Title

Computer Science 604

Advanced Algorithms

Lecture 1: Introduction

David Juedes

School of EECS

juedes@cs.ohiou.edu

Purpose of this course

- Introduction to Advanced Algorithmic Techniques for those who want to perform research in ALGORITHMS and COMPUTATIONAL COMPLEXITY.
- Introduction to Advanced Algorithmic Techniques for those who want to apply these techniques to specific application areas (e.g., RESEARCH in Bioinformatics)
- Introduction to Advanced Algorithmic Techniques for those who want to apply those techniques to programming challenges (e.g., FaceBook Puzzles, Google CodeJam, Facebook Hacker Cup, TopCoder Open).

Introduction to Algorithms Research

Top Conferences:

- U.S.: STOC (ACM), FOCS (IEEE), SODA (SIAM), CCC
- Europe: ICALP, STACS, SWAT, WG

Top Journals:

- Journal of the ACM, Computational Complexity, Theoretical Computer Science, Algorithmica, SIAM Journal on Computing, Journal of Computer and System Sciences, Journal of Algorithms, Theory of Computing Systems / Mathematical Systems Theory - MST, ACM Transactions on Algorithms - TALG

Introduction

The study of algorithms and complexity forms the foundation of much recent research in computer science.

Ex: Factorization, Crytography (RSA), Genome Analysis, etc.

Algorithms research is particularly exciting since it forms the boundary between implementation, practice, and pure theoretical research.

What is more exciting is that there is still much work to be done!!!!!

Intro, cont'd

Algorithms research is almost old as modern mathematics. An early example of this is Euclid's algorithm for GCD, which is 3000 years old. However, modern research in algorithms and complexity did not begin until the 50's and 60's when the notions of "complexity" and "efficient algorithm" were (more or less) formalized.

Research in algorithms seeks to answer the following types of questions.

Types of Questions

1. Is there a better algorithm for problem X? (Is there a more efficient algorithm? Is there an algorithm with a better performance ratio, etc?)

2. Why can't we find a better (or good) algorithm for problem X?

The search for answers has led to some of the more interested work in computer science.

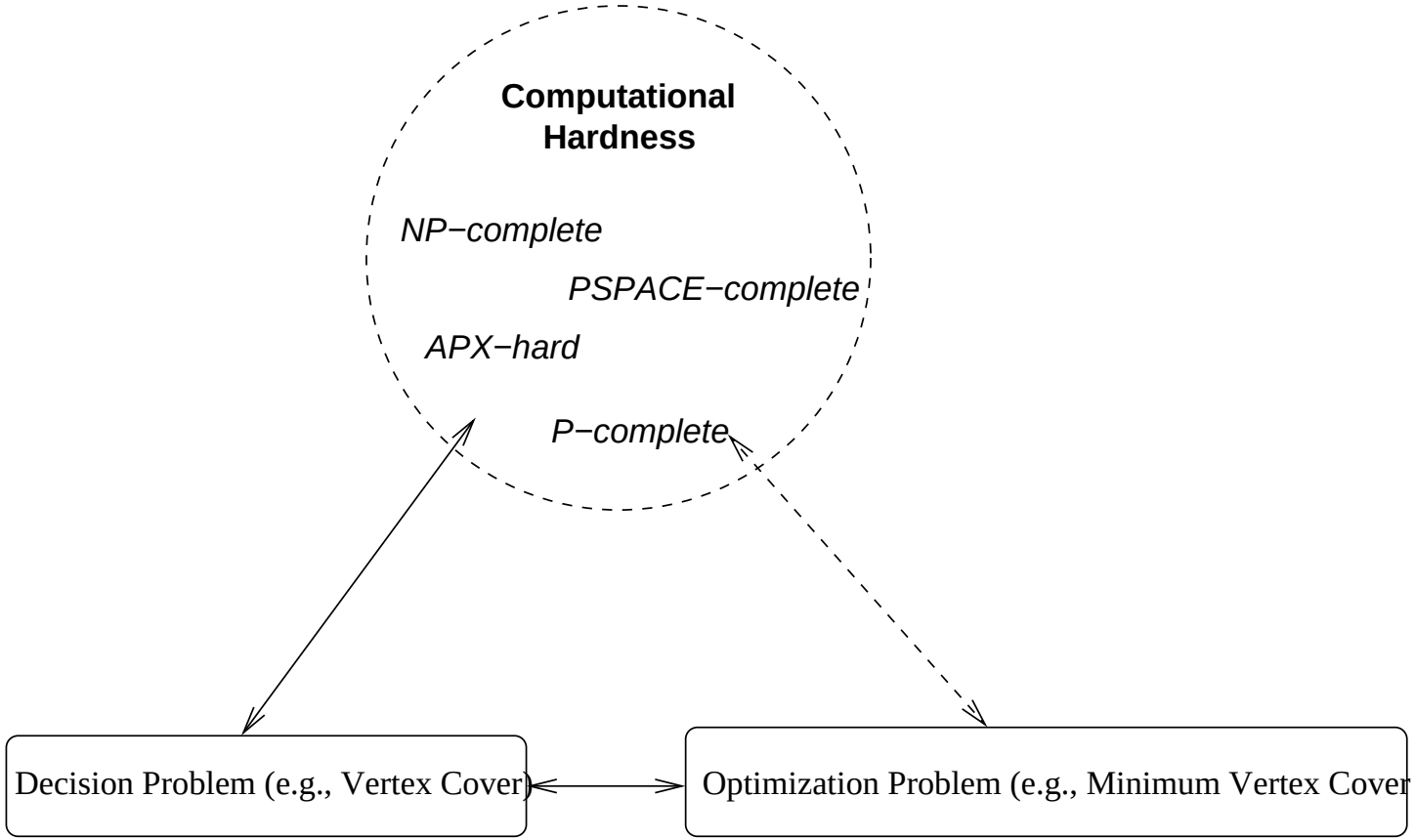
Efficient Algorithms

It appears that efficient algorithms cannot be found for all problems.

For many problems, such as boolean formula satisfiability (*SAT*), the best known algorithms to solve these problems run in time that is exponential in the size of the input. The big question is "Is this the best we can do?" Or, is it the case that we are just not smart enough?

Research that began in the early 1970's (and is ongoing) provides a partial answer to the above question, namely, that it is very likely that SAT and other related problems do not have efficient algorithms because they are *NP-Complete* (e.g.).

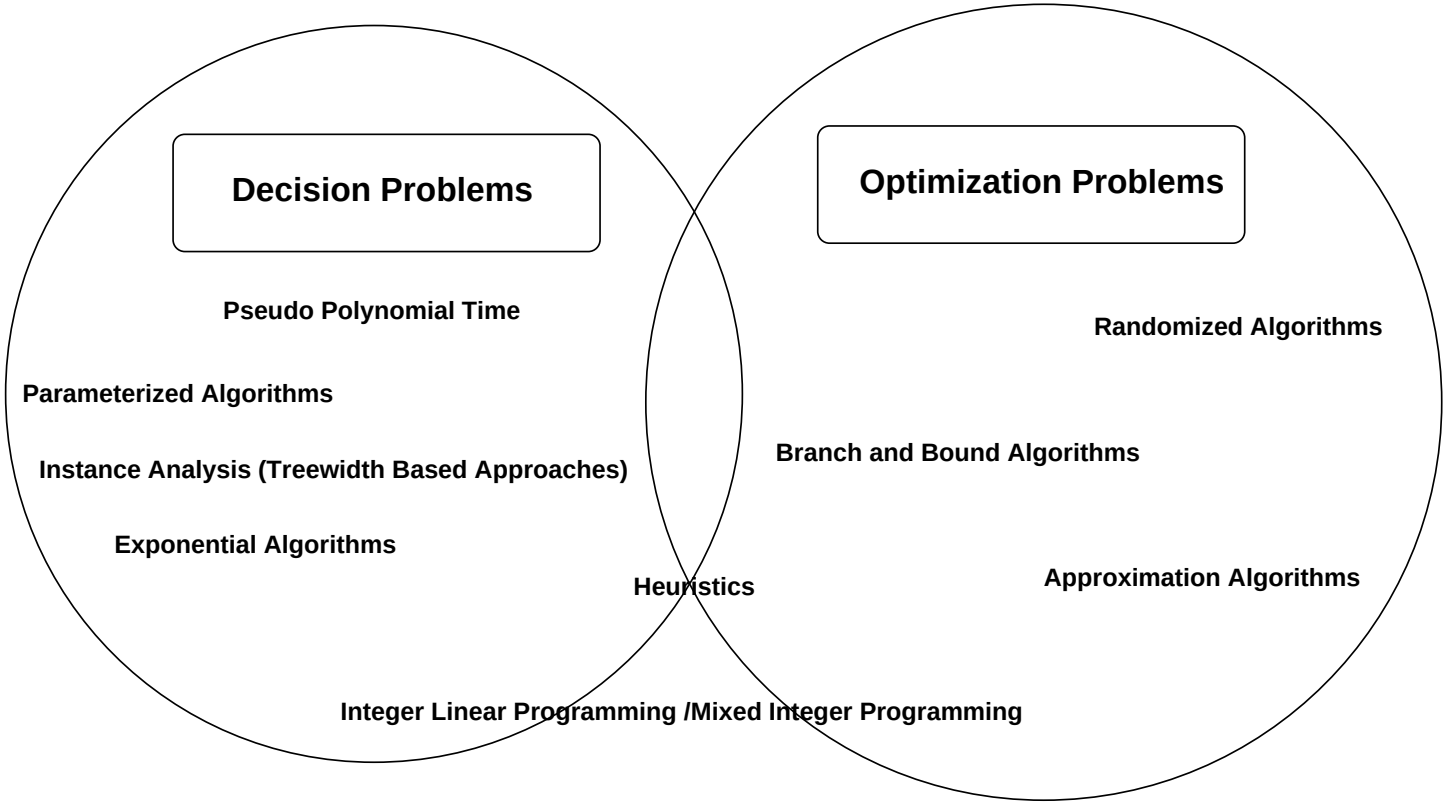
The Picture of Complexity



Most "interesting" problems have some aspect (version) that is computationally hard.

Tackling Complexity

The good news is that there are a number of techniques that can be used to handle "complexity."



Cont'd

Some approaches are better than others. Often, the hardest part is providing the correct definition of the problem.

We'll spend some time looking at specific problems that you may be interested in and exploring various approaches

This Course

This course examines recent advanced topics in algorithm design and analysis. We'll cover

- Quick Review of NP-Completeness (1970's – 1990s)
- Quick Review of Matching and Matching Algorithms
- Graph Algorithms and Analysis (Some of this is Pre 1970)
- Instance-based algorithms/Parameterized Algorithms (Mid 1980's – Today)
- Polynomial approximation schemes (1970s– Today)
- Approximation schemes (1970s – Today)
- Approximation Algorithms/Approximation Analysis (Mid 1990's — Today)
- Non-approximability results (1990s)

Cont'd

My main goals for you from this class are the following

1. That you develop a solid understanding of the concepts associated with formal algorithm development for NP-hard/complete problems.

2. That you are able to apply these concepts to develop algorithms (and **prove** properties of your algorithms) for specific problems that you encounter in your research.

3. That you are able to implement these various algorithms.

I want to pay special attention to **your** problems (e.g, computational biology or real-time systems), and implementations of various algorithmic approaches.

Example: Graph Algorithms

Many of the problems that we will examine will have a graph-theoretic flavor. As an example, let us consider independent sets in graphs.

Problem - INDEPENDENT SET

Given a graph $G = (V, E)$, a set $S \subseteq V$ is independent if no two vertices in S are connected by an edge in G. That is, $\forall x, y \in S, (x, y) \notin E$.

Natural problems:

1. *Optimization Problem*
   - Input: graph $G = (V, E)$
   - Output: Independent set S in G of maximum cardinality

Independent Set

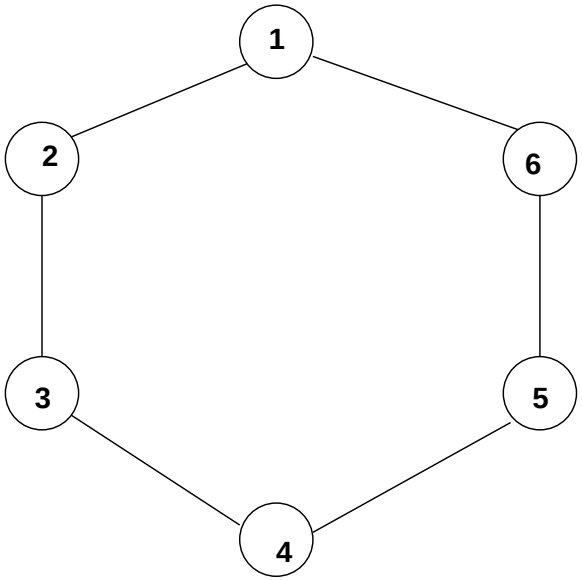
2. *Decision Version*

   - Input: graph $G = (V, E)$ and an integer k
   - Output: Does G have an Independent Set S with $|S| \geq k$?

As we will see later, the decision and the optimization versions are closely related.

Notice that the optimization version is probably the most natural of the two.

Example



The graph given in figure 1 has 6 vertices, and two maximal independent sets, namely, $\{1, 3, 5\}$ and $\{2, 4, 6\}$.

Independent Set, cont'd

One practical application of INDEPENDENT SET is that of class scheduling. We can define the vertices of the graph to be the students we wish to schedule.

We place an edge between two students if the two students' schedules cover all possible class times.

INDEPENDENT SET would then give a maximal set of students that could meet at a given time.

Questions

Questions to think about:

(Easy) Assume that we have an efficient algorithm that solves the optimization problem. Show that this allows us to give an efficient algorithm that solves the decision version.

(Harder) Assume that we have an efficient algorithm that solves the decision version. Give an efficient algorithm that solves the optimization problem.

Algorithms for INDEPENDENT SET

Solution #1 (Brute Force)

1. Generate all the subsets of V.
2. Check to see if each subset is independent or not.
3. Return the largest independent set.

Analysis

How fast does this run? (Worst Case, Best Case, Average Case?)

Analysis

- There are $2^{|V|}$ subsets of V.
- Checking to determine whether or not a given subset of V is independent takes $O(n^2)$ steps, where $n = |V|$.
- Therefore, our algorithm takes $O(n^2 2^n)$ steps to complete.

Solution #2

Let's denote the maximum # of vertices in an independent set by **maxset(G)**. Fix some vertex v^* in the graph. We can distinguish two different types of independent sets in the graph, namely, those that contain v^* and those that do not.

If an independent set S contains v^*, then what does the rest of S consist of?

We know that S does not contain any of the vertices in the neighborhood of v^*, denoted Nghd(v^*). Moreover, the remaining elements of S form an independent set. These elements are an independent set in $G - \{v^*\} - \text{Nghd}(v^*)$.

Similarly, if v^* is not in an independent set S, then S is an independent set in the graph $G - \{v^*\}$.

Recursive Algorithm

From our discussion about v^*,

$$\text{maxset}(G) = \max \left\{ \begin{array}{l} \text{maxset}(G - \{v^*\}), \\ \text{maxset}(G - \{v^*\} - \text{Nghd}(v^*)) + 1 \end{array} \right\}.$$

This observation gives the following recursive algorithm for maxset(G).

maxset(G)

1. If G contains no edges, return $n = |V|$.
2. Otherwise, pick a vertex v^* with at least one neighbor.
3. Return max{maxset($G - \{v^*\}$), maxset($G - \{v^*\} - \text{Nghd}(v^*)) + 1$}.

What's the running time of this algorithm?

Analysis

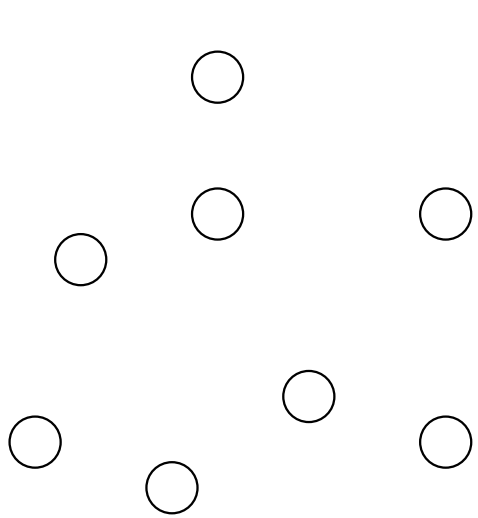
This algorithm has running time determined by the recurrence

$$T(n) = T(n-1) + T(n-2) + cn^2.$$

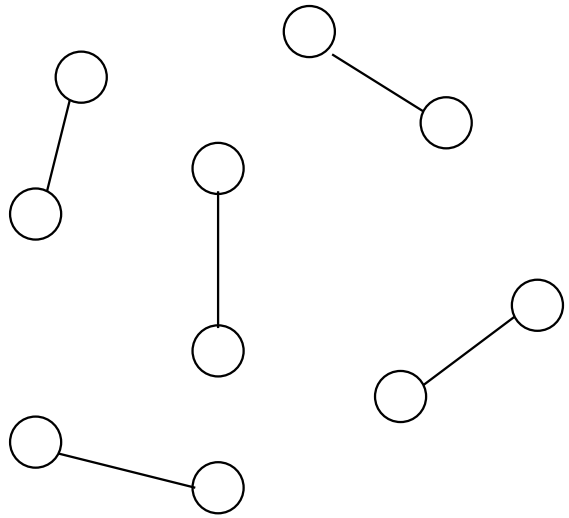
The solution to this recurrence yields a running time of $O(1.619^n)$.

Another Algorithm

We can do better if we make the neighborhoods of v^* larger. So, how can we force every neighborhood of v^* to contain at least 2 vertices? If a graph contains no vertex of degree 2, then it looks like:



Vertices of degree 0



Vertices of degree 1

Cont'd

Thus, a maximal independent set is just every vertex of degree 0 along with one of each pair of vertices of degree 1.

How fast is this algorithm?

Analysis

Using this fact, we can get an algorithm that has recurrence relation

$$T(n) \leq cn^2 + T(n-1) + T(n-3)$$

which yields a running time of $O(1.47^n)$.

Discussion, cont'd

Problems such as INDEPENDENT SET, clique, and SAT are known to have exponential time algorithms, but it is not known if these problems have algorithms that solve them efficiently (i.e., in polynomial time).

Efficient Computation

We generally consider a problem to be efficiently solvable if there is an algorithm that solves the problem in $O(n^k)$ steps, for some fixed k. We can classify all such problems into a complexity class. In this case, the complexity class is denoted P. (For technical reasons, P contains only decision problems.)

Next time

Problems such as clique, INDEPENDENT SET, SAT, etc. are not known to have efficient deterministic algorithms. However, all of these problems are known to have efficient *non-deterministic* algorithms.

The fact that these problems have efficient nondeterministic algorithms will help us understand their deterministic complexity.

....

We'll look at nondeterminism in the next set of lecture notes.