Computational Complexity I
COMP4500/7500
Advanced Algorithms & Data Structures

October 12, 2020

Overview

- The complexity class P.
- (And then we will look at and compare NP, NP-hard and NP-complete problems.)
- Definitions of key concepts: *P* (polynomial time), *NP* (nondeterministic polynomial time), *NP Hard*, *NPC* (NP-complete)
- Problem reduction: showing a problem is *NPC*
- Travelling Salesman Problem (TSP)
  - Relationship to other “hard” problems
  - Approximation

Polynomial time: tractable

- Polynomial-time algorithms are considered tractable.
- In practice, an algorithm of complexity $\Theta(n^{100})$ is not tractable.
- However, if a problem has a polynomial-time algorithm, there is usually an efficient polynomial time algorithm.

Polynomial time: computational models

The class of problems solvable in polynomial time on a serial random-access machine (as we have been using) is the same as:

- the class of problems that can be solved in polynomial-time on an abstract Turing machine;
- the class of problems that can be solved in polynomial-time on a parallel computer when the number of processors grows polynomially with the input size.

For many reasonable models the class of problems that can be solved in polynomial time is the same.

Polynomial time: closure properties

- Polynomials are closed under addition, multiplication and composition.
- For example, if the output of a polynomial-time algorithm F is fed as input to another polynomial-time algorithm B, the composite algorithm is polynomial time.

Polynomial time

Which **problems** have polynomial-time algorithms?

Problems that *seem* similar may not be...

Shortest vs longest simple paths

- Can find the **shortest paths** from a single source vertex in a directed graph in $O(|V|.|E|)$ time, i.e. polynomial time.
- Finding a **longest simple path** between two vertices is difficult.
- Even determining whether a graph contains a **simple path with at least a given number of vertices** is difficult (it is *NP-complete*).

Euler tour vs hamiltonian cycle

- An **Euler tour** in a connected directed graph is a cycle in which each edge of the graph is traversed exactly once.
- Finding an Euler tour can be solved in $O(|E|)$ time.
- A **hamiltonian cycle** in a graph G is a simple cycle that contains all the verticies ($G.V$) in the graph.
- Determining **whether a graph has a hamiltonian cycle** is difficult (it is *NP-complete*).

2-CNF vs 3-CNF satisfiability

- A boolean formula like

  $$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$$

  has a satisfying assignment $x_1 = 1, x_2 = 0, x_3 = 1$, i.e., an assignment such that it evaluates to 1.
- A boolean formula is in **conjunctive normal form (CNF)** if it is a conjunction of clauses, each of which is a disjunction of literals, each of which is either a variable or its negation.
- It is in **k-CNF form** if it is in CNF form in which each disjunction has exactly k literals – the example above is in 2-CNF form.
- Determining whether a **2-CNF** formula is satisfiable can be done in polynomial time.
- Determining whether a **3-CNF** formula is satisfiable is difficult (it is *NP-complete*).

Defining classes of problems

Which problems have polynomial-time algorithms?

To study this, in complexity theory we study **classes of problems** (not algorithms):

- We focus on **decision problems** rather than optimisation problems;
- We focus on **concrete problems** rather than abstract problems.

Decision problems

- **Decision problems** have a simple 1 or 0 answer as their solution.
- **Optimisation problems** usually have closely related decision problems.
- If the optimisation problem is “easy”, the related decision problem is also “easy”.
- If the decision problem is “hard”, the related optimisation problem is also “hard”.

Decision problem: example

- Finding a SHORTEST PATH in a graph is an optimisation problem.
- Deciding if a graph G has a PATH from u to v with at most k edges is a related decision problem.

```
SOLVE-PATH($G, u, v, k$)
1  $p =$ SOLVE-SHORTEST-PATH($G, u, v$)
2  **if** $length(p) \leq k$
3      **return** 1
4  **else**
5      **return** 0
```

- SHORTEST-PATH can be solved in polynomial time and hence PATH can be solved in polynomial time.

Abstract problems

Definition (Abstract problem)

An *abstract problem* is a binary relation on a set I of problem instances and a set S of problem solutions

May be multiple solutions for a given problem instance.

Definition (abstract decision problem)

An *abstract decision problem* is a function from an instance set I to the solution set $\{0, 1\}$

Unique 0 or 1 solution for any given instance.

Encodings

- Need to represent problem instances in a way a program can understand.
- An **encoding** of a set S of abstract objects is a mapping from S to the set of binary strings $\{0, 1\}^*$.
- Encode a natural number in the usual way $0, 1, 10, 11, 100, 101, 110, 111$
- Encode a compound object as a binary string combining the representations of its constituent parts.

Concrete problems

Definition (concrete problem)

A *concrete problem* is a problem whose instance set is the set of binary strings $\{0,1\}^*$.

Definition (solves)

An algorithm *solves* a concrete problem in time $O(T(n))$ if, when it is provided a problem instance i of length $n = |i|$, the algorithm produces a solution in $O(T(n))$ time

The complexity class P

Definition (polynomial-time solvable)

A concrete problem is *polynomial-time solvable* if there exists an algorithm to solve it in $O(n^k)$ time for some constant k.

Definition (complexity class P)

The *complexity class P* is the set of concrete decision problems that are polynomial-time solvable.

Definition: The complexity class P

The *complexity class P* is the set of concrete decision problems that are polynomial-time solvable.

- *Decision problems* have a simple 1 or 0 (yes/no) answer
- A *concrete problem* is a problem whose instance set is the set of binary strings $\{0,1\}^*$
- A concrete problem is *polynomial-time solvable* if there exists an algorithm to solve it in $O(n^k)$ time for some constant k, where n is the length of the binary input string.

Definition: The complexity class *P*

How do we know if a concrete decision problem is in *P*?

To show a problem *A* is in *P* we could:

- Come up with a polynomial-time algorithm to solve *A*
- If we already know that another problem *B* is in *P*:
    - find a *polynomial-time reduction algorithm* to transform an instance α of a decision problem *A* into an instance β of a decision problem *B*.

Reductions of decision problems

Definition (polynomial-time reduction algorithm)

A *polynomial-time reduction algorithm F*

- Transforms an instance α of a decision problem *A* into an instance β of a decision problem *B*
- The transformation algorithm is polynomial time
- The answers are the same: if $\beta = F(\alpha)$ then $A(\alpha) = B(\beta)$

Reducing decision problem *A* to decision problem *B*

- Assume REDUCE-A-TO-B is a polynomial time algorithm to reduce an instance α of *A* to an instance β of *B*
- Can solve problem *A* in terms of *B*

```
SOLVE-A($\alpha$)
1   $\beta$ = REDUCE-A-TO-B($\alpha$)
2   return SOLVE-B($\beta$)
```

- If SOLVE-B is polynomial time, then SOLVE-A is also
- If *B* is “easy”, then so is *A*

Definitions: The complexity class *NP*

The *complexity class NP* is the set of concrete decision problems for which a solution (a *certificate*) can be checked (*verified*) in polynomial time.

- **N**ondeterministic **P**olynomial: if you could simultaneously check all solutions to a given input, then the problem would be polynomial-time solvable.
  i.e. you could *nondeterministically* pick the right solution and check it in *polynomial* time
- Problems for which we can't even verify a solution in polynomial time are unlikely to have a polynomial-time (efficient) algorithm to generate solutions.

$P = NP$???

- P is a subset of NP trivially
- Are there some problems where a solution is quick to verify, but which can't be implemented efficiently?

  Unknown.

  But most people (everyone?) thinks there are, we just haven't worked out how to prove it yet.
- This is the famous "$P = NP$" problem.

You can pick up $1million if you disprove (or prove) it.

Example of an *NP* problem: TSP

The Travelling Salesman Problem (TSP) is to find the shortest path that visits every vertex exactly once in a graph and returns to where it starts. (Assume positive weights)

- For the sake of reasoning, we deal with a *decision problem*: is there a tour of length at most k.
- A solution S is a list of vertices in the order visited.

```
TSP-CHECK($G, S, k$)
1  Check that every vertex is in $S$ exactly once
2  Check that every pair of adjacent vertices in $S$
       is connected by an edge in $G$
3  Check there is an edge from the last vertex to the first
4  Sum the weights of all such edges and call the total $t$
5  return $t \leq k$
```

Example of an *NP* problem: TSP

- TSP $\in$ *NP* is trivial to see: the above checking algorithm for TSP is polynomial time.
- We would like it to be in *P*, i.e., for there to be a polynomial-time algorithm that generates solutions to TSP.
- Frustratingly, we just don't know (yet) whether it is in *P* or not.
  This is despite
  - TSP being of practical (commercial) importance, and
  - decades of research into efficient solutions to TSP specifically and computational complexity generally.

$P = NP$??

Some progress has been made...

NP Hard

A concrete decision problem *B* is *NP*-hard when every problem $A \in NP$ is *polynomial-time reducible* to *B*.

- Problem *A* is *polynomial-time reducible* to *B* if there exists a polynomial-time reduction algorithm to transform every instance α of *A* into an instance β of *B*.

And so ...

If any one of these problems can be solved by a polynomial-time algorithm, then *all* problems in *NP* can be solved by a polynomial time algorithm.

NP-complete

A concrete decision problem *B* is *NP*-complete if and only if :

- it is *NP*-hard, and
- it is in *NP*

And so ... these are the hardest problems in *NP*.

NP-complete

Are there any *NP*-complete problems? Yes!

- It turns out TSP is in NPC.
- Thus, if you find a way to solve TSP efficiently, you can solve a large class of important problems efficiently, and at the same time prove $P = NP$.
- In other words, you are unlikely to find an efficient solution to TSP.

That the set NPC is non-empty is a helpful clue to $P = NP$.

It seems unlikely that *all* problems in *NP* are polynomial-time solvable, so "probably" the NPC problems are *not* polynomial-time solvable.

NP-complete

If you can prove that the problem you are trying to solve is *NP*-complete, you know that it is unlikely that it is polynomial-time solvable.

NP recap

- Recall: *NP* is the set of problems for which it is not unreasonable to think there may be there may be efficient solving algorithms
- We have identified a useful subset, *P*, for which efficient algorithms exist.
- Many (theoretically and practically) important algorithms are in NP, but we *don't know* whether or not they are in *P*.
- Interestingly, we have found the subset NP-complete, which are the "hardest" of any in NP.
- We are working towards learning a technique for showing that a problem is NP complete, then it is reasonable to stop trying to find an efficient algorithm

NPC and Cook's theorem

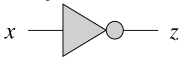
Cook's theorem:

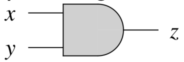
The CIRCUIT-SAT problem is NP complete.

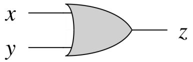
- This is the "first" NP complete problem.
- A *circuit* refers to a type of logical circuit, AND and OR gates, etc.
- Cook showed that any problem in NP can be **reduced** *in polynomial time* to a circuit, that is, restated as a problem using a circuit.
- Thus, if one can solve CIRCUIT-SAT in polynomial time, and your problem can be reduced to a circuit in polynomial time (any problem in *NP* can), then you can solve your problem in polynomial time.

CIRCUIT-SAT

A *circuit* is a basic concept from computer science; perform logical operations on binary inputs.
Usually consider three types of gates:







x	$\neg x$
0	1
1	0

(a)

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

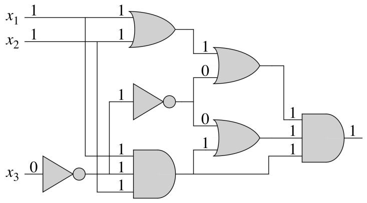
(b)

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

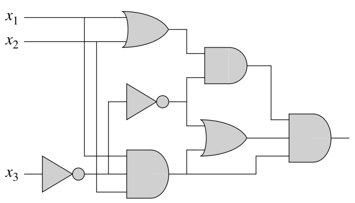
(c)

CIRCUIT-SAT

Compose these elements into a full circuit with no cycles and one output:



(a)



(b)

SAT

- CIRCUIT-SAT asks for an assignment of 1 or 0 to each of the variables $(x_1, x_2, \ldots)$ that produces '1' as the output.
- Figure (b) has no such assignment.

CIRCUIT-SAT is NP Hard

Theorem

CIRCUIT-SAT is NP Hard

Proof outline:

- all inputs may be encoded in binary format *in polynomial time*, and all problems stated as circuits.
- Thus, if CIRCUIT-SAT can be solved in polynomial time, all other *NP* problems can be as well

CIRCUIT-SAT is NP Hard

How does Cook's theorem help? Run this algorithm for solving problem *L* with input *i*:

```
SOLVE-L($i$)
1   Transform (reduce) the input $i$ to $i'$ for CIRCUIT-SAT
2   return CIRCUIT-SAT($i'$)
```

- Step 1 is polynomial time and applies to all problems in *NP* (from Cook's theorem)
- hence if Step 2 is also polynomial time, all problems in *NP* can be solved in polynomial time
- (including those in NPC)

NPC and Cook’s theorem

Proving that there was at least one problem in NPC was difficult to do.

We had to show that *all* problems in *NP* were polynomial-time reducible to CIRCUIT-SAT.

Now that someone else has done the difficult part, showing that other problems are NP-hard is easy!

Reductions

Let X be your own problem.

If you can show that CIRCUIT-SAT is polynomial-time reducible X, then X is NP-hard.

Why?

If CIRCUIT-SAT reduces to X, and if you have a polynomial-time solution to X, then you can solve CIRCUIT-SAT in polynomial time, and hence can then solve all NP problems in polynomial time.

That is, implement CIRCUIT-SAT thus:

```
CIRCUIT-SAT($i$)
1  Transform (reduce) the input $i$ to $i'$ for $X$
2  **return** X($i'$)
```

Reductions

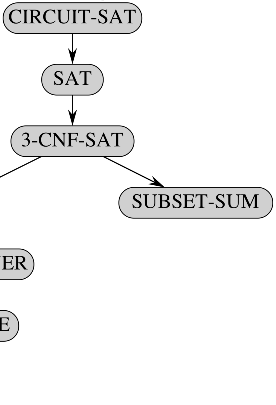
Let X be your own problem.

If you can show that CIRCUIT-SAT is polynomial-time reducible X, then X is *NP*-hard.

To show that X is *NP*-complete you *also* need to show it is in *NP*!

Reductions

In practice, reducing CIRCUIT-SAT directly to X may be tedious; instead choose a known NP Hard problem that is closer to X.



Read "$\rightarrow$" as "reduces to".

Note that polynomial-time reductions are transitive.

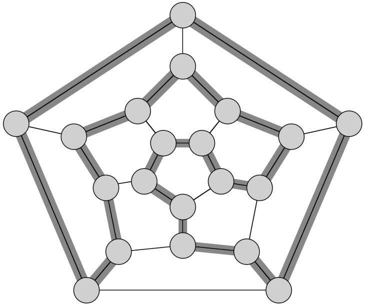
Travelling Salesman problem (TSP)

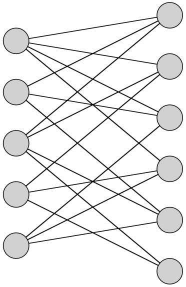
In a *weighted* graph, find the least-cost path that visits every vertex exactly once, returning to the start.
Recall the verification algorithm TSP-CHECK(G, S, k).
To show TSP is in NPC we need

1. $TSP \in NP$
   The verification problem is polynomial-time (from earlier).
2. TSP is NP-Hard.
   Reduce HAM-CYCLE to TSP.

Hamiltonian Cycle

A Hamiltonian Cycle is a simple path through an *unweighted* graph that contains every vertex, starting and ending at the same vertex.


(a)


(b)

A graph is *Hamiltonian* if it contains a *Hamiltonian Cycle*.
(a) is Hamiltonian; (b) is not
The HAM-CYCLE problem is to find a(ny) Hamiltonian cycle.

Reduce HAM-CYCLE to TSP

Recall:

- Assume we can solve TSP in polynomial time
- Then how can we use that (non-existent?) algorithm to solve HAM-CYCLE?

The trick is in setting the weights in the TSP version:

- If an edge exists, give it weight 0
- If not, give it weight 1.

Use the TSP algorithm to find a tour with weight 0.

Recap

- Important classes of problems:
  - P: Polynomial-time to solve
  - NPC (NP-complete): “Probably” worse than polynomial-time to solve
  - NP: Problems that may feasibly be polynomial-time (we can check a solution in polynomial-time)

$P \subseteq NP$ ✓ $\quad NPC \subseteq NP$ ✓ $\quad P = NP$??? $\quad NPC \neq P$???

- Important first problem: CIRCUIT-SAT(recall Cook’s Theorem)
- This forms the basis of a chain of reductions, hence the set NPC is non-empty, non-singleton.
- Using *reduction* we can show a new problem is also NPC.
- Hence that problem is unlikely to have an efficient algorithm for solving it.

End of the story?

- In practice you won’t keep your job long if you give up.
- Many real problems have efficient algorithms that give a good approximation;
- For TSP, this is enough in practice (*reduce* cost, not necessarily *minimise* cost).

Approximate TSP

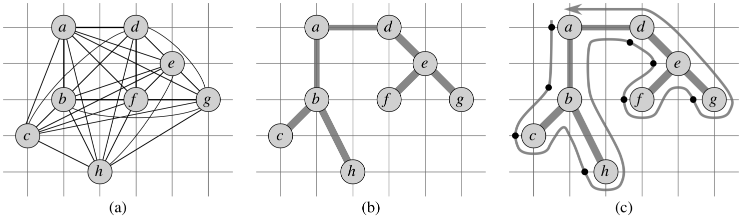
```
APPROX-TSP-TOUR($G, c$)
1  select a vertex $r \in G.V$ to be a root vertex
2  computer a minimum spanning tree $T$ for $G$ from root $r$
   using MST-PRIM($G, c, r$)
3  let $L$ be the list of vertices visited in a preorder tree walk of $T$
4  return the Hamiltonian cycle $H$ that visits the vertices in
   the order $L$
```

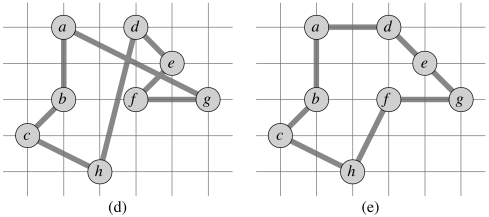
This algorithm returns a tour of no more than twice the cost of an optimal tour assuming the graph satisfies the *triangle inequality*:

For all vertices u, v and w in V

$$c(u, w) \leq c(u, v) + c(v, w)$$

Approximate TSP





Tour (d) is that returned by APPROX-TSP-TOUR, cost 19.074

Tour (e) is an optimal tour, cost 14.715

Approximate TSP

If the graph does not satisfy the triangle inequality, there are no known good, polynomial-time approximation algorithms

Recap

- Computational complexity: important classes of problems:
    - P, NP, NP-complete
- Understanding the limits of what is computationally feasible may save you wasting time (?) trying to find an optimal solution when an approximate solution will have to do
- Reducing one problem to an instance of another is the method-of-choice for proving your problem is “hard”.