Complexity Classes (Ch. 34)

The class P: class of problems that can be *solved* in time that is polynomial in the size of the input, n.

- if input size is n, then the worst-case running time is $O(n^c)$ for constant c.
- problems in P are considered "tractable" (if not in P, then not tractable)

Complexity Classes

The class NP: class of problems with solutions that can be *verified* in time that is polynomial in the size of the input.

- Imagine we are given a problem instance, along with a "certificate" of a solution (really a potential solution) of a problem. Then the problem is in NP if we can verify that the certificate is correct for the problem instance in time polynomial in the size of the input.
- Relies on the fact that checking a solution is easier than computing it (e.g., check that a list is sorted, rather than sorting it.)

NP-Completeness

The class NP-Complete (NPC): class of the "hardest" problems in NP.

- **this class has property that if any NPC problem can be solved in polynomial time, then all problems in NP can be solved in polynomial-time.**
- **actual status of NPC problems is unknown**
  - **No polynomial-time algorithms have been discovered for any NPC problem**
  - **No one has been able to prove that no polynomial-time algorithm can exist for any of them**
- **informally, a problem is NPC if it is in NP and is as "hard" as any problem in NP.**
- **we will use *reductions* to grow our set of NPC problems from one initial problem.**

P, NP, NPC...how are they related?

Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial-time without even being given a certificate.

So $\mathbf{P} \subseteq \mathbf{NP}$.

By definition, NPC $\subseteq$ NP

P, NP, NPC...how are they related?

Is NP $\subseteq$ P ???

- open problem, but intuition says no
- probably the most famous open problem in CS
- seems plausible that the ability to guess and verify a solution in polynomial-time is more powerful than computing a solution from scratch (in deterministic polynomial-time)
- so...we think P $\neq$ NP, but no one has proven it one way or the other (despite enormous effort).

P, NP, NPC...why do we care?

So...why do we care to know whether a problem is NP-Complete?

- **if it is, then finding a polynomial-time algorithm to solve it is unlikely.**
- **better to spend your time looking for:**
  - an efficient **approximation algorithm** to find solution close to optimal
  - **heuristics** that give correct answer with high probability

Decision Problems

Showing problems are either P or NP confines us to the realm of decision problems (problems with yes/no answers)

Example: Shortest paths

- **general (optimization) problem: What is the length of the shortest x to y path?**
- **decision problem: Is there an x to y path of length ≤ k?**

Decision Problems

Rationale for studying decision problems:

- **if the decision problem is hard (i.e., not solvable in polynomial time), the general problem is at least as hard**
- **for many problems, we only need polynomial extra time to solve the general problem after we solve the decision problem**
- **decision problems are easier to study and results are easier to prove**
- **all general problems can be rephrased as decision problems**

The general Traveling Salesman Problem:

- *instance*: a set of cities and the distance between each pair of cities (given as a graph).
- *goal*: Find a tour of minimum cost. (optimization problem)

Example: Traveling Salesman Problem (TSP) Decision Version

- *instance*: a set of cities and the distance between each pair of connected cities (given as a graph), and a bound B.
- *question*: is there a "tour" that visits every city exactly once, returns to the start, and has total distance ≤ B?

Example: Traveling Salesman Problem (TSP)

Is TSP ∈ NP?
To determine this, we need to show that we can verify a given solution (list of cities) in polynomial-time (i.e., time $O(n^k)$), where n is the number of cities and k is a constant).

Given an encoding of a TSP instance (a graph) and a certificate (a list of all cities in the order they are visited) and a bound B,

- **check that each city is in certificate exactly once**
- **check that the start city is also the end city**
- **check that total distance ≤ B**

All can be done in O(n) time, so TSP ∈ NP.

Reductions

Let L_1 and L_2 be two decision problems. Suppose we have a polynomial-time algorithm A_2 to decide L_2 but no algorithm to decide L_1. We can use A_2 to decide L_1 as well (still in polynomial-time).

All we need to do is find a polynomial-time reduction f from L_1 to L_2 ($L_1 \leq_p L_2$):

- f transforms an input for L_1 into an input for L_2 such that the transformed input is a yes-input for L_2 iff the original input is a yes-input for L_1
- f must be computable in polynomial-time (in the size of the input)
- if such an f exists, we say $L_1 \leq_p L_2$

Polynomial-time Reduction

We have a problem B that we know how to solve in polynomial-time and we would like to have a polynomial-time algorithm for problem A. We want to show that A $\leq_p$ B (B is known to be "easy" and A's running time is unknown)

Suppose we have a procedure that transforms any instance α of A into an instance β of B with the following characteristics:

1. The transformation is polynomial-time
2. The answers are the same. That is, the answer for α is "yes" iff the answer for β is also "yes"

Polynomial-time Reduction for "easiness"

We call this procedure a polynomial-time "reduction algorithm" because it gives us a way to show that A can be solved in polynomial-time (p-time).

1. **Given an instance α of A, use a p-time reduction algorithm to transform it to an instance β of B $\in$ P.**
2. **Run the p-time decision algorithm for B on the instance β**
3. **Use the answer for β as the answer to α**

In simple terms, we use the "easiness" of problem B to prove the "easiness" of problem A by showing **A $\leq_p$ B**.

Polynomial-time Reduction for "NPC-ness"

We can also use reduction to show that a problem is NPC. We can use a reduction to show that no p-time algorithm can exist for a particular problem B, assuming none exists for A.

Given an instance α of A for which we have no evidence of a p-time solution and a reduction algorithm to transform instance α of A to instance β of B,

1. **Convert the input α for A into input β for B**
2. **Run the decision algorithm for B on the instance β**
3. **If B has a p-time algorithm, then using the p-time transformation algorithm, we could convert an instance of A into an instance of B and solve A in p-time, a contradiction to the assumption that no p-time solution exists for A.** **A $\leq_p$ B.**

Reduction

To show that a problem Q is NPC, choose some known NPC problem P and reduce P to Q.

1. **Since P is NPC, all problems R in NP are reducible to P; that is, R $\leq_p$ P.**
2. **Show P $\leq_p$ Q.**
3. **Then all problems R in NP satisfy R $\leq_p$ Q, by transitivity of reductions.**
4. **Therefore, Q is NPC.**

Polynomial-time Reduction Ex: Hamiltonian Circuit Problem to TSP

The Hamiltonian Circuit Decision Problem (HC):
Instance: An undirected graph G = (V, E)
Question: Is there a simple cycle in G that includes every node?

The Traveling Salesman Decision Problem (TSP):
Instance: A set of cities, distances between each city-pair, and bound B
Question: Is there a "tour" that visits every city exactly once, returns to the start, and has total distance ≤ B?

Polynomial-time Reduction Ex: HC to TSP

Claim: HC $\leq_p$ TSP
Proof: To prove this, we need to do 2 things:

1. Define the transformation f mapping inputs for HC decision problem into inputs for TSP, and show this mapping can be computed in polynomial-time in size of HC input.
   - f must map the input G = (V, E) for HC into a list of cities, distances, and a bound B for input to TSP
2. Prove the transformation is correct.

Polynomial-time Reduction Ex: HC to TSP

1. Definition of transformation f for HC $\leq_p$ TSP:
Given the HC input graph G = (V, E) with n nodes:

- create a set of n cities labeled with names of nodes in V.
- add edges to make G completely connected
- set intercity distances $d(u,v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ 2 & \text{if } (u, v) \notin E \end{cases}$
- set bound B = n (since HC circuit must be of length n)

Note: Transformation of input can be computed in $O(n^2)$ time. Describe an algorithm to do so.

Polynomial-time Reduction Ex: HC to TSP

2. Prove the transformation f for HC $\leq_p$ TSP is correct

We will prove this by showing that $x \in$ HC iff $f(x) \in$ TSP

2(a) if $x \in$ HC, then $f(x) \in$ TSP
2(b) if $f(x) \in$ TSP, then $x \in$ HC

Proof of 2(a):

- o $x \in$ HC means HC input G = (V, E) has a Hamiltonian circuit. Wlog, suppose it is the ordering $(v_1, v_2, \ldots, v_n, v_1)$.
- o Then $(v_1, v_2, \ldots, v_n, v_1)$ is also a tour of the cities in f(x), the transformed TSP instance.
- o The distance of the tour $(v_1, v_2, \ldots, v_n, v_1)$ is n (= B), since each consecutive pair is connected by an edge and all edges in the input to HC have wt = 1.
- o Thus, $f(x) \in$ TSP, as required.

Polynomial-time Reduction Ex: HC to TSP

Proof of 2(b): if $f(x) \in$ TSP, then $x \in$ HC

- o $f(x) \in$ TSP means there exists a tour in TSP input of cities that has a total distance $\leq n = B$. Wlog, suppose the tour goes through cities $(v_1, v_2, \ldots, v_n, v_1)$.
- o Since all intercity distances are either of weight 1 or 2 in f(x), and there are n = B intercity "legs" in the tour, each "leg" in tour must have distance 1 and no "leg" with distance 2 is in the tour.
- o So G must have an edge of weight 1 between each consecutive pair of cities on the tour, and therefore $(v_1, v_2, \ldots, v_n, v_1)$ must be a Hamiltonian circuit in G
- o Thus, $x \in$ HC, as required. ■

Polynomial-time Reduction Ex: HC to TSP

Since **HC $\leq_p$ TSP, then**

- **o If there exists a polynomial-time algorithm for TSP, then there exists a polynomial-time algorithm for HC (HC is no harder than TSP)**
- **o If there does not exist a polynomial-time algorithm for HC, then there does not exist a polynomial-time algorithm for TSP (i.e., TSP is *at least* as hard as HC)**

NP-Completeness...and our use for polynomial-time reductions...

Definition:
A decision problem L is *NP-Complete (NPC)* if:
1. $L \in$ NP, and
2. for every $L' \in$ NP, $L' \leq_p L$ (i.e., every L' in NP can be transformed to L -- so L is at least as hard as every problem in NP).

Note: If L only satisfies condition 2, it is called NP-Hard. I.e., for NP-hard problems, no one has shown that a problem instance can even be *verified* in polynomial time.

An example of an NP-hard problem that is not NPC is the Halting Problem, for which a solution can't be verified in polynomial time.

Theorem 34.4: Suppose $L \in$ NPC:

- **o if there exists a polynomial-time algorithm for L, then there exists a polynomial-time algorithm for every $L' \in$ NP, i.e., P = NP**
- **o if there does not exist a polynomial-time algorithm for L, then there does not exist a polynomial-time algorithm for any $L' \in$ NPC, i.e., P $\neq$ NP**