Complexity Theory

Programme

- Graph coloring: a practical application

  This problem will serve as our touchstone throughout the

  lecture as we attempt to ground our discussion of complexity theory.

- Key concepts and terms:

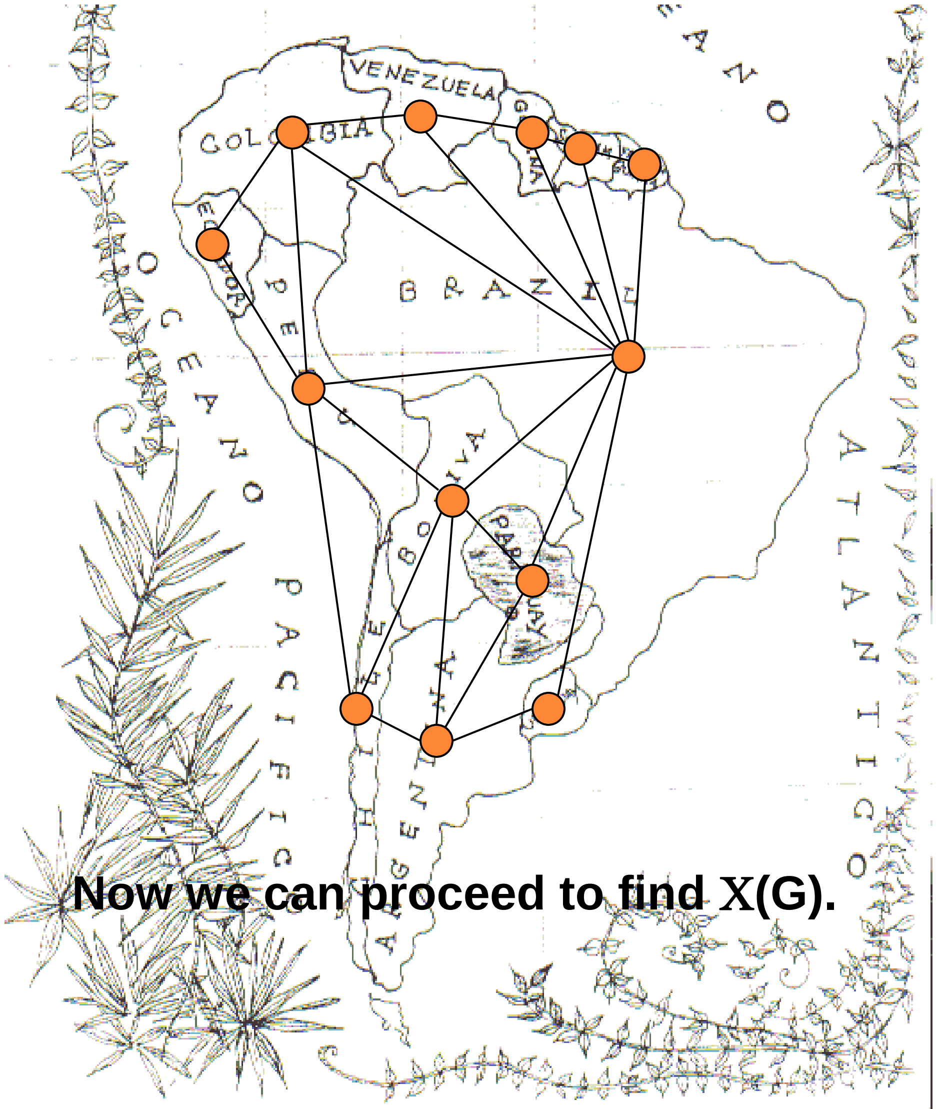
  Polynomial runtime, decision problem, determinism, and reducibility

- Introduction to problem classes:

  P, NP, and NP-Complete

- Implications and further considerations

The Map Coloring Problem



Now we can proceed to find Χ(G).

The Question:

What is the minimum number of colors necessary to color this map so that no two bordering countries share the same color?

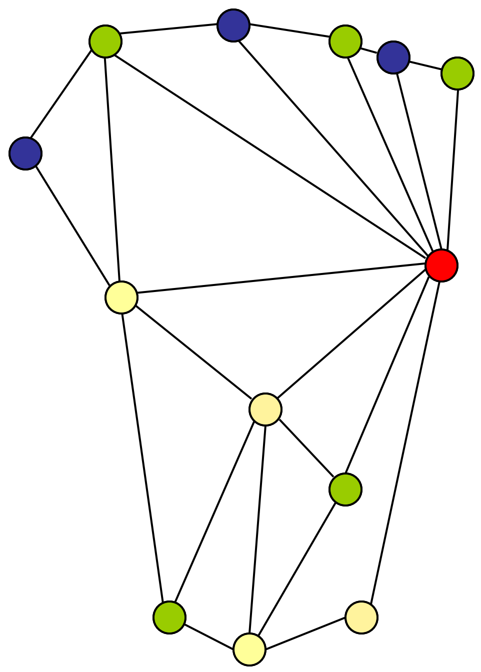
One Approach:

Model the map using a graph in a way that allows us to find the answer to this question by finding the chromatic number[1] of the graph.

So how can we do this?

[1] Recall: the *chromatic number* of G, Χ(G), is the minimum number of colors needed to ensure that no two adjacent vertices in G have the same color.

The Map Coloring Problem



It turns out that $\chi(G) = 4$

Now that we have our graph:

- How might we begin our inquiry?
  - Is G 2-colorable? 3-colorable? 4-colorable?
  - What is its min or max degree?
  - Does it have an embedded K_c (complete graph on c vertices)?
- How do we determine these questions (e.g., whether a graph is 3-, 4-, or *K*-colorable)? What *is* the algorithm?
- How *efficient* of an algorithm can we discover to answer these questions?

For arbitrary graphs, even the most efficient algorithms for determining three- or four-colorability (and beyond) essentially come down to this:

Try a LOT of combinations!

K-COLORING AND ITS COMPLEXITY

- A perfectly valid (although inefficient) approach: try all possible colorings!
- For a graph with *n* vertices, and a palette of some number *k* of colors, how many possible colorings (valid or not) are there?
  - Answer: k^n
- This doesn't seem like an efficient approach! It's certainly not polynomial.
- **Definition 1.** An algorithm has *polynomial* runtime if that runtime can be expressed as $O(n^k)$ for some constant *k*, where *n* is the input size.
  - So these are polynomial runtimes:
    - $T(n) = n^{10} + 3n^5 + 60$
    - $T(n) = n \log n$
  - But these are not:
    - $T(n) = n! + n$
    - $T(n) = 2^n + 4n^2 + 12$

Why hate on non-polynomial runtimes?

	n	n^2	n^3	n^{10}	2^n	$n!$	n^n
1 hr	3.6×10^{15}	6.0×10^{7}	1.5×10^{5}	35	51	17	13
10 hrs (~1/2 day)	3.6×10^{16}	1.8×10^{8}	3.3×10^{5}	45	54	18	14
100 hrs (~ 4 days)	3.6×10^{17}	6.0×10^{8}	7.1×10^{5}	56	58	19	14
1,000 hrs (~ 1 month)	3.6×10^{18}	1.8×10^{9}	1.5×10^{6}	71	61	20	15
10,000 hrs (~ 1 year)	3.6×10^{19}	6.0×10^{9}	3.3×10^{6}	90	64	20	16
100,000 hrs (~ 1 decade)	3.6×10^{20}	1.8×10^{10}	7.1×10^{6}	113	68	21	16

An estimate of the size of input that can be handled in the given time for an algorithm of the given order. These values will of course evolve as processing power increases.

Taken from Johnsonbaugh and Schaefer's 2004 textbook, *Algorithms* (Prentice Hall).

Toward the Study of Complexity

- But some problems just don’t seem to be willing to yield solutions (algorithms) that run in polynomial time (e.g., the graph coloring problem.)
- So people started asking...
  - Why can’t we seem to find polynomial solutions to certain problems?
  - What are the best algorithms we can come up with to solve them?
  - What are some of the properties shared by these problems?
- To enable this inquiry, theoreticians began dividing problems into classes.
- And so complexity theory arrived on the scene. But before we dive in...

A PRELIMINARY DEFINITION

- **Definition 2.** A *decision problem* is a question (or problem) for which all instances of the problem (or inputs) have either a yes or no answer.
- Examples:
  - Is a given graph 4-colorable?
  - Does a string *A* contain a string *B* as a substring?
  - Is a set *Y* a proper subset of a set *Z*?
- Note that many problems can be massaged into the form of a decision problem. For example, instead of asking for the chromatic number of a graph, we might instead ask whether it is *k*-colorable for the integer *k*.
- Questions that compute values, rather than making yes or no decisions, are called *function problems*. (This is tangential to our discussion, though.)

The first and most obvious class

- **Definition 3. P** is the class of decision problems that can be decided (solved) in polynomial time.
- Examples:
  - Is an array *X* in sorted order?
  - In a sorted array of integers *A*, are there two integers that sum to a target *T*?
  - Is a given graph *G* two-colorable?
- But we already have solutions with polynomial runtimes for the problems in **P**, and so they aren't all that interesting to us in the context of the current discussion.
- Before we can move on to the next class of problems...

An Interlude on the Topic of Determinism

- **Definition 4.** A process is said to be *deterministic* if, given the same input, it always produces the same output, and passes through the same states in order to arrive at that output.
- Example: Consider a simple program that you may have written in your first programming course: it asks for two integer inputs, prints the sum of those integers, and terminates.
- The program always follows a prescribed recipe; it never deviates from the plan. Furthermore, given the same two integers, it always produces the same output (barring a hardware error or other strange glitch).
- We might observe that it’s difficult to conceive of a non-deterministic program in our traditional model of computation.

Approaching non-determinism

- Question: Is a program that prints *n* random integers to the screen (where *n* is some integer input by the user) deterministic or not?
- Answer: Yes. And no. Really, it depends.
- If you can write a program that truly produces random numbers, then the program will indeed be non-deterministic.
- Most programs do not exhibit any sort of truly random behavior. The appearance of random behavior comes from a pseudo-random number generator, which is actually a deterministic process. Given the same seed (typically the current time), it will always produce the same sequence of “random” numbers as a result.
- Here’s the rub: that seed, the current time, is implicitly part of the input to the system. So yes, the output is changing each time the program is executed, but so is the input.

WHERE'S THE NON-DETERMINISM?

- It's certainly not on our computers.
- Any *truly* random behavior (does it exist?) would be non-deterministic.
- Is the mind a deterministic machine? (Or rather, "machine.")
  - The 0-1 thought experiment
- We can conceive of abstract functions that behave non-deterministically.
- Example: Let us return to the problem of determining whether a graph is *k*-colorable. Imagine a function that could go to each vertex and guess (following no discernible process) what color each vertex should get. Also suppose that it always guesses right, and that it can do so in constant time. Then certainly, this function is non-deterministic!

Classes of Problems Again

- **Definition 3a. P** is the class of decision problems that can be decided (solved) *deterministically* in polynomial time. (I left out that detail before.)
- **Definition 5. NP** is the class of decision problems that can be decided *non-deterministically* in polynomial time, and for which a solution can be verified *deterministically* in polynomial time.
- Example: 3-Colorable Problem
- Algorithm:

```
for each vertex v
    color[v] = guess{1..k}
for each vertex v
    for ever other vertex w
        if (v is adjacent to w) and (color[v] == color[w])
            return false
return true
```

- The two parts of Definition 5 are actually equivalent, but this is beyond the scope of our discussion.

Problem reducibility

- **Definition 6.** The decision problem A is *reducible* to the decision problem B if there is some function *f* (a deterministic polynomial time algorithm) that transforms instances of A into instances of B; and furthermore, if *f*(*a*) = *b*, then the answer to *b* is yes if and only if the answer to *a* is yes (where *a* is an instance of A, and *b* is an instance of B).

- This is a lot less intimidating than it looks. In other words, we transform the question to allow ourselves to ask a different question.
  - A ***very*** loose analogy: recall how in the first few slides we transformed a map into a graph and re-formed the question to ask whether G was k-colorable.

- Since we can transform instances of A into instances of B, then if we can solve B, we have essentially solved A. Thus, we tend to say that B is at least as hard as A.

- These reductions can get very complex, and are beyond the scope of this discussion. The important thing to take away here is the concept.

A FINAL CLASS OF PROBLEMS

- **Definition 7. NP-Complete** is the class of decision problems that are *complete* for **NP**. That is, for a problem *X* to be **NP-Complete**:
  - *X* must be in **NP**
  - Every problem in **NP** is reducible to *X*
- Or in other words:
  - The problems in **NP-Complete** are all **NP**
  - Every problem in **NP** is reducible to every problem in **NP-Complete**
- Examples of **NP-Complete** problems:
  - 3-Coloring Problem
  - Traveling Salesman Problem
  - Hamiltonian Cycle Problem
- Other classes exist that you might be interested to investigate, although we do not have time to discuss them here:
  - **NP-Hard**, **Co-NP**, **Co-NP-Complete**, **EXP**, and many more

Implications and Discussion

- What if we found a polynomial runtime solution to a problem in **NP**?
  - Then we simply reassign it to **P**
- What if we found a polynomial solution to an **NP-Complete** problem?
  - Then we would have **P** = **NP**. (Why?)
- What are the chances that **P** = **NP**?
  - Because so many brilliant people have spent so much time trying to discover polynomial run-time algorithms for these problems and been unsuccessful, many people believe that **P** is NOT equal to **NP**.
- Note that sometimes a minor change in the specification of a problem can send it to another class altogether.
  - Example: The 2-colorable problem is in **P**, but the 3-colorable problem is in fact **NP-Complete**.
- If you encounter a problem from **NP** in the real world, you shouldn't get too down on yourself for not being able to find a polynomial solution to it.

The take-away from today's lecture

- **P** is the class of problems with known solutions that run deterministically in polynomial time.
- **NP** is the class of problems ...
- **NP-Complete** is the class of problems ...
- We're not sure if **P** = **NP**.
- If any single **NP-Complete** problem has a deterministic polynomial time solution, then certainly, **P** = **NP**.
- Can you remember some examples of problems in each of these classes?
- Do you recall the definitions of *polynomial runtime*, *decision problem*, *determinism*, and *reducibility*?
- If you know that a problem is in **P**, what can you say about it? What about a problem in **NP**? **NP-Complete**?