

1. Describe a proof that, for any three NP-problems, A, B, C , we have $A \leq_m B$ and $B \leq_m C$ implies $A \leq_m C$.

* There are two reductions

→ $A \leq_m B$, there exists a polynomial-time computable function f such that for all $x, x \in A$ iff $f(x) \in B$

→ $B \leq_m C$, there exists a polynomial-time computable function g such that for all $y, y \in B$ iff $g(y) \in C$

$$\& h(x) = g(f(x))$$

→ Since f and g are polynomial-time computable, the composition of two polynomial-time functions is also polynomial-time computable. This would mean h is polynomial time

* verifying the reduction made,

→ If I take an instance x of A

→ by definition of $A \leq_m B$ and $B \leq_m C$, $f(x) \in B$ iff $g(f(x)) \in C$

→ by construction, $g(f(x)) = h(x)$

→ This means, $x \in A$ iff $h(x) \in C$, which is the condition for $A \leq_m C$

* By constructing a polynomial time function h that reduces A to C , it is shown that $A \leq_m C$. moreover, the transitivity of many-to-one reductions among NP problems is solved.

2. Is there a path on G such that every node of G is covered exactly once?

→ Verifying the Hamiltonian path

input: A directed graph G and a sequence of nodes $p = (v_1, v_2, \dots, v_n)$

output: 'True' if p is a Hamiltonian path in G , 'False' otherwise.

1. ~~create~~

let visited be an array of size n with all entries initialized to 'False'

visited = ['False'] * n

2. For each node in sequence p :

→ For each node v_i in the sequence p ,

→ if visited[index of (v_i)] is 'False', v_i has not been visited yet

→ set visited[index of (v_i)] to 'True' to mark v_i as visited

→ Else, it means v_i has already been visited,

so return 'False' (the sequence is not a Hamiltonian path)

for i in range(len(p)):

if not visited[index of $(p[i])]$:

visited[index of $(p[i])]$ = True

else:

return False

3. Check if all nodes are visited:

→ after marking visited nodes, check if every node in G is visited

→ if any entry in visited is still False, return False

* for visited in visited

if not visited:

return False.

4. Check for edges between consecutive nodes

for i in range(len(p)-1): → For each consecutive pair of nodes (v_i, v_{i+1}) in p :

if not G .has_edge($p[i]$, $p[i+1]$): → Check if there is a directed edge from v_i to v_{i+1} in G

return False

→ if there is no edge return False

Conclusion

→ The introductio(n) is a hypothetical function that returns the index of the node in the graph's node list. The verification process is polynomial in the number of nodes n and, thus, confirms that the Hamiltonian path

5. If the sequence p passes is in NP

both the node visit test and the edge existence test, then p is a Hamiltonian path

return True

3. Is there a path on G such that every node of G is covered?

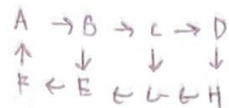
1. verification in polynomial time

→ verifying p is valid by checking two conditions

- p is a simple path of no repeated vertices
- p covers every node of G

2. Non-deterministic polynomial time guessing

→ Since verification process can be done in polynomial time, the guessing of the path can also be done in polynomial time



→ If both searches visit all nodes, then there exists a path that covers all nodes, if either search fails to visit all nodes, such a path doesn't exist.

4.

Show that we also have a deterministic polynomial time algorithm that decides whether C_1 and C_2 are equivalent.

The algorithm to decide equivalence of C_1 and C_2

1. Generate all inputs for 2^n where n is # of inputs for each

2. For each input, evaluate C_1 and C_2

3. Compare C_1 and C_2 outputs

4. If there exists any inputs for which the outputs of C_1 and C_2 are different then they are not equivalent

5. If outputs for C_1 and C_2 are the same then return equivalent

→ A deterministic polynomial-time algorithm to decide whether C_1 and C_2

are equivalent, but this is the boolean circuit is satisfiable.

→ 1. For each possible input (i_1, i_2, \dots, i_n) where i_j can be 0 or 1, generate the output y_1 produced by C_1 and the output y_2 produced by C_2

→ 2. If there exists any input for which y_1 is different from y_2 , then C_1 and C_2 are not equivalent, return equivalent

→ 3. If for all inputs y_1 is equal to y_2 then C_1 and C_2 are equivalent, return equivalent

→ A deterministic polynomial time algorithm to decide whether a circuit is satisfiable.