CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- PS5 due Dec. 9
- SA6 returned
- PS4 grading update
- Project 4 due 11:59pm, Monday, Dec. 12
- ***Important*** admin notes:
  - Change to syllabus: To be counted for credit, all PSs and SAs (including revisions) must be submitted by *11:59pm, Sunday, Dec. 11* [extended from Dec. 9]
  - Final TA hours of semester: Thursday, Dec. 8
- Additions to our schedule:
  - Make-up lecture: Friday, Dec. 9, at 1pm (Davis 117)
  - Office hours: Monday, Dec. 12, 11am

Acknowledgment: Some ideas in today's notes are influenced by *https://www.cs.princeton.edu/~wayne/cs423/lectures/np-complete*

Which is easier? (pt. 1)

- Which is easier: *finding* a solution, or *checking / verifying* a solution?
- Consider the problem of checking if a number is *composite* (i.e., not a prime number)
  - Input: Integer n
  - Output: Yes if n is composite—if there exists some number $1 < c < n$ such that n is a multiple of c; No otherwise (i.e., n is prime)
- Which is easier…
  - Finding if n = 437,669 is composite? Or…
  - Checking that c = 541 is a factor of n?

This kind of thing can be useful in *cryptography* applications

It's useful to think of this as solving the problem *when you're given a solution*, and all you have to do is confirm it

Thanks to: https://www.cs.princeton.edu/~wayne/cs423/lectures/np-complete

Which is easier? (pt. 1)

- Which is easier: *finding* a solution, or *checking / verifying* a solution?
- Consider the problem of checking if a number is *composite* (i.e., not a prime number)
  - Input: Integer n
  - Output: Yes if n is composite—if there exists some number $1 < c < n$ such that n is a multiple of c; No otherwise (i.e., n is prime)
- Which is easier…
  - Finding if n = 437,669 is composite? Or…
  - Checking that c = 541 is a factor of n?

This kind of thing can be useful in *cryptography* applications

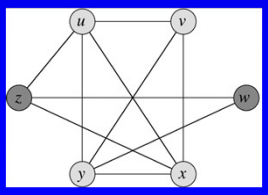
Vocabulary: ***certificate***
- We call the possible solution that was given to us to be checked a certificate

Here, 541 is a *certificate*

Thanks to: https://www.cs.princeton.edu/~wayne/cs423/lectures/np-complete

Which is easier? (pt. 1)

- Which is easier: *finding* a solution, or *checking / verifying* a solution?
- A *clique* in an undirected graph G=(V,E) is a subset V'⊆V s.t. each pair in V' is connected by an edge in E
  - I.e., a clique corresponds to a complete subgraph of G
- The CLIQUE problem
  - Input: graph G and integer k
  - Output: Yes if there is a clique of size k in G, No otherwise



- Which is easier…
  - Finding a clique of size 4 in a graph G? Or…
  - Checking that vertices {u,v,x,y} are a clique of size 4 in graph G?

Vocab review: Here, the set {u,v,x,y} would be a *certificate*

Which is easier? (pt. 2)

- Which is easier: An *optimization* problem, or its related *decision* problem?
- Consider the optimization problem of finding a minimum spanning tree (MST) of a graph
  - Compare that to the related decision problem MST-Decision: Given a graph G and a number w, does G have an MST with total weight at most w?

Which is easier? (pt. 2)

- Which is easier: An *optimization* problem, or its related *decision* problem?
- Consider the optimization problem of finding a minimum spanning tree (MST) of a graph
  - Compare that to the related decision problem MST-Decision: Given a graph G and a number w, does G have an MST with total weight at most w?

Some observations:
1. **If it's easy (fast) to solve the optimization problem, it's easy (fast) to solve the decision problem—just use the optimization problem answer!**
2. **It's not true the other way around! If it's easy (fast) to solve the decision problem for a particular value of w, it's not necessarily easy to solve the optimization problem—you might have to check a lot of different values of w!**

Which is easier? (pt. 2)

- Which is easier: An *optimization* problem, or its related *decision* problem?
- Consider the optimization problem of finding a minimum spanning tree (MST) of a graph
  - Compare that to the related decision problem MST-Decision: Given a graph G and a number w, does G have an MST with total weight at most w?

Some observations:
1. **If it's easy (fast) to solve the optimization problem, it's easy (fast) to solve the decision problem—just use the optimization problem answer!**
2. **It's not true the other way around! If it's easy (fast) to solve the decision problem for a particular value of w, it's not necessarily easy to solve the optimization problem—you might have to check a lot of different values of w!**
3. **So, if the decision problem can't be solved easily, we know the optimization problem can't be solved easily either!** [This is subtle… let's think about it….]

Map Visitation Problem (MVP)

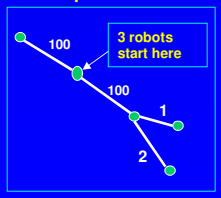
- So… you're at your job, and your supervisor tells you to write an efficient (poly-time) algo to solve the MVP in the general case (not just special cases of it). What do you do? Multiple choice…
  1. Use a minimum-spanning tree approach
  2. Use an all-pairs shortest paths approach
  3. Have a long conversation with your supervisor about ~~wasting~~ better uses of your time

Options 1 and 2 won't work! But before you do option 3, you should *know* you're right…

Input: Integer *k* (number of robots);
weighted undirected graph *G = (V,E)* (map for visitation);
depots $D=\{v_1, \ldots, v_k\}$ (starting points for each robot)

Output: Min-max length of the paths for visiting all nodes in *G*, given starting depots *D*

Example:



Review: Reductions, Subroutines, and Time Complexity

- In general—and in these specific examples—key to using a subroutine is to *meet its specifications*
  - Structure your algorithm to give subroutine the proper input
  - Structure your algorithm to use subroutine's output to solve the original problem
- In these cases, can *reduce* one problem A to another problem B, where there's already a subroutine / algorithm for B
  - Algorithm that does the "reducing" can be called a *reduction*
- *Important but maybe obvious note:*
  - Complexity of resulting algorithm depends on complexity of subroutine

This is not surprising—the complexity of every algorithm depends on the complexity of its parts. I'm just making that point explicit here because we'll use it again later.

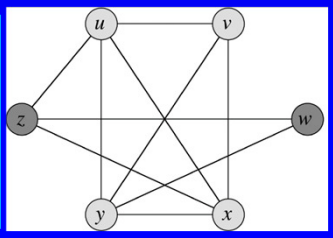
We'll be considering decision problems from here on out, unless otherwise specified

Graph Problems: CLIQUE

- A *clique* in an undirected graph G=(V,E) is a subset V'⊆V s.t. each pair in V' is connected by an edge in E
  - I.e., a clique corresponds to a complete subgraph of G
- The CLIQUE problem
  - Input: graph G and integer k
  - Output: Yes if there is a clique of size k in G, No otherwise

On Project 2, someone on your team designed a brute force algorithm for CLIQUE that was *not* polynomial time (*poly time*, for short) .

Can you design a poly time algorithm for CLIQUE?

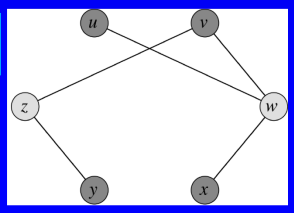


Graph Problems: VERTEX COVER

Also called *Network Cover* on Project 2

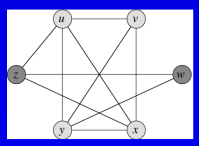
- A *vertex cover* of an undirected graph G=(V,E) is a subset V'⊆V s.t. if (u,v) is in E, then at least one of u or v is in V'
  - That is, if a vertex "covers" its incident edges, a vertex cover is a set that "covers" all of E
  - The size of a vertex cover is the number of vertices in it
- The VERTEX COVER problem
  - Input: graph G and integer k
  - Output: Yes if there is a vertex cover of G with size k, No otherwise

Graph for which vertex set {w, z} is a vertex cover of size 2.

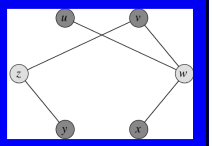


Graph Problems and Subroutines / Reductions

- Recall, focusing only on decision problems
- The CLIQUE problem
  - Input: graph G and integer k;
    Output: Yes if there is a clique of size k in G,
    No otherwise



- The VERTEX COVER problem
  - Input: graph G and integer k;
    Output: Yes if there is a vertex cover of G with size k,
    No otherwise

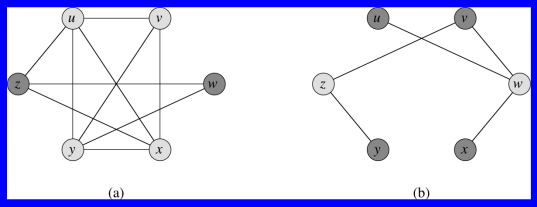


- How could we use a solution for one of these as a subroutine to solve the other?
  - For example, assume we had a solution for VERTEX COVER. How could we use that to create a solution for CLIQUE?

Reduction from CLIQUE to VERTEX COVER

- For a graph G=(V,E) we can consider the *complement* of G: ~G=(V,~E)
  - That is, the complement of G consists of the same vertices as in G and all the edges (pairs of vertices, unordered) not in G
- Given an instance (G_c,k_c) of CLIQUE, construct instance (G_{vc},k_{vc}) of VERTEX COVER, where
  - G_{vc} is the complement of G_c
  - k_{vc} is |V| - k_c, that is, the number of vertices *not in* the (possible) clique for the CLIQUE instance

Is this a polynomial time reduction?



What does this reduction say about the complexity of VERTEX COVER?

- Assume that CLIQUE *cannot* be solved in polynomial time

This is a huge assumption! Note that we're not just saying *one algorithm* is not poly time. We're saying *every algorithm* is not poly time! We're talking about the complexity of *the problem*, not just of an algorithm.

- Then, because we can reduce CLIQUE to VERTEX COVER in poly time…
  - we know VERTEX COVER *cannot* be poly time, either!
  - How do we know? Argument by contradiction:
    - If VERTEX COVER could be solved in poly time, we'd use that poly time algorithm as our subroutine…
    - Then we'd be able to solve CLIQUE in poly time by this reduction
    - But CLIQUE (by assumption at the beginning of this slide) can't be poly time, so our assumption that VERTEX COVER could be poly time must be false!

This is deep.

And useful.

Tractability and Complexity

- Tractability
  - It is generally accepted that a problem is *tractable* (solvable in practice, not just in theory) if there is a polynomial-time algorithm for it
  - … i.e., in class $O(n^k)$ for some constant k
  - If a problem requires an exponential time solution (or worse!), it is considered intractable—without an efficient solution
- Goal: Classify *problems* by how efficiently they can be solved
  - Big-O or Theta classes: (Relatively) fine-detail distinctions
  - Tractability distinctions based in larger classes:
    - P: problems solvable in *polynomial* time (hence the name *P*)
    - NP: problems with solutions *verifiable* in polynomial time
    - PSPACE : problems solvable in polynomial space (no restriction on time)
    - EXPTIME: problems solvable in exponential time
    - EXPSPACE: problems solvable in exponential space

More about verifiability, coming soon!

Etc., etc., etc. …

The Class NP:
A Quick Introduction

- NP: the class of problems solvable in *nondeterministic polynomial* time (hence the name *NP*)
  - Somewhat loosely, that means that a problem is in NP if it can be "verified" in polynomial time…
    - Could think of it as: If we were given a *certificate* of a solution (essentially, a potential solution), we could check it for correctness (is it actually a solution?) in ***polynomial time in input size***
    - Could think of it as: If given a "correct" certificate, we could solve the problem in polynomial time
  - Note that if a solution to a problem can be *found* in polynomial time, that problem can be verified in polynomial time

Recall: This is about decision problems, not (related) optimization problems

  - What does this say about the relationship between P and NP?
  - (For more about nondeterminism in computation, see CS378!)

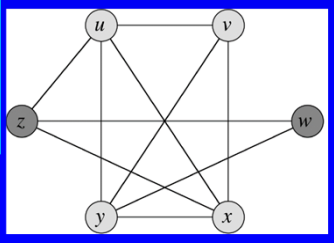
CLIQUE is in NP

- A *clique* in an undirected graph G=(V,E) is a subset V'⊆V s.t. each pair in V' is connected by an edge in E
  - I.e., a clique corresponds to a complete subgraph of G

We'll focus on |V'| > 1. Do you see why?

- The CLIQUE problem
  - Input: graph G and integer k
  - Output: Yes if there is a clique of size k in G, No otherwise

CLIQUE is in NP!
What would a *poly-time verification algorithm* take as input?
- **What would the *certificate* be?**
- **What would the verification algorithm do, using the certificate?**

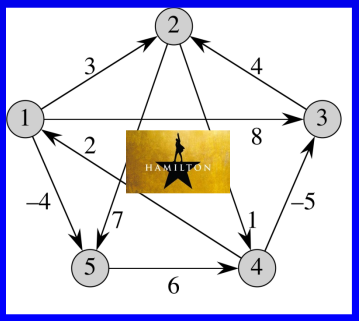


HAMILTONIAN CYCLE is in NP

Also called *Hamiltonian Tour* on Project 2

In some sources, "Hamiltonian" is capitalized, but not in CLRS

- Consider the hamiltonian cycle problem:
  - A *hamiltonian cycle* of a connected, directed graph G=(V,E) is a *simple* cycle that contains each vertex in V (though perhaps not every edge in E)
  - The hamiltonian cycle problem: Given connected digraph G, does G contain a hamiltonian cycle?
- *Example*: Is there a hamiltonian cycle in this graph?



Historical Note: Hamiltonian cycle isn't named after *that* Hamilton.

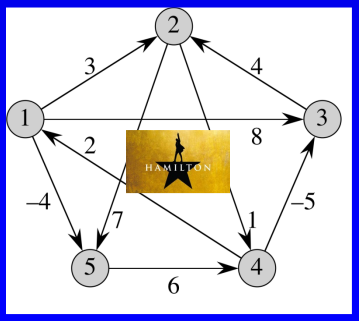
(Less Historical Note: I have a great idea for an NP-Completeness musical. Lin-Manuel, call me!)

(Although nobody knows for sure, experts believe that an NP-Completeness musical would run for a very, very long time!)

HAMILTONIAN CYCLE is in NP

In some sources, "Hamiltonian" is capitalized, but not in CLRS

- Consider the hamiltonian cycle problem:
  - A *hamiltonian cycle* of a connected, directed graph G=(V,E) is a *simple* cycle that contains each vertex in V (though perhaps not every edge in E)
  - The hamiltonian cycle problem: Given connected digraph G, does G contain a hamiltonian cycle?
- *Example*: Is there a hamiltonian cycle in this graph?



HAMILTONIAN CYCLE is in NP!
What would a poly-time verification algorithm take as input?
- What would the *certificate* be?
- What would the verification algorithm do, using the certificate?

Which is easier? (pt. 3)

- Recall the Hamiltonian cycle problem:
  - A H*amiltonian cycle* of a connected, directed graph G=(V,E) is a *simple* cycle that contains each vertex in V (though perhaps not every edge in E)
  - The Hamiltonian cycle problem: Given connected digraph G, does G contain a Hamiltonian tour?
- Consider the Euler tour problem:
  - An *Euler tour* of a connected, directed graph is a cycle that uses each edge exactly once (though it can visit vertices multiple times)
  - The Euler tour problem: Given connected digraph G, does G contain an Euler tour?
- How long (what complexity) would it take to:
  - … solve each of these problems?
  - … verify that a possible solution was correct, if we were given one?
  - And is there a meaningful difference in tractability between the problems?

No joke: This is an exceptionally important slide.

P, NP, and NP-Completeness

- We know P ⊆ NP, because if a solution can be found in polynomial time, one can be checked in polynomial time
- Is NP ⊆ P? Good question. **(One of the best around, actually.)**

Extra credit exercise: Prove or disprove NP ⊆ P. (You would get an A+ for this course. Oh, and also $1,000,000… at least. Really.)

  - It is generally believed that P≠NP—problems in P are tractable, and NP-problems (not in P) are thought to be intractable
  - Thus, it's important to determine if a problem is in NP, or at least as hard as a problem that is in NP (and not known to be in P)
- Complexity class *NPC*: Class of *NP-complete* problems
  - A problem is NP-complete if a) it is in NP; and b) it is at least as hard as every problem in NP
  - So, if one NP-complete problem is tractable, *all* problems in NP are tractable!

NPC problems are the hardest problems in NP—thus, they're presumed intractable

Using Reductions To Show Intractability

… at least, we think they're intractable…

- If we know problem B has no polynomial time algorithm, then consider a reduction from B to A:
  - Takes an instance **β** of B
  - Transforms it in polynomial time into an instance **α** of A…
  - … such that a decision on **α** would give us a decision on **β**
- Then, if A could be solved in polynomial time, B could also
- Thus, by contradiction, A must not be solvable in polynomial time
- More generally: A is *at least as hard* (to solve in poly time) as B
- A similar procedure will be used to show NP-completeness:
  - If B is NP-complete, A must be at least as difficult (to solve in poly time), so A is a candidate for being NP-complete
  - This is a common use of polynomial-time reductions

NP-Completeness

- More formally / concisely, here's what we mean by NP-completeness (complexity class NPC):
  - L ∈ NPC iff
    1. L ∈ NP
    2. For *every* L' in NP, L' reduces to L
- An implication:
  - If L is NP-complete and L is in P, then …?

- **If L satisfies 2., it is called *NP-hard*.**
- **If L is NP-hard and also in NP, then it's NP-complete.**

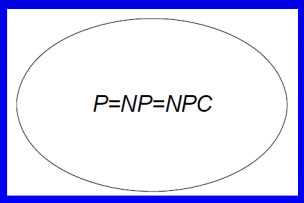
NP-Completeness

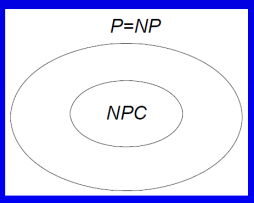
- More formally / concisely, here's what we mean by NP-completeness (complexity class NPC):
  - L ∈ NPC iff
    1. L ∈ NP
    2. For *every* L' in NP, L' reduces to L
- An implication:
  - If L is NP-complete and L is in P, then P = NP
  - *So*, if P ≠ NP, then if L is NP-complete, L is not in P

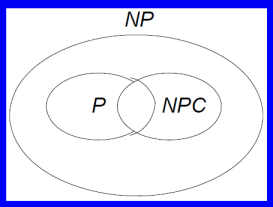
- **If L satisfies 2., it is called *NP-hard*.**
- **If L is NP-hard and also in NP, then it's NP-complete.**

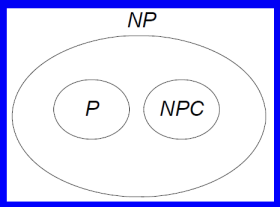
We think P ≠ NP. So, we believe all NP-Complete problems are not possible to solve in poly time. But we don't know for sure!

Which of these represents something possible about P and NP?









List of NP-Complete Problems (partial list; from Wikipedia)

https://en.wikipedia.org/wiki/List_of_NP-complete_problems

- Graphs and hypergraphs

- Graphs occur frequently in everyday applications. Examples include biological or social networks, which contain hundreds, thousands and even billions of nodes in some cases (e.g. Facebook or LinkedIn).

- 1-planarity[1]
- 3-dimensional matching[2][3]
- Bipartite dimension[4]
- Capacitated minimum spanning tree[5]
- Route inspection problem (also called Chinese postman problem) for mixed graphs (having both directed and undirected edges). The program is solvable in polynomial time if the graph

- Games and puzzles

- Battleship
- Bulls and Cows, marketed as Master Mind: certain optimisation problems but not the game itself.
- Eternity II
- (Generalized) FreeCell[51]
- Fillomino[52]
- Hashiwokakero[53]
- Heyawake[54]
- (Generalized) Instant Insanity[55]
- Kakuro (Cross Sums)
- Kuromasu (also known as Kurodoko)[56]
- Lemmings (with a polynomial time limit)[57]
- Light Up[58]

List of NP-Complete Problems (partial list; from Wikipedia)

https://en.wikipedia.org/wiki/List_of_NP-complete_problems

Point being, there are a lot of them….

… and the Map Visitation Problem, which is NP-Complete, isn't even on this list!

Map Visitation Problem (MVP)

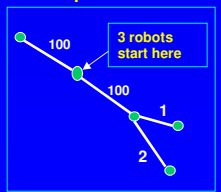
- So… you're at your job, and your supervisor tells you to write an efficient (poly-time) algo to solve the MVP in the general case (not just special cases of it). What do you do? Multiple choice…
  1. Use a minimum-spanning tree approach
  2. Use an all-pairs shortest paths approach
  3. Have a long conversation with your supervisor about ~~wasting~~ better uses of your time

Options 1 and 2 won't work! But before you do option 3, you should *know* you're right… and NP-Completeness can help!

Input: Integer *k* (number of robots);
weighted undirected graph *G = (V,E)* (map for visitation);
depots $D=\{v_1, \ldots, v_k\}$ (starting points for each robot)

Output: Min-max length of the paths for visiting all nodes in *G*, given starting depots *D*

Example:



Managing NP-Completeness

NP-Completeness is everywhere! How do we deal with it?

- Focus on special cases that we *can* efficiently solve
  - A lot of important problems fall into this category!
  - Can work on *approximation algorithms* instead of exact solutions
- Use intractability to your benefit
  - Example: Cryptography
- Keep trying to prove P = NP
  - (Good luck!)

Thanks to: https://www.cs.princeton.edu/~wayne/cs423/lectures/np-complete

Headlines!

~~Coming Up on~~ A Recap Of CS375…

- This is a lot!
- Thank you for your work this semester!

- *What's the fastest sorting algorithm?*
  - Is it quicksort?
  - The answer might surprise you! (Or it might not!)
- *All the ~~hottest fashion trends for programmers~~ most important complexity classes for programmers!*
  - With essential algos from each one!
- *Common myths about recursion, debunked!*
  - (Some of the answers here *actually* might surprise you!)
- *How to make an algorithm not just a little faster, but a lot faster!*
- *Could **your next problem** be **NP-Complete**? And how would you know?*

- Plus… *language / NLP applications*, a little peek into *what a compiler / interpreter does*, and *more*!