Lecture # 18

Complexity Theory

Complexity Theory

- So far in the course, we have been building up a"bag of tricks" for solving algorithmic problems i.e. what sort of design paradigm should be used: divide-and-conquer, greedy, dynamic programming?

- What sort of data structures might be relevant: trees, heaps, graphs? What is the running time of the algorithm? All of this is fine if it helps you discover an acceptably efficient algorithm to solve your problem.

Complexity Theory

- The question that often arises in practice is that you have tried every trick in the book and nothing seems to work. Although your algorithm can solve small problems reasonably efficiently (e.g., $n \leq 20$), but for the really large problems you want to solve, your algorithm never terminates.

- When you analyze its running time, you realize that it is running in exponential time, perhaps $n^{\sqrt{n}}$, or 2^n, or 2^{2n}, or $n!$ or worse!.

Complexity Theory

- By the end of 60's, there was great success in finding efficient solutions to many problems. But there was also a growing list of problems for which there seemed to be no known efficient algorithmic solutions.

- People began to wonder whether there was some *unknown paradigm* that would lead to a solution to these problems. Or perhaps some proof that these problems are inherently hard to solve and no algorithmic solutions exist that run *under exponential time*.

Complexity Theory

- Near the end of the 1960's, a remarkable discovery was made. Many of these hard problems were interrelated in the sense that if you could solve any one of them in polynomial time, then you could solve all of them in polynomial time.
- This discovery gave rise to the notion of *NP-completeness*.

- This area is a radical departure from what we have been doing because the emphasis will change. The goal is no longer to prove that a problem *can* be solved efficiently by presenting an algorithm for it. Instead we will be trying to show that a problem *cannot* be solved efficiently.

Complexity Theory

- Up until now all algorithms we have seen had the property that their worst-case running time are bounded above by some *polynomial* in **n**.

- A *polynomial time algorithm* is any algorithm that runs in $O(n^k)$ time. A problem is solvable in polynomial time if there is a polynomial time algorithm for it.

- Some functions that do not look like polynomials (such as O(n log n) are bounded above by polynomials (such as $O(n^2)$).

Complexity Theory

- Some functions that do look like polynomials but they are not. For example, suppose you have an algorithm that takes as input a graph of size n and an integer k and run in $O(n^k)$ time.
- Is this a polynomial time algorithm?

- No, because k is an input to the problem so the user is allowed to choose k= n, implying that the running time would be $O(n^n)$.

- $O(n^n)$ is surely not a polynomial in n. The important aspect is that the exponent must be a constant independent of n.

Decision Problems

- Most of the problems we have discussed involve optimization of one form of another. For e.g. Finding the shortest path, finding the minimum cost spanning tree and maximize the knapsack value etc.
- For rather technical reasons, the NP-complete problems we will discuss will be phrased as *decision problems.*
- A problem is called a *decision problem* if its output is a simple "yes" or "no" (or you may this of this as true/false, 0/1, accept/reject.)
- We will phrase many optimization problems as decision problems. For example, the MST decision problem would be: Given a weighted graph G and an integer k, does G have a spanning tree whose weight is at most k?

Complexity Classes

- Class P: This is the set of all decision problems that can be *solved* in polynomial time. We will generally refer to these problems as being "easy" or "efficiently solvable".

- Class NP: This is the set of all decision problems that can be *verified* in polynomial time. This class contains P as a subset. It also contains a number of problems that are believed to be very "hard" to solve.
- The term "NP" does not mean "not polynomial". Originally, the term meant "non-deterministic polynomial" but it is a bit more intuitive to explain the concept from the perspective of verification.

Complexity Classes

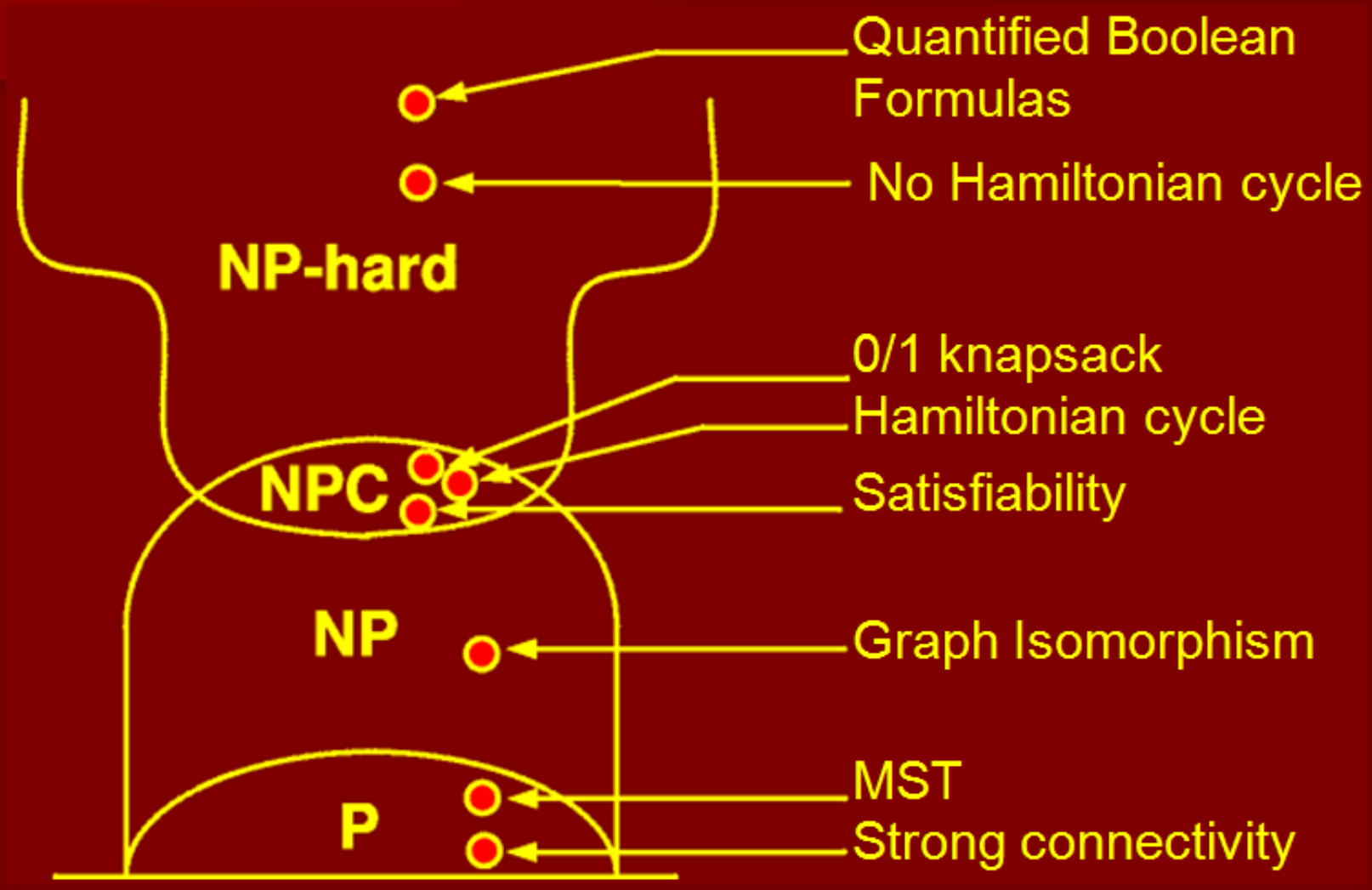
- Class NP-hard: In spite of its name, to say that a problem is NP-hard does not mean that it is hard to solve. Rather, it means that if we could solve this problem in **polynomial** time, then we could solve all **NP** problems in polynomial time.
- Note that for a problem to NP-hard, it does not have to be in the class NP.

- Class NP-complete: A problem is NP-complete if (1) it is in NP and (2) it is NP-hard.

Complexity Classes

- The following Figure illustrates one way that the sets P, NP, NP-hard, and NP-complete (NPC) might look.
- We say might because we do not know whether all of these complexity classes are distinct or whether they are all solvable in polynomial time.
- The Graph Isomorphism, which asks whether two graphs are identical up to a renaming of their vertices. It is known that this problem is in NP, but it is not known to be in P. The other is QBF, which stands for Quantified Boolean Formulas. In this problem you are given a Boolean formula and you want to know whether the formula is true or false.

Complexity Classes

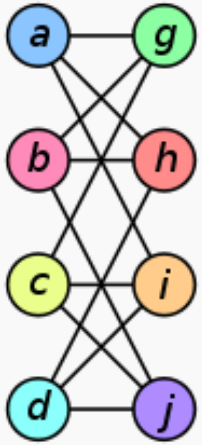
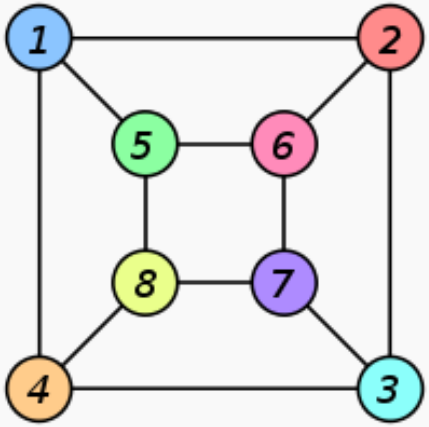


Graph Isomorphism

- In graph theory, an isomorphism of graphs *G* and *H* is a bijection between the vertex sets of *G* and *H*

f: V(G) → V(H)

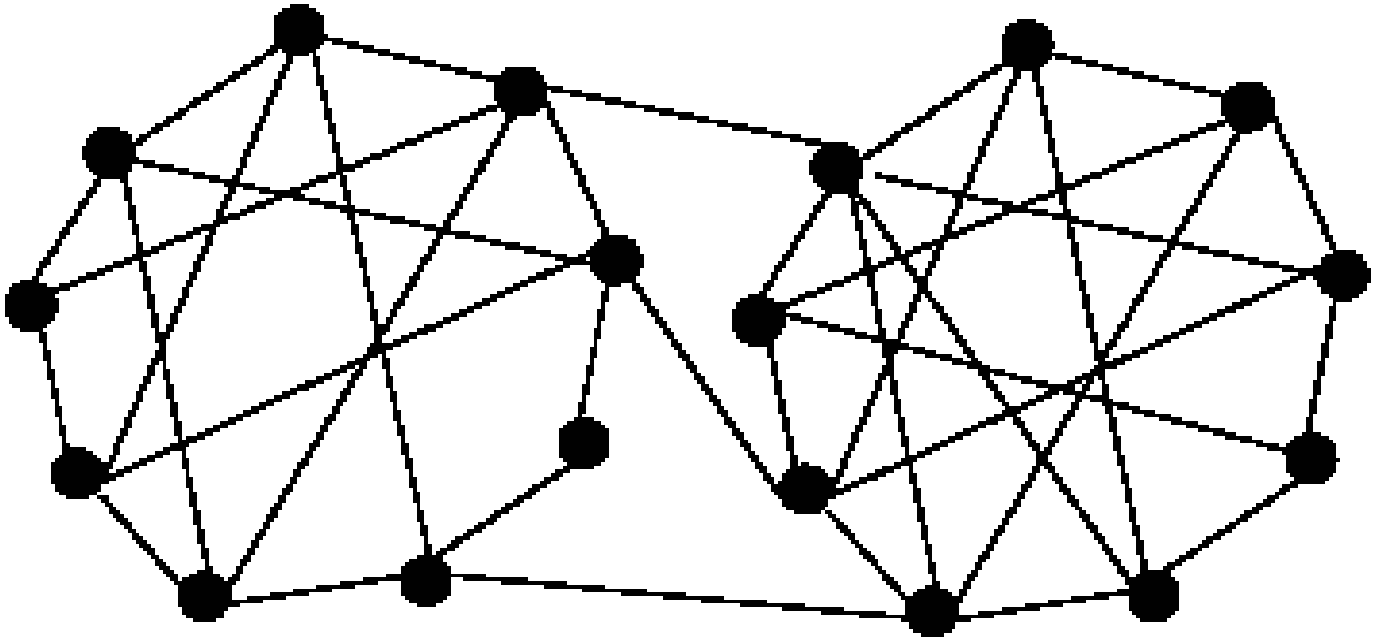
such that any two vertices *u* and *v* of G are adjacent in *G* if and only if *f*(*u*) and *f*(*v*) are adjacent in *H*

Graph G	Graph H	An isomorphism between G and H
		*f*(*a*) = 1 *f*(*b*) = 6 *f*(*c*) = 8 *f*(*d*) = 3 *f*(*g*) = 5 *f*(*h*) = 2 *f*(*i*) = 4 *f*(*j*) = 7

Polynomial Time Verification

Verification

If you tell me that this graph is 3-colorable,

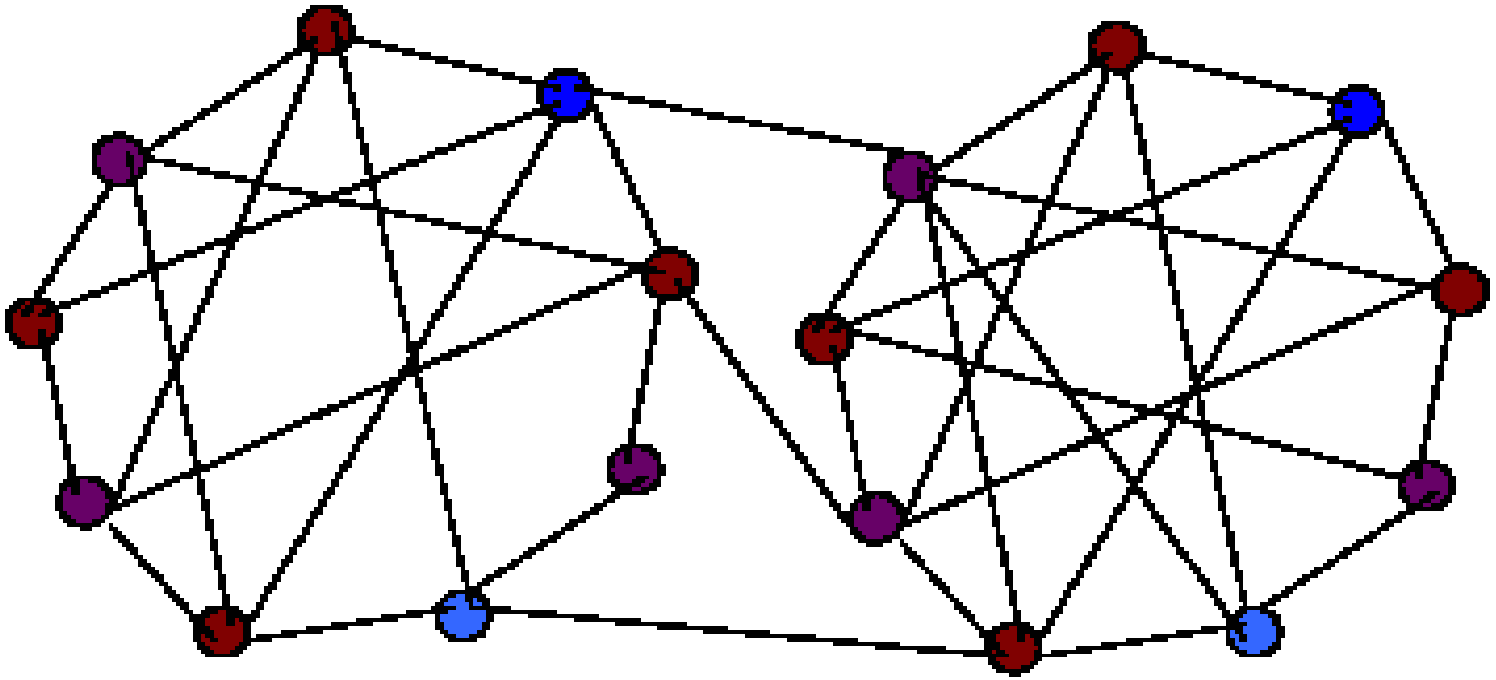


it is very difficult for me to check whether you are right.

Polynomial Time Verification

Verification

But if you tell me that this graph is 3-colorable and give me a solution, it is very easy for me to verify whether you are right.

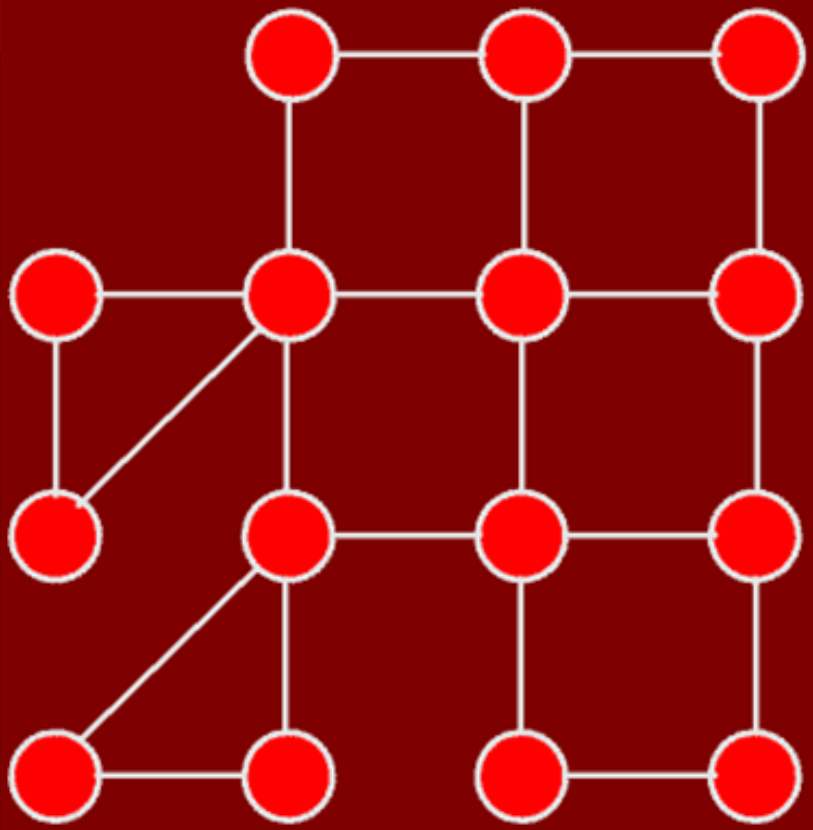


Polynomial Time Verification

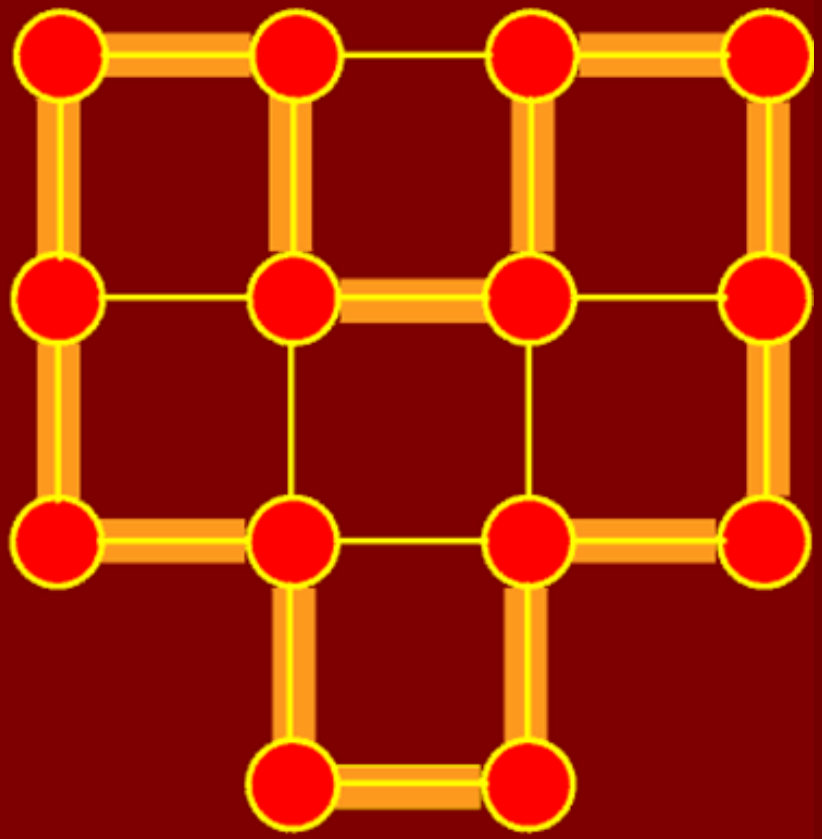
- Before talking about the class of NP-complete problems, it is important to introduce the notion of a *verification algorithm.*
- Many problems are hard to solve but they have the property that it *easy to verify the solution* if one is provided. Consider another example : **Hamiltonian cycle problem**.

- Given an undirected graph G, does G have a cycle that visits every vertex exactly once? There is no known polynomial time algorithm for this problem.

Polynomial Time Verification



Non-Hamiltonian



Hamiltonian

Polynomial Time Verification

- The piece of information that allows verification is called a ***certificate***.

- Note that not all problems have the property that they are easy to verify. For example, consider the following two:
  - UHC ={ (G) | G has a Unique Hamiltonian Cycle}
  - ¬HC ={ (G) | G has no Hamiltonian Cycle}

The Class NP

- The class NP is a set of all problems that can be verified by a polynomial time (VP) algorithm. Why the set is called "NP" and not "VP"? The original term NP stood for *non-deterministic polynomial time.*
- This referred to a program running on a non-deterministic computer that can make guesses. Such a computer could non-deterministically guess the value of the certificate and then verify it in polynomial time.
- Observe that $P \subseteq NP$. In other words, if we can solve a problem in polynomial time, we can certainly verify the solution in polynomial time.
- More formally, we do not need to see a certificate to solve the problem; we can solve it in polynomial time anyway.

The Class NP

- However, it is not known whether P = NP. It seems unreasonable to think that this should be so.
- Being able to verify that you have a correct solution does not help you in finding the actual solution.
- The belief is that P ≠ NP but no one has a proof for this as well.

Coping with NP-Completeness

- With NP-completeness we have seen that there are many important optimization problems that are likely to be quite hard to solve exactly.
- Since these are important problems, we cannot simply give up at this point, since people do need solutions to these problems.
- Here are some strategies that are used to cope with NP-completeness:

Coping with NP-Completeness

- Use brute-force search: Even on the fastest parallel computers this approach is viable only for the smallest instance of these problems.
- Heuristics: A heuristic is a strategy for producing a valid solution but there are no guarantees how close it to optimal. This is worthwhile if all else fails.
- General search methods: Powerful techniques for solving general combinatorial optimization problems. Branch-and-bound, A*-search, simulated annealing, & genetic algorithms
- Approximation algorithm: This is an algorithm that runs in polynomial time (ideally) and produces a solution that is within a guaranteed factor of the optimal solution.

- **Combinatorial optimization** is a topic in theoretical computer science and applied mathematics that consists of finding the least-cost solution to a mathematical problem in which each solution is associated with a numerical cost. In many such problems, exhaustive search is not feasible.

Thank You …

&

best of luck for exams