CS 341: ALGORITHMS

Lecture 20: final TSP reduction, and the complexity class \textbf{NP}

Slides by Trevor Brown (some material from Doug Stinson)

trevor.brown@uwaterloo.ca

(http://tbrown.pro)
THIS TIME

• Announcement: course evaluations
  • [evaluate.uwaterloo.ca](http://evaluate.uwaterloo.ca) until Apr 3
    • Please submit an evaluation!
    • Try to avoid evaluating effects of COVID19

• Finishing TSP reductions

• Complexity class NP
What about reducing TSP-Optimization to TSP-Dec?

**Problem 7.5**

TSP-Optimization

Instance: A graph $G$ and edge weights $w : E \rightarrow \mathbb{Z}^+$.  
Find: A hamiltonian cycle $H$ in $G$ such that $w(H) = \sum_{e \in H} w(e)$ is minimized.

Need to return the actual minimum Hamiltonian Cycle!

**Problem 7.7**

TSP-Decision

Instance: A graph $G$, edge weights $w : E \rightarrow \mathbb{Z}^+$, and a target $T$.  
Question: Does there exist a hamiltonian cycle $H$ in $G$ with $w(H) \leq T$?

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?

Given only a single bit of information per call to the oracle
TSP-Optimization $\leq^T_P$ TSP-Dec

Algorithm: TSP-Optimization-Solver($G = (V, E), w$)

external TSP-OptimalValue-Solver, TSP-Dec-Solver

$T^* \leftarrow$ TSP-OptimalValue-Solver($G, w$)

if $T^* = \infty$ then return ("no hamiltonian cycle exists")

$w_0 \leftarrow w$

$H \leftarrow \emptyset$

for all $e \in E$

\[
\begin{align*}
    & w_0[e] \leftarrow \infty \\
    & \text{if not TSP-Dec-Solver($G, w_0, T^*$)} \\
    & \quad \text{then } w_0[e] \leftarrow w[e] \\
    & \quad H \leftarrow H \cup \{e\}
\end{align*}
\]

return ($H$)

[Correctness] Loop invariant: there exists a HC of weight $T^*$ in $w_0$

By the end of the loop, $H$ contains all finite edges in $w_0$

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

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

We return here IFF no Hamiltonian cycle exists...

then $e$ is part of every minimum Hamiltonian cycle, and we add it to $H$
(and add it back into the graph)

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.

So some HC $C$ of weight $T^*$ is contained in $H$
At the end of the algorithm, there is a Hamiltonian Cycle $C$ of optimal weight $T^*$ **contained in** $H$.

If $H$ is precisely $C$, then we are done. **Suppose not** to obtain a contradiction.

In this case, there are some other edges in $H$ as well.

Let $e$ be one such edge.

Consider the iteration when $e$ was considered (and not removed). Let $G'$ denote the graph $G$ at this point in time.

Observe $H \subseteq G'$ (since we only remove edges over time). In particular, **this means** $C \subseteq G'$.

Since the algorithm **does not remove** $e$ in this iteration, removing $e$ would remove all Hamiltonian Cycles of weight $T^*$ in $G'$. In particular, **if we remove $e$ then $C \not\subseteq G'$**.

However, $e$ is not part of $C$.

So, since we start with $C \subseteq G'$, **if we remove $e$ from $G'$, then we will still have $C \subseteq G'$**, which is a contradiction!
Let's assume $O(1)$ for reading/writing/arithmetic operations on weights (and each weight takes $O(1)$ space).

### Algorithm: TSP-Optimization-Solver

```
Algorithm: TSP-Optimization-Solver(G = (V, E), w)

external TSP-OptimalValue-Solver, TSP-Dec-Solver
T* ← TSP-OptimalValue-Solver(G, w)
if T* = ∞ then return ("no hamiltonian cycle exists")

w₀ ← w
H ← ∅
for all e ∈ E
    do
        { w₀[e] ← ∞ if not TSP-Dec-Solver(G, w₀, T*)
        then
            { w₀[e] ← w[e]
            H ← H ∪ {e}
        }
return (H)
```

### Adjacency matrix representation for $V, E$

- $O(n^2)$ to copy matrix
- $O(n^2)$ to create empty array with one slot per $(u, v) \in V^2$
- $0(1)$ per iteration

### What's Size($I$)?

(What’s a “reasonable” representation?)

- $Size(I) = O(|V|^2 + \sum_{u,v \in V} 1)$
- $Size(I) = O(n^2)$

### What’s the runtime on such an input?

Runtime = $poly(Size(I)) + O(n^2)$

- How does $O(n^2)$ relate to $Size(I)$?
- Clearly $O(n^2) \in O(1 \cdot Size(I)^1)$
- So $O(n^2)$ is in $poly(Size(I))$

How would this change if we assumed each weight $w$ took $O(\log w)$ space, and operations on it took $O(\log w)$ time?
So this is a correct reduction. Is it a polytime reduction?

What's Size(I)?
(What's a “reasonable” representation?)

Adjacency matrix representation for $V, E$
and weight matrix representation for $w$

$$\text{Size}(I) = O(|V|^2 + \sum_{u,v \in V} \log w(u, v))$$

What's the runtime on such an input?

Runtime = \( \text{poly}(\text{Size}(I)) \) + \( O(n^2 + \sum_{u,v \in V} \log w(u, v)) \)

Clearly \( O(n^2 + \sum_{u,v \in V} \log w(u, v)) \in \text{poly}(\text{Size}(I)) \)

So, this is still a polytime reduction

Let's assume \( O(\log w) \) time for reading/writing/arithmetic operations on each weight $w$ (and \( O(\log w) \) space).

Suppose we show this is \( \text{poly}(\text{Size}(I)) \)

This should not be surprising, since the same \( O(\log w) \) terms are introduced into both space and time complexities...

Unit cost vs non-unit cost assumptions do not usually make a difference...
Show polytime equivalence by reducing problems to each other.

RECAP

• Showed three flavours of TSP are polytime-equivalent (i.e., if you can solve one flavour in polytime, you can solve all three flavours in polytime)
  • One of these was a decision problem (yes/no), and the other two were not (total weight, actual cycle)

• Decision and non-decision flavours of a problem are usually polytime-equivalent

• Proofs for a polytime Turing reduction
  • Correctness (return value is correct for every possible input)
  • Polytime (runtime is polynomial in the input size assuming a “reasonable” input representation”)

We will learn about a restricted class of reductions with clearer correctness guidelines soon...
COMPLEXITY CLASS $\text{NP}$

$\text{NP}$: Non-deterministic polynomial time
Suppose I give you a certificate consisting of an array of numbers, and claim it represents such a subset. Of course, I might lie and give you a subset that does not sum to zero… Finding such a subset can be extremely difficult.

If I’m telling the truth, then we call this a yes-certificate. It is is essentially a proof that “yes” is the correct output.

In this case, yes: (-3) + (-2) + 5 = 0

I could even give you numbers that are not in the input…

Can you use a yes-certificate to solve the problem efficiently?

Can you determine whether I am lying in polynomial time?
Certificates

Certificate: Informally, a certificate for a yes-instance $I$ is some “extra information” $C$ which makes it easy to verify that $I$ is a yes-instance.

Certificate Verification Algorithm: Suppose that $Ver$ is an algorithm that verifies certificates for yes-instances. Then $Ver(I, C)$ outputs “yes” if $I$ is a yes-instance and $C$ is a valid certificate for $I$. If $Ver(I, C)$ outputs “no”, then either $I$ is a no-instance, or $I$ is a yes-instance and $C$ is an invalid certificate.

Polynomial-time Certificate Verification Algorithm: A certificate verification algorithm $Ver$ is a polynomial-time certificate verification algorithm if the complexity of $Ver$ is $O(n^k)$, where $k$ is a positive integer and $n = \text{Size}(I)$.

For example, for subset-sum, a correct $Ver(I, C)$ should return “yes” only if $C \subseteq I$ and $\text{sum}(C) = 0$.

It can be hard to define a certificate for a no-instance... E.g., how to create a certificate that proves no subset sums to 0?
**SUBSET-SUM: ALGORITHM VIA VERIFYING CERTIFICATES**

**Type of a certificate:**
set of integers

1. `SubSetSum(X[1..n])`
2. `for` every possible subset `S` of `X`
3. `if` `sumsToZero(S)` `then` return `true`
4. `return` `false`

- **Generating all certificates** is expensive; exponential time!
- **But verifying one** certificate is fast; runtime is $\text{poly}(|S|)$
- **Input** to `verify` is $I = (X, S)$. Runtime is $O(|S|)$, which is $O(\text{Size}(I)) = O(|X| + |S|)$ (and, hence, poly-time)
- **Verify certificate** $S$ (valid + sums to zero)
- **Generate every subset certificate** $S$
- **A certificates that does not** sum to zero doesn’t really prove anything (would need to know that all certificates sum to non-zero)
- **What does this brute force solution with certificate verifying** have to do with NP?
Suppose instead of generating every possible subset, there exists a poly-time non-deterministic oracle which magically returns a subset that sums to 0 if one exists and otherwise returns any set of integers.

Given such an oracle, this algorithm would solve subset-sum in poly-time.

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”

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).
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...

**GENERALIZING BEYOND SUBSET-SUM**

- You can solve any decision problem in non-deterministic poly-time if you have:
  1. a poly-time non-deterministic oracle, and
  2. a poly-time *verify* algorithm
- Such that:
  - If $I$ is a **yes**-instance, then the oracle returns a **yes**-certificate $C$ (i.e., a “proof” the answer is “yes”) and $\text{verify}(I, C)$ returns *true*
  - If $I$ is a **no**-instance, then $\text{verify}(I, C)$ returns *false* for all $C$ (i.e., it must be impossible to fool $\text{verify}$ into returning true)
- The algorithm:

```plaintext
1. SolveAnyProblemWithOracle(I)
2. C = Oracle(I)
3. return verify(I, C)
```

Could you “fool” the subset-sum verify function?
Oracle

Guesses solution in \( O(1) \) time

Verifies solution in poly-time

Problem: plug in USB drive

DID YOU JUST... PLUG THAT IN CORRECTLY ON YOUR FIRST TRY?

Oracle

As we are about to see: existence of an oracle + poly-time verifier for this problem means problem is in NP

Oracle

WITCH

verify
DEFINING NP

Intuition: For a yes-instance, there must exist some certificate that verify would accept (and, if one exists, the oracle would find it, solving the problem). For a no-instance, verify must always reject.

- A decision problem \( \Pi \) is solved by a poly-time verify alg. iff:
  - for every yes-instance \( I \), there exists a certificate \( C \) such that \( \text{verify}(I, C) \) returns true, and
  - for every no-instance \( I \), \( \text{verify}(I, C) \) returns false for every \( C \)

- The complexity class \( \text{NP} \) denotes the set of all decision problems that can be solved by poly-time verify algorithms

- Note: it is not necessary to be able to implement an oracle for a problem to be in \( \text{NP} \). We can simply assume an oracle exists, and show a poly-time verify algorithm exists.
Always keep the following in mind: finding a certificate can be much more difficult than verifying a given certificate.

As a rough analogy, finding a proof for a theorem can be much harder than verifying the correctness of someone else’s proof.
MECHANICS OF SHOWING A PROBLEM IS IN NP

• How to show $\Pi \in NP$
  1. Define a type of certificates
  2. Design a poly-time $verify(I, C)$ algorithm
  3. Correctness proof
     • Case 1: Let $I$ be any yes-instance; Find $C$ such that $verify(I, C) = true$
     • Case 2: Let $I$ be any no-instance, and $C$ be any certificate; Prove $verify(I, C) = false$

Example Case 1: Let $I$ be a yes-instance. There is a subset in $I$ that sums to 0. Let $C$ be any such subset. Clearly verify($I, C$) will return true.

Example Case 2: Let $I$ be a no-instance, $C$ be any certificate. No subset of $I$ sums to 0. So, $\text{sum}(C) \neq 0$ and verify returns false, or $C \not\subseteq I$ and verify returns false.

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?

Sum elements of $C$
if nonzero return false.
Else test $C \subseteq I$ with two loops.

for $x$ in $C$
    $\text{found}_x = false$
for $y$ in $I$
    if $x = y$ and not $\text{used}[y]$ then
        $\text{used}[y] = true$
        $\text{found}_x = true$
    if not $\text{found}_x$ return false
return true

$\mathcal{O}(|C||I|)$ time

This is certainly polytime...

So, subset-sum $\in NP$
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?

**Problem 7.2**

**Hamiltonian Cycle**

**Instance:** An undirected graph $G = (V, E)$.  
**Question:** Does $G$ contain a hamiltonian cycle?

A **hamiltonian cycle** is a cycle that passes through every vertex in $V$ exactly once.
A certificate consists of an array of node names (in 1...n), which might represent a Hamiltonian cycle.

1. `HamiltonianCycleVerify(G=(V,n,E,m), X)`
2. if size(X) is not n then return false
3. used[1..n] = array containing all false
4. for i = 1..n
5.     if used[X[i]] then return false
6.     used[X[i]] = true
7. for i = 1..(n-1)
8.     if no edge X[i] to X[i+1] then return false
9. if no edge X[n] to X[1] then return false
10. return true

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

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

Yes-instance implies there is a Hamiltonian cycle. Suppose $X$ is a sequence of $n$ consecutive nodes on that cycle. Then we return true!

So, Hamiltonian Cycle is in NP.
HOW ARE P AND NP RELATED?

- $P \subseteq NP$
  - Consider a problem $\Pi \in P$
  - We show there exists a poly-time $verify(I, C)$ such that:
    - For every yes-instance $I$ of $\Pi$, $verify(I, C) = true$ for some $C$
    - For every no-instance $I$ of $\Pi$, $verify(I, C) = false$ for all $C$
  - By definition, there is a poly-time algorithm $A$ to solve $\Pi$
    - Implement $verify(I, C)$ by simply running $A(I)$ [ignoring $C$]
    - Regardless of what $C$ is, $verify(I, C)$ satisfies the above
  - How about $NP \subseteq P$? Million dollar question. We think not.