A problem is said to be very hard, but its solution can be verified relatively easily, i.e., in polynomial time. Is this problem in NP?

Given two problems A and B, how do we check the claim that A is as hard as or even harder than B, in another word, A is “B-hard”?

What is NP-Complete?
What is Cook’s Theorem? Why is it important?
What are the three requirements for Polynomial Reductions?
Polynomial Reductions -- a formal way to say “as hard as”.

- Reduction is a transformation from one problem to another. Formally, let T be a function from input set for a decision problem U into the input set for a decision problem V, such that
  - For every string x, if x is a yes input for U, then T(x) is a yes input for V.
  - For every string x, if x is a no input for U, then T(x) is a no input for V.
    (Or equivalently, if T(x) is a yes input for V, then x is a yes input for U).

- T is a polynomial reduction when it can be computed in polynomially bounded time.
- Problem U is polynomially reducible to V, denoted as U ≤ₚ V, if there exists a polynomial reduction from U to V.

In practice, a polynomial reduction from the optimization problem U to the decision problem V can be constructed as follows: Suppose there is an algorithm S for the decision problem, then use S as a subroutine to construct an algorithm W that solves the optimization problem. Assume that S take constant time. If W runs in polynomial time, then the reduction is in polynomial time. Therefore, we found algorithm W that solves U in as much time as needed (module a polynomial) for solving V.
Example: A dominant set of a graph G is a subset of vertices W such that every vertex of G is either in W or is adjacent to a vertex in W.

Decision Problem
Input: A graph G and integer k
Question: Does G have a dominant set of size at most k?

Optimization Problem:
Input: A graph G
Output: A smallest dominant set of G

The Claim: the optimization problem for dominant set is no harder than* the decision problem.

Proof: Use Polynomial reduction.

1. For i from 1 to |G|, call the algorithm for the decision problem as a subroutine on input (G,i). This shall give us M, the size of the smallest dominant set of G.

2. Initialize all vertices in G as unmarked. Now choose an unmarked vertex v in G, add a new vertex X attached by an edge to v. Ask the decision algorithm if the modified graph has a dominant set of size M. If yes, then mark v as a member of the dominant set and leave X in the graph. If no, mark v as non-member and remove the vertex X. Repeat until M vertices are marked as members of the dominant set.

* Module a polynomial reduction.

- What to do in case of NP-Complete problems?
  - Use a heuristic
  - Find an approximate algorithm
  - Use exponential time algorithm anyway
Approximation algorithm: fast algorithms (i.e., polynomially bounded) that are not guaranteed to give the best solution but will give one that is close to the optimal.

Measurement of performance. For an approximate algorithm A and input I, the performance can be measured by the ratio

\[ r_A(I) = \frac{\text{value returned by A for } I}{\text{opt}(I)} \]

or \[ r_A(I) = \frac{\text{opt}(I)}{\text{value returned by A for } I} \]

Note that \( r_A(I) \geq 1 \).

Worst-case performance

For known optimal value

\[ R_A(m) = \max \{ r_A(I) \mid I \text{ such that } \text{opt}(I) = m \} \]

For any input

\[ S_A(n) = \max \{ r_A(I) \mid I \text{ of size } n \} \]

Bin Packing

For \( n \) objects with sizes \( s_1, \ldots, s_n \) where \( 0 < s_i \leq 1 \), find the smallest number of bins (each of capacity one) into which the objects can be packed (and find an optimal packing).

The First Fit Decreasing Strategy

Place an object in the first bin in which it fits.

binpackFFD(S, n, bin)
Initialize all used[] as 0.0 // space used up in bin j
1. for i ← 1 to n
2. for j ← 1 to n
3. if used[j] + s[i] < 1.0
4. bin[i] ← j // bin[i] is the index of the bin into which object li is placed
5. used[j] ← used[j] + s[i]
6. break // j loop
7. // continue i loop
Lemma 13.9 Let $S = (s_1, \ldots, s_n)$ be an input, in non-increasing order, for the bin packing problem and let $\text{opt}(S)$ be the optimal number of bins for $S$. All of the objects placed by FFD in extra bins (i.e., bins with index larger than $\text{opt}(S)$) have size at most $1/3$.

Proof

Let $i$ be the index of the first object placed by FFD in bin $\text{opt}(S)+1$. $S_i$ must be no larger than $1/3$.

Suppose $S_i > 1/3$. Then $S_1, \ldots, S_{i-1} > 1/3$, and are placed in bins $B_j$ for $1 \leq j \leq \text{opt}(S)$, at most two objects each bin. For some $k \geq 0$, the first $k$ bins hold one object each, and bins $B_{k+1}, \ldots, B_{\text{opt}}$ hold two objects each. $S_i > 1/3$ can not fit even by an optimal solution. But an optimal solution must fit object in in one of the first $\text{opt}(S)$ bins. Therefore, the assumption that $S_i > 1/3$ must be false.

Lemma 13.10 For an input $S = (s_1, \ldots, s_n)$ the number of objects placed by FFD in extra bins is at most $\text{opt}(S) - 1$.

Proof

Assume FFD puts $\text{opt}(S)$ objects with sizes $t_1, \ldots, t_{\text{opt}(S)}$ in extra bins. Let $b_i$ be the used space for each of the first $\text{opt}(S)$ bins. Then

$$\sum_{j=1}^n s_j \geq \sum_{i=1}^{\text{opt}(S)} b_i + \sum_{i=1}^{\text{opt}(S)} t_i = \sum_{i=1}^{\text{opt}(S)} (b_i + t_i) > \text{opt}(S)$$

However, since all the objects fit in $\text{opt}(S)$ bins, we must have

$$\sum_{j=1}^n s_j \leq \text{opt}(S).$$

Therefore, the assumption of $\text{opt}(S)$ objects being put into extra bins is not valid.
Lemma 13.10 \( R_{FFD}(m) \leq (4/3) + (1/3m) \). \( S_{FFD}(n) \leq 3/2 \), and for infinitely large \( n \), \( S_{FFD}(n) = 3/2 \).

Proof
Let \( S = (s_1, \ldots, s_n) \) be an input with \( \text{opt}(S) = m \). FFD puts at most \( m-1 \) objects, each of size at most 1/3, into extra bins, so FFD uses at most \( m + \lceil (m-1)/3 \rceil \) bins. Thus
\[
r_{FFD}(S) \leq (m + \lceil (m-1)/3 \rceil )/m \leq 1 + (m+1)/3m \leq 4/3 + (1/3m).
\]
Therefore, the worst-case ratio
\[
R_{FFD}(S) \leq 4/3 + (1/3m).
\]
Regardless of \( n \), the largest \( r_{FFD}(S) \) is achieved when \( m = 2 \). Therefore,
\[
S_{FFD}(n) \leq 4/3 + (1/6) = 3/2.
\]

Traveling Salesman Problem (TSP)
- Optimization problem: Given a complete, weighted graph, find a minimum-weight Hamiltonian cycle
- Decision Problem: Given a complete, weighted graph and an integer \( k \), is there a Hamiltonian cycle with total weight at most \( k \)?

Nearest-Neighbor Strategy
select an arbitrary vertex \( s \) to start the cycle \( C \)
\( v = s \)
while there are vertices not yet in \( C \)
select an edge \( vw \) of minimum weight, where \( w \) is not in \( C \).
add edge \( vw \) to \( C \)
\( v = w \)
Add the edge \( vs \) to \( C \)
return \( C \).
Shortest-Link Strategy

shortestlinkTSP(V, E, W)
R = E; // R is remaining edges
C = empty; // C is cycle edges
while R is not empty
  Remove the lightest edge, vw, from R
  If vw does not make a cycle with edges in C
    and vw would not be the third edge in C incident on v or w
    Add vw to C
  Add the edge connecting the endpoints of the path in C
return C.

Performance evaluation

Theorem 13.22 Let A be any approximation algorithm for the TSP. If there is any constant c such that r A(I) ≤ c for all instances I, then \( P = NP \).
Computing with DNA

Why DNA computing?
- The information density of DNA is greater than that of silicon: 1 bit can be stored in about one cubic nanometer.
- Operations on DNA are massive parallel

Adleman’s DNA algorithm for Hamiltonian path problem
1. Generate DNA strands to represent paths in G
2. Use biochemical processes to extract strands satisfying the following
3. A. Extract strands that start at v start and end at v end (discard the rest)
4. B. Extract strands that include n vertices (Discard the rest)
5. C. Extract strands that contain every vertex
6. Any strand that remains represents a Hamiltonian path from v start to v end. If no strand remains, G has no such path

Diagram of a graph: vertex 1 connected to vertices 2, 3, 4; vertex 2 connected to vertices 1, 3; vertex 3 connected to vertices 1, 2, 4; vertex 4 connected to vertices 1, 2, 3; vertex 5 connected to vertices 1, 2, 3; vertex 6 connected to vertices 1, 2, 3.