote 4—is just

1.2 Reductions

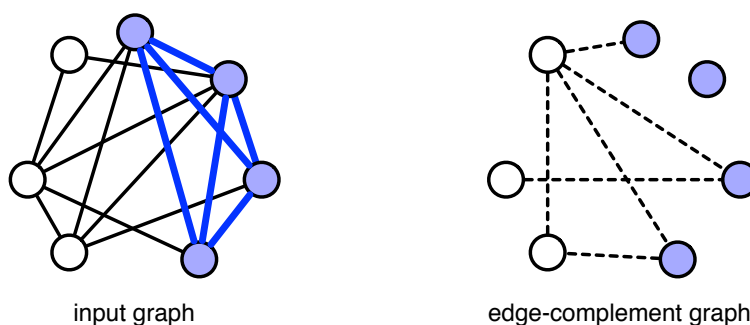
To prove that some problem Q is NP-hard, we use a reduction argument. That is, we demonstrate that another, known hard problem R can be solved *if* we have an algorithm that solves Q . Thus, since we know R is hard, Q must also be hard. Or rather, if Q were easy, then R would also be easy, which contradicts what we know about R .

To prove that a problem Q is NP-hard, show how to reduce a known NP-hard problem to Q .

Just as big- O notation is a statement about upper bounds, and just as we are mainly interested in worst-case behavior of our algorithms, the above rule is a statement about how hard some instances are for Q . In every problem, there will be instances that are easy, but these do not determine whether Q is hard *in general*. The hardness of Q is determined by the hardness of its hardest problems. This implies that if we have an algorithm that solves Q , and R is a special case of Q , then our algorithm can solve the hardest cases in R .

That is, a reduction simply shows that our hard problem R is a special case of our current problem Q , and this suffices to prove the hardness of Q .

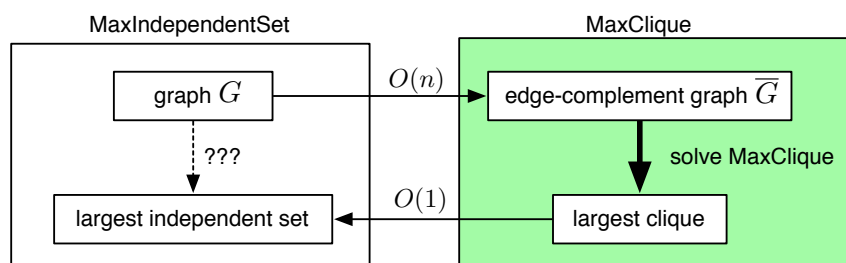
Example 1: Clique (from Independent Set). Suppose we are given an arbitrary graph $G = (V, E)$. The problem of MaxClique is to compute the number of nodes in the largest clique, which is a subgraph of size $k \leq |V|$ in which every pair of vertices is connected by an edge. That is, find the largest clique in G . The left-hand graph below is a small example G , with the largest clique, of size 4, highlighted; the right-hand figure shows the edge-complement graph \overline{G} , with the same vertices highlighted.



To prove that MaxClique is NP-hard, we will reduce it to MaxIndependentSet, which is known to be NP-hard (by reduction to 3SAT, which is NP-hard by reduction to SAT). Let G be an arbitrary graph. An *independent set* in G is a subset of vertices of G such that for every pair of vertices u, v

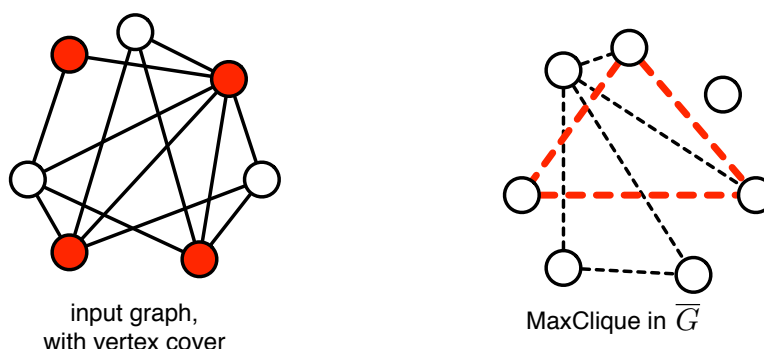
one. Another is Moore and Mertens, *The Nature of Computation* (Oxford University Press, 2011).

in the independent set, there is no edge (u, v) in G . The maximum independent set problem asks to find the largest independent set in G .



Here is the proof that MaxClique is NP-hard. Let the edge-complement graph $\overline{G} = (V, \overline{E})$ where $\overline{E} = V \times V - E$, i.e., it contains an edge if and only if that edge is not contained in E . Now, a set of vertices is independent in G if and only if the same vertices define a clique in \overline{G} , and we can compute the largest independent set in a graph by computing the largest clique in the edge-complement graph.

Example 2: Vertex Cover (from Independent Set). Suppose we are again given an undirected graph $G = (V, E)$. The problem of Vertex Cover is to find a set of vertices $C \subseteq V$ such that every edge $e \in E$ is “touched” by one of the vertices in the cover $v \in C$. The MinVertexCover problem is to find the smallest vertex cover of G . Here is an example of a vertex cover.



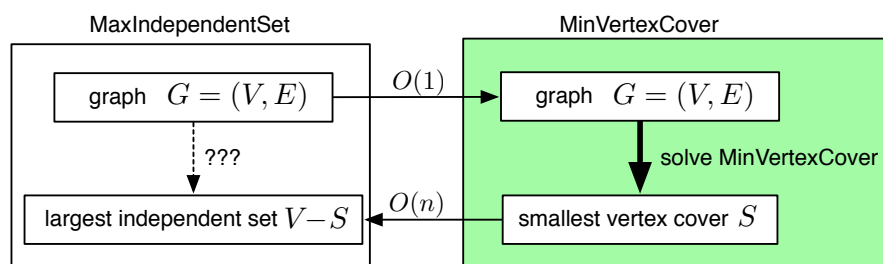
To prove that MinVertexCover is NP-hard, we make use of the following fact.

Lemma: If S is an independent set on G , then $V - S$ is a vertex cover.

Proof: Suppose $S \subseteq V$ is an independent set and $(u, v) \in E$. At most one of u, v can be in S , which implies that at least one or the other or both are in $V - S$. Thus, every edge is covered by

$V - S$. Similarly, suppose $V - S$ is a vertex cover and $u, v \in S$. Then $(u, v) \notin E$ because otherwise (u, v) would not be covered by $V - S$, which implies that S is an independent set. \square

Thus, to find the largest independent set, we simply need to find the vertices that are not in the smallest vertex cover on the same graph (and vice versa).



Example 3: Traveling Salesman. One of the most famous NP-hard problems is called the Traveling Salesman Problem, or TSP.⁹ In this problem, the input is a collection of n cities and the cost of travel d_{ij} between each pair of cities i, j , which we represent as an $n \times n$ matrix. The output is a *tour* that visits each city exactly once and then returns to the starting city, and we seek the tour with the smallest possible weight.

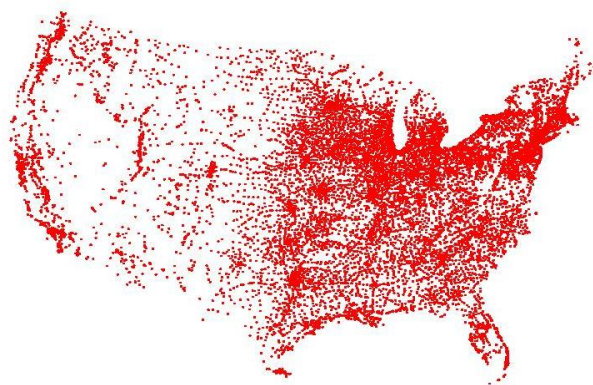
We can represent the solution to this problem as a permutation $\pi : \{1 \dots n\} \rightarrow \{1 \dots n\}$ where π_i denotes i successor in the sequence (much like the *pred* function in spanning-tree algorithms). Such a permutation implies a tour if we require that the n th element of π returns us to the 1st city, so that $1 \rightarrow \pi(1) \rightarrow \pi(\pi(1)) \rightarrow \dots \rightarrow 1$. With this formalization in hand, we can define the optimization and decision versions of TSP:

TSP (optimization)	TSP (decision)
Input: an $n \times n$ matrix $d_{ij} \geq 0$	Input: an $n \times n$ matrix $d_{ij} \geq 0$ and constant k
Output: a permutation π that minimizes $\text{cost}(\pi) = \sum_{i=1}^n d_{i, \pi_i}$	Question: is there a permutation π with $\text{cost}(\pi) \leq k$?

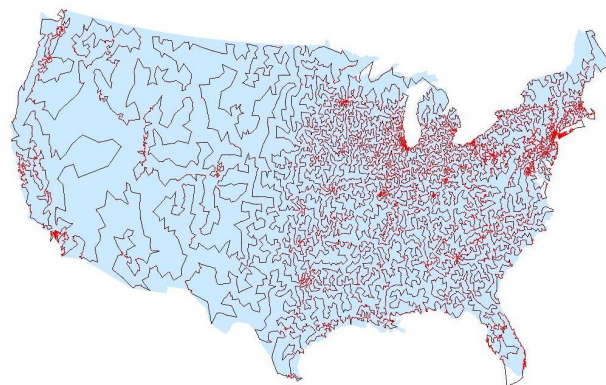
This formalization makes it immediately clear that there are $n!$ possible tours, and an exhaustive search of them to find the best tour would take exponential time.

⁹An excellent and comprehensive text on both historical and modern approaches for solving TSP is *The Traveling Salesman Problem* by Applegate, Bixby, Chvátal & Cook (Princeton University Press, 2007). Chapter 1 of this book is available online here: <http://press.princeton.edu/chapters/s8451.pdf>, and has a number of pretty pictures showing TSP solutions for both real and imaginary instances.

A pleasant property of Traveling Salesman problems, particularly those in Euclidean space, is that they make pretty pictures. Here is an example, using the 13509 cities in the contiguous United States that had a population of at least 500 in 1998, showing both the cities on the left and the optimal tour on the right.¹⁰



13509 cities, USA



13509 city tour

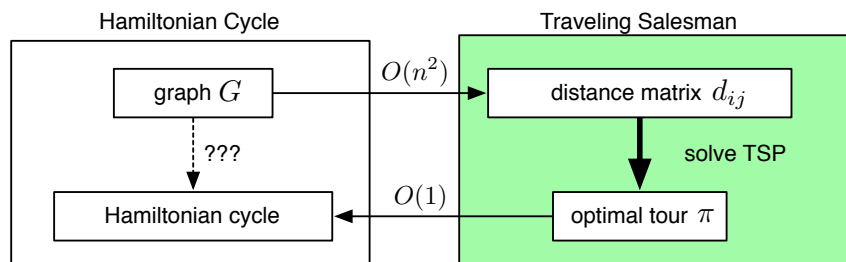
TSP is very similar to another NP-hard problem called Hamiltonian Cycle, in which we are given an unweighted, undirected graph $G = (V, E)$ and we seek a permutation π of the vertices such that each $(i, \pi_i) \in E$. We can prove that TSP is also NP-hard via a reduction from Hamiltonian Cycle by showing how to convert any instance of Hamiltonian Cycle into an instance of TSP.

Given G , we define a distance between each pair of vertices as

$$d_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ n & \text{if } (i, j) \notin E \end{cases}.$$

Thus, each pair of vertices that forms an edge in G is connected by a “cheap” edge in our TSP instance, while each pair that is not connected in G is connected by an edge whose cost is greater than the cost of the Hamiltonian cycle (if it exists) in our TSP instance. Under this reduction, which takes $O(n^2)$ time, G has a Hamiltonian cycle if and only if the corresponding TSP instance has a tour of cost n , while if G has no Hamiltonian cycle, $\text{cost}(\pi) \geq 2n - 1$.

¹⁰Images courtesy of William J. Cook at Georgia Tech, <http://www.isye.gatech.edu/~wcook/>, who is one of the authors of the text listed in footnote 9. The solution itself is by those authors. For much more on TSP, see <http://www.math.uwaterloo.ca/tsp/>.



2 Approximation algorithms

It may seem hopeless to solve NP-hard problems directly, since we have no polynomial-time algorithms for solving them and the alternative is either exhaustive search or heuristics (which do not provide guarantees, and thus may misbehave badly on some inputs). For optimization problems, there is a third path: *polynomial-time approximation schemes* (PTAS), which provide a guarantee on how far their result is from the optimal solution.

Let A be an algorithm, let x be an instance, let $A(x)$ be the quality of the solution produced by A on x , and let $\text{OPT}(x)$ be the quality of the optimal solution. We say that A is a ρ -approximation for a maximization problem if, for all instance x :

$$\frac{A(x)}{\text{OPT}(x)} \geq \rho, \quad (1)$$

where $\rho \leq 1$ and we would like ρ to be as large as possible. For a minimization problem, we say

$$\frac{A(x)}{\text{OPT}(x)} \leq \rho, \quad (2)$$

where $\rho \geq 1$ and we would like ρ to be as small as possible. It turns out that not all problems are equally hard to approximate, and sometimes we can get ρ to be arbitrarily close to 1, albeit often at the cost of increased running time.

Example 1: Vertex Cover. Recall that in the Vertex Cover (optimization) problem, we are given a graph $G = (V, E)$ and we see a cover set of vertices $C \subseteq V$ such that for every edge $(u, v) \in E$, at least one or both of $u, v \in C$. That is, every edge in G touches a vertex in the cover. Let C_{opt} denote the optimal vertex cover for G .

Here is a very simple 2-approximation algorithm for Vertex Cover. Let $C = \emptyset$, initially. We then choose an arbitrary edge (u, v) such that $u, v \notin C$, i.e., an edge such that neither of its end points are in the cover. Place both of u, v in C and repeat until there are no such edges. When the

algorithm halts, C is a vertex cover. (How long does this algorithm take to run when G is an adjacency list? What about an adjacency matrix?)

How large will C be? First, observe that $|C| = 2k$ if we selected k edges before halting. Since these vertices have no edges in common (do you see why?), and because each edge has at least one endpoint in C_{opt} , we have $|C_{\text{opt}}| \geq k$. Thus, C is at most twice the size of the optimal vertex cover, and

$$\rho = \frac{|C|}{|C_{\text{opt}}|} \leq \frac{2k}{k} = 2 . \quad (3)$$

This approximation ratio is tight when G is a complete bipartite graph on $2n$ vertices, where the algorithm selects all $2n$ vertices whereas a cover that consists of one side of the graph contains only n vertices.¹¹

Example 2: Metric Traveling Salesman. The general, we do not have algorithms for approximating TSP with any reasonable approximation ratio, and it has been proved that we have no hope of doing so unless $P=NP$. However, there are variations of TSP, which are still NP-hard, for which good approximation algorithms exist. In particular the Metric Traveling Salesman, or Δ TSP, in which we restrict distances obey the triangle inequality

$$d_{ij} \leq d_{ik} + d_{kj} \quad \text{for all } i, j, k , \quad (4)$$

yields constant-factor approximations.¹² Below, we will describe a 2-approximation algorithm for Δ TSP. A 3/2-approximation algorithm, due to Christofides, uses similar ideas.

Claim: the minimum spanning tree T provides a 2-approximation to the optimal Δ TSP tour.

Proof: Let OPT be the cost of the optimal tour on G . We can transform the MST T into a tour by converting each undirected edge in the tree $(u, v) \in T$ into a pair of reciprocated directed edges $\{(u, v), (v, u)\}$, and then skipping vertices we have already visited on the tour. There now exists an Eulerian tour of T because every vertex has even degree, and the cost of this tour is an upper bound on OPT :

$$\text{OPT} \leq 2\ell(T) , \quad (5)$$

where $\ell(T)$ is the length of the MST T .

¹¹A greedy heuristic, in which we repeatedly choose the $v \in V - C$ that would cover the most uncovered edges, actually performs much worse than this approximation algorithm, and has an approximation ratio of $\rho = \Theta(\log n)$. Can you identify a graph G produces this poor performance?

¹²Metric TSP is NP-hard by reduction from Hamiltonian Cycle.

Observe also that if we remove a single edge from the optimal tour, we have a spanning tree consisting of a path with $n - 1$ edges. This path has length at most OPT , and is at least as long as the MST T , implying the lower bound

$$\ell(T) \leq \text{OPT} , \tag{6}$$

where $\ell(T)$ is the length of the Eulerian tour on T . □

Christofides algorithm improves upon this algorithm by recognizing that the only vertices in T that prevent the existence of an Eulerian tour are the odd-degree vertices. Thus, instead of doubling the degree in T of every vertex, it only adds a single edge to each odd-degree vertex. There must be an even number of such odd-degree vertices in T (do you see why?), by creating a perfect matching on these vertices, we can “Eulerify” T .

The cost of our tour, when we skip vertices we have previously visited, is now $\ell \leq \ell(T) + \ell(M)$, where M is the length of the perfect matching on the odd-degree vertices of T . It can be shown that $\ell(M) \leq \text{OPT}/2$ (a fact that depends on the triangle inequality), which implies $\ell \leq \frac{3}{2}\text{OPT}$.

3 On your own

1. Read Chapter 34, NP-Completeness in CLRS
2. Read Chapter 35, Approximation Algorithms CLRS