Need to return the actual minimum Hamiltonian Cycle!

Given only a single bit of information per call to the oracle, we already know how to get the weight $T^*$ of the minimum HC.

Idea: Use $T^*$ along with calls to the oracle to somehow figure out which edges are involved in the minimum HC.

Already know this call is poly-time reducible to TSP-Dec!

If removing edge $e$ removes every Hamiltonian cycle of minimum weight then $e$ is part of every minimum Hamiltonian cycle, and we add it to $H$ (and add it back into the graph).

To remove any dependence on this "other oracle," simply replace this call with the reduction code we showed.

At the end, the graph contains precisely the edges that are part of every minimum Hamiltonian cycle.

[Correctness] We prove: $H$ is exactly the minimum Hamiltonian Cycle.

We return here iff no Hamiltonian cycle exists...
Let's assume O(1/gap) time for reading/writing/arithmetic operations on each weight w (and O(gap) space).

Algorithm: TSP-Optimization-Solver(G = (V, E, w))
- external TSP-Optimal-Value-Solver
- ```
  if TSP-Optimal-Value-Solver(G, w) = null then return null
```
- generate adjacency matrix for G
- ```
  for all u in V \\
  for all v in V \\
  if u ≠ v then \\
  if w(u,v) > 0 then add to matrix
  else if w(u,v) < 0 then add -w(u,v) to matrix
  else add 0 to matrix
```
- ```
  for all u in V \\
  for all v in V \\
  if u ≠ v then \\
  if v(u,v) > 0 then add to matrix
  else if v(u,v) < 0 then add -v(u,v) to matrix
  else add 0 to matrix
```
- return matrix

This should not be surprising, since the same O(1/gap) time was introduced into both space and time complexities.

So this is a correct reduction, tell a polytime reduction?

What's Size(I)?

What is the runtime on such an input?

Clearly, O(\(\sigma_u, v \in V \log w(u,v)\)) to copy matrix

What's Size(\(I\))?

(What's a "reasonable" representation?)

Why?

Runtime = \(\sigma(\sigma_u, v \in V \log w(u,v)\) + O(\(n^2\) + \(\sigma_u, v \in V \log w(u,v)\))

What does this brute force solution with certificate verifying have to do with NP?
Given such an oracle, this algorithm would solve subset-sum in poly-time. If there exists a subset that sums to 0, then C is one such subset, and we return true. Otherwise, either C is not a subset of the input (return false), or C sums to a non-zero value (return false).

The "non-deterministic" part of the oracle is how it "magically returns" a yes-certificate if one exists. Non-deterministic is the N in NP: "Non-deterministic polynomial time."

Our definition of NP will not explicitly involve non-deterministic oracles, but it is based on certificate verification, which makes more sense if you think of such oracles. Could you "fool" the subset-sum verify function?

The problem of subset-sum as an example:

Let each certificate be an arbitrary set of integers. How to verify an arbitrary set of integers C is a subset of input I with sum zero?

1. Sum elements of C if nonzero, return false.
2. For each cert C (if C a subset of I with sum 0)
   - For y in C, if y not used [then used [y]]=false, used y]
   - For all x in I, if x=y, [then used [y]=true, used y]
   - if not found, return false; else return true.

So, subset-sum ∈ NP.
Let’s show that this problem is in NP!

Have to find a poly-time verify algorithm...

Type of certificate? Array of nodes (which may or may not represent a Hamiltonian cycle)

How to verify that a given array of nodes represents a cycle?

How about a Hamiltonian cycle?

This is a verify algorithm that we imagine being called on the certificate 𝑋 produced by oracle 𝐺

If 𝐺 is a no-instance of the problem, then "every certificate should cause verify to return false"

Easier to prove contrapositive: "If verify returns true, then 𝐺 is a yes-instance."

If 𝐺 is a yes-instance of the problem, then must show exists some certificate 𝑋 for which this procedure returns true.

If we return true, then the graph contains cycle with 𝑛 distinct nodes...

Million dollar question. We think not.

How about 𝑁𝑃 ⊆ 𝑃?

Million dollar question. We think not.

How are 𝑃 and 𝑁𝑃 related?

• 𝑃 ⊆ 𝑁𝑃
  • Consider a problem 𝐼 ∈ 𝑃
  • We show there exists a poly-time verify(𝐼, 𝑋) such that:
    • For every yes-instance 𝐼 ∈ 𝐼, verify(𝐼, 𝑋) = true for some 𝑋
    • For every no-instance 𝐼 ∈ 𝐼, verify(𝐼, 𝑋) = false for all 𝑋
  • By definition, there is a poly-time algorithm 𝐴 to solve 𝐼
    • Implement verify(𝐼, 𝑋) by simply running 𝐴(𝐼) (ignoring 𝑋)
    • Regardless of what 𝑋 is, verify(𝐼, 𝑋) satisfies the above
  • How about 𝑁𝑃 ⊆ 𝑃?