Theory of NP-completeness

The theory of NP-completeness is based on **decision problems**. A decision problem is a problem that has only two possible outputs: yes or no (i.e., accept or reject).

Hamiltonian Cycle Decision Problem: Given a graph G, **<u>does</u>** G have a Hamiltonian cycle?

The **class P** is the set of decision problems that can be decided (solved) in polynomial time.
The **class NP** makes use of a "guess" function ("nondeterminism").

Informally, a decision problem L is in the **class NP** (nondeterministic polynomial) if you can write an algorithm that, for any instance I of the problem,

- constructs a guess in polynomial time, and
- checks the guess (answers yes/no for the instance I) in polynomial time.

This is often referred to as **polynomial verifiability**.

Example: The Hamiltonian Cycle Decision Problem is in the class NP
Determining if an arbitrary graph has a Hamiltonian cycle is a "hard" problem (think: backtracking). However, given a graph with *n* vertices, we can construct a guess (a random ordering of *n* vertices) and check that guess in polynomial time. Specifically, for a guess:

- check that all n vertices are visited and thus no vertex is repeated
- check if consecutive vertices in the sequence are adjacent

We can write an algorithm to do this in polynomial time (see algorithm below).

```
HCVerify(A) {                    // A  is an n x n adjacency matrix
     for i = 1 to n
          visited[i] = false
     for i = 1 to n {                 // create a random guess S
          S[i] = guess(V)         // pick a random vertex (1..n)
          visited[S[i]] = true    // mark that random vertex as visited
     }
     for i = 1 to n {
          if (!visited[i])        // not all vertices are visited
               return false
     }
     for i = 1 to n-1  {         // check consecutive vertices
adjacent
          if (A[S[i]][S[i+1]] != 1)
               return false
     }
     if A[S[n]][S[1]] != 1       // check last vertex adjacent to
first
          return false
     return true
}
```

The running time of HCVerify is $\theta(n)$, which is polynomial. So the Hamiltonian Cycle Problem is in the class NP.

Optimization problems
Many problems involve solving an "optimization" problem. For example:

- Traveling Salesperson Problem: find the tour of **minimum** total cost
- Shortest Path Problem: find a path of **minimum** total length
- Knapsack: find the set of items of **maximum** total size that fits in the knapsack
- Coin Change: make A cents using the **fewest** number of coins
- Maximum Matching: find a matching of **maximum** total weight

Question: If the theory of NP-completeness is based on decision problems (problems with only solutions yes or no), how do we apply this theory to optimization problems?

Answer: Introduce a bound

TSP Traveling Salesperson Problem (as an ***optimization*** problem**)**

Given a weighted graph G, determine a Hamiltonian cycle of minimum total weight.

Instance: a weighted graph G with weights w

Output: a sequence of vertices in G such that the sequence of vertices forms a Hamiltonian cycle and the total weight of the edges is a minimum possible

TSP Traveling Salesperson Problem (as a ***decision problem)***

Instance: A weighted graph G with weights w and an upper bound B

Output: Answer yes/no to question:
Does there exist a Hamiltonian cycle of G with total cost at most B?

NOTE:

(1) If we can solve an optimization problem efficiently, then we can solve the related decision problem efficiently.
(Use the minimum value found from the optimization problem and compare to the bound B. Then answer yes or no accordingly.)

(2) If the decision problem is "hard", then the related optimization problem must be at least as "hard" as well.
(If we cannot solve the decision problem efficiently, then there's little or no hope in solving the optimization problem efficiently.)

Example: Show that TSP is in the class NP.

```
TSPVerify(w, B)
{
//  w  is an  n x n matrix, w[i][j] is the weight of edge ij
//  Assumption: w[i][j] > 0 whenever i and j are adjacent
//  B  is a bound
//  Form a guess (a tour of n cities) and check to see if
//  (1) the guess is a tour and (2) total cost of tour <= B

     for i = 1 to n
          visited[i] = false

     for i = 1 to n
     {
          S[i] = guess(V)    // pick a vertex to visit
          visited[S[i]] = true
     }

     for i = 1 to n
     {
          if (!visited[i]) // not all vertices in the cycle
               return false
     }









}
```

Running time:

The **class P** is the set of decision problems that can be decided (solved) in polynomial time.
The **class NP** makes use of a "guess" function ("nondeterminism").

Important Result: $P \subseteq NP$ The class P is contained in the class NP
If you can solve the problem in polynomial time, you can verify a guess in polynomial time.

Unsolved Problem: Is $P = NP$?
Considered by many to be the most important open problem in the field of computer science.
It is one of the seven Millennium Prize Problems (selected by the Clay Mathematics Institute in the year 2000) to carry a $1,000,000 prize for the first correct solution.

Commonly held assumption: $P \neq NP$ (but no one has been able to prove it).

Question: What does it mean for a problem to be **NP-complete**?

Informally, a decision problem is **NP-complete** if

- it is in the class NP and
- it is "just as hard" as EVERY other problem in the class NP

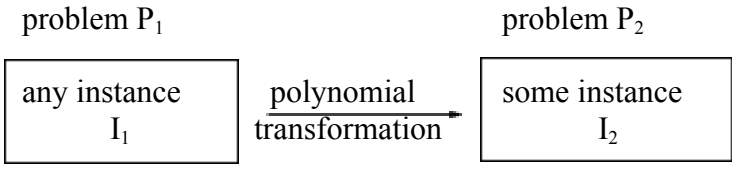
How do we show a problem is "just as hard" as another? "polynomial-time reduction"

The Class NP-complete

How do we show a decision problem is "just as hard" as another?

Reductions: A polynomial-time reduction is used to show that one decision problem is "just as hard" as another. The process involves taking an arbitrary, general instance of one problem and transforming it to some instance of the other problem in polynomial time.

Polynomial-time reduction from problem P_1 to problem P_2
(shows P_2 is "just as hard" as P_1)



Modify an arbitrary, general instance I_1 of problem P_1 by

- adding/deleting/transforming a polynomial amount of items for I_1,
- producing some instance I_2 of problem P_2 such that
- the answer for I_1 is "yes" for problem P_1 **if and only if** the answer for I_2 is "yes" for problem P_2

IMPORTANT CONCEPT: We are taking the set of all instances for one problem and transforming them to some (not necessarily all) of the instances for the other problem such that we get the same answer ("yes" for the instance I_1 $\Leftrightarrow$ "yes" for the transformed instance I_2).

Example: Find a polynomial-time reduction from Hamiltonian Cycle to Traveling Salesman Problem.

Take an arbitrary, general instance of Hamiltonian Cycle problem.

Instance I_1: A graph G

An instance of TSP is

Instance: A weighted graph G' and an upper bound B

Transform I_1 by defining $G' = G$ and defining the weights of all edges of G' to be 1 and defining $B = n$. Call this transformed instance I_2.

In particular, if A is the adjacency matrix of G, we define $w = A$ to be the weight matrix of the weighted graph G'. The transformation algorithm can be written:

```
n = A.length
B = n
for i = 1 to n
  for j = 1 to n
    w[i][j] = A[i][j]
```

This transformation algorithm takes time $O(n^2)$ and thus is polynomial.

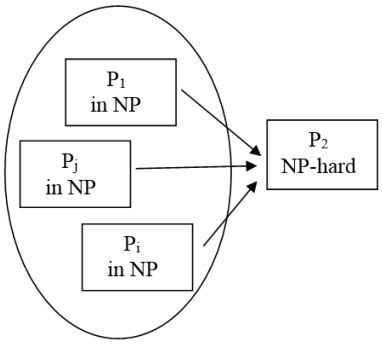
Claim: I_1 has a Hamiltonian cycle if and only if I_2 has a minimum tour of total weight ≤ B.

(=>) Assume I_1 has a Hamiltonian cycle.

(<=) Next assume that I_2 has a minimum tour of total weight at most B = n.

IMPORTANT CONCEPT: In this transformation, we are taking a set of instances (graphs) for the Hamiltonian cycle problem and transforming them to a SUBSET of the instances (weighted graphs) for the Traveling Salesman Problem such that we get the same answer for the instance and the transformed instance. We are NOT saying that every instance of TSP is a transformation FROM the Hamiltonian cycle problem. **This polynomial-time reduction shows that solving the Traveling Salesman Problem is AT LEAST as hard as solving the Hamiltonian cycle problem.**

Definition A problem P_2 is **NP-hard** if for **EVERY** problem P_i in the class NP, there is a polynomial-time reduction from P_i to P_2.



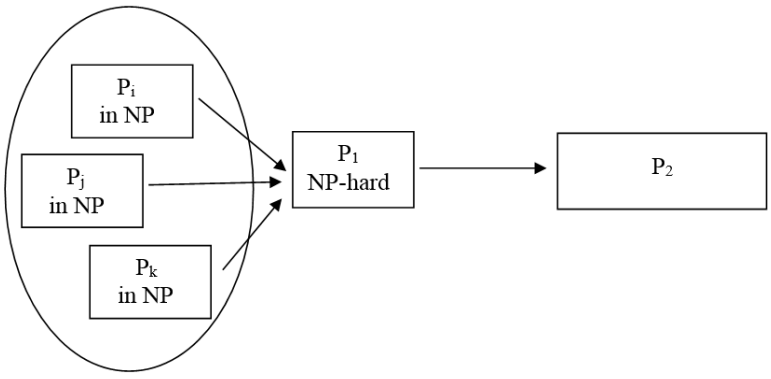
every problem in NP

Definition A problem P_2 is **NP-complete** if (1) P_2 is in the class NP and (2) P_2 is NP-hard.

There are some key NP-complete problems that have been proven: satisfiability (SAT), 3-satisfiability (3-SAT), vertex cover (VC), Hamiltonian cycle (HC).
These have been proven to be NP-complete (the most difficult proof of which is SAT; it involves creating a non-deterministic Turing machine that runs in polynomial time that transforms any problem in NP).

To prove a problem P_2 is NP-complete:

(1) prove that P_2 is in **NP**
(2) find a polynomial-time reduction **from** a known NP-hard (or NP-complete) problem P_1 to the problem P_2



NP-complete problems and polynomial-time reductions