Undecidability

Thursday, April 4th
Outline For Today

1. Beyond Intractability: Undecidability
2. What to do with NP-complete Problems
3. Important Things to Remember In the Final Exam
4. Algorithmic Topics Beyond CS 341
Outline For Today

1. Beyond Intractability: Undecidability
2. What to do with NP-complete Problems
3. Important Things to Remember In the Final Exam
4. Algorithmic Topics Beyond CS 341
Are NP-complete Problems The Most Difficult Problems That Exist?

No!

And many more classes that is not in this picture. However, all of these problems are solvable.
There are also unsolvable problems.

We prove problem A is unsolvable by showing that:

- The existence of an algorithm that solves A leads to a contradiction (paradox).
HALTING Problem (1936, Church-Turing)

◆ Input: Computer Program A & an input D to A
  ▪ All input represented as binary bits
  ▪ Note D can be any input (including an algorithm)

◆ Output: YES: if A halts and returns on D
  NO: if A doesn’t halt (gets into an infinite loop)

Claim: Halting Problem is Undecidable
  i.e.: there cannot be an algorithm that solves HALTING

Existence of such an alg. leads to a contradiction/paradox!
A Fun Example of a Paradox
(Bertrand Russell, 1901)

◆ Self-referential statements: A trick to achieve paradoxes

“Suppose there is a village in which the barber shaves those and only those who don’t shave themselves. That is:

Case 1: if person A shaves himself →
the barber does not shave A.

Case 2: If A does not shave himself →
the barber shaves A.”

Q: Does the barber shave himself?
If yes → barber doesn’t shave himself
If no → barber shaves himself

Contradiction either way! Such a barber cannot exist!
Example 1:

```plaintext
procedure A(D):
    while (true);
    print ("still running");
```

Never halts on any D!

Example 2:

```plaintext
procedure A(D):
    return 5;
```

Always halts on every D!

But cannot have an alg that will decide if an algorithm A will halt or not on an arbitrary (A, D)!
Proof that HALTING is Undecidable

Suppose, for contradiction, \( \exists \) an alg. \( M \) that solves HALTING

Suppose w.l.o.g \( M \) returns YES if \( (A, D) \) halts and NO o.w.

\[
\text{procedure } M(A, D):
\]

\[
\text{if } A \text{ halts on } D: \text{ return YES;}
\]

\[
\text{else return NO}
\]

Then we can design the following algorithm OPP-M:

\[
\text{procedure } OPP-M(A, D):
\]

\[
\text{// if } A \text{ halts on } D, \text{ go into an infinite loop}
\]

\[
\text{if } M(A, D) == YES: \text{ while (true) } \{ \ldots \};
\]

\[
\text{else return 1; } \text{// else halt}
\]
Let’s Make OPP-M Self Referential

Note: Some algs take as an input another algs (or program) 
Ex: compilers take as input source codes of other programs.

Design OPP-M-S, for self, that takes one input:

```plaintext
procedure OPP-M-S(A):
    // if A halts on A, go into an infinite loop
    if M(A, A) == YES: while (true) { ... };
    else return 1; // else halt
```
Consider OPP-M-S on Input OPP-M-S

procedure OPP-M-S(OPP-M-S):
    // if OPP-M-S halts on OPP-M-S, loop forever
    if M(OPP-M-S, OPP-M-S) == YES: while (true) {...};
    else return 1; // else halt

If OPP-M-S halts on OPP-M-S

⇒ OPP-M-S loops forever on OPP-M-S (contradiction)!

If OPP-M-S does not halt on OPP-M-S

⇒ OPP-M-S halts on OPP-M-S (contradiction)!

Q.E.D

Conclusion: \( \not\exists \) an algorithm M that solves HALTING!
Conclusion: HALTING is Unsolvable/Undeceidable

Note however that HALTING is a well-defined computational problem!
That is, we know every program either halts on a particular input or it does not!
Yet we cannot write an algorithm that solves HALTING (no matter how much time/memory/network/electricity we give to the algorithm).

Doesn’t mean we cannot solve it on some inputs: e.g., compilers report some infinite loop cases
Proving Other Problems are Undecidable by “Reductions”

Just like we spread intractability among NP-complete problems we can also spread undecidability through reductions.
HALT-ALL : Reducability among Undecidable Problems

◆ Input: A Computer Program A
◆ Output: YES: if A halts on all possible inputs

NO: if A doesn’t halt on all inputs, i.e., ∃ an input D on which A doesn’t halt.

Claim: HALT-ALL is Undecidable
Reducing HALTING to HALT-ALL

Goal: Take an instance of Halting problem \((A, D)\) and convert to an equivalent HALTING-ALL problem \(A`\) s.t.

\[
A \text{ halts of } D \iff A` \text{ halts on all instances}
\]

**procedure** \(A`(input \ x)\)

// ignore input \(x\)

\(A(D); \text{ return YES;}\)

Notice the input to \(A`\) is ignored.

\(\Leftarrow\) If \(A`\) halts on all instances then \(A\) halts on \(D\)

\(\Rightarrow\) If \(A\) halts on \(D\); then \(A`\) halts on all instances

Therefore: HALTING-ALL is Undecidable
Note about Reductions Among Undecidable Problems

Our reduction from HALTING-HALT-ALL took constant time.

But in general when proving that a problem C is undecidable through a reduction from a known undecidable problem C`, the reduction does NOT have to be poly-time.

It can actually take any (finite amount of) time.

But recall when proving NP-completeness through reductions, the reductions HAVE TO BE poly-time.
Logicomix

Book by Christos Papadimitriou (theoretical CS @Berkeley)

About how Russell and other logicians of his time (1920s & 30s), e.g., Hilbert, Whitehead, Godel, contributed to the birth of CS
CS 341 Diagram

Fundamental (& Fast) Algorithms to Tractable Problems

- MergeSort
- Strassen’s MM
- BFS/DFS
- Dijkstra’s SSSP
- Kosaraju’s SCC
- Kruskal’s MST
- Floyd Warshall APSP
- Topological Sort
- ...

Common Algorithm Design Paradigms

- Divide-and-Conquer
- Greedy
- Dynamic Programming

Mathematical Tools to Analyze Algorithms

- Big-oh notation
- Recursion Tree
- Master method
- Substitution method
- Exchange Arguments
- Greedy-stays-ahead Arguments

Intractable Problems

- P vs NP
- Poly-time Reductions
- Undecidability

Other (Last Lecture)

- Randomized/Online/Parallel Algorithms
The rest will be about what to do when you see an Intractable (NP-complete) problem.

But first a little bit of history...
Alan Turing: Founder of Computer Science
“On Computable Numbers, with an Application to the Entscheidungsproblem”:  

“We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions”
Turing’s Answer To What Computation Is

Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. The behavior of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound $B$ to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite.
Let us imagine the operations performed by the computer to be split up into "simple operations" which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape.
Besides these changes of symbols, the simple operations must include changes to the observed squares. ... I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount.
Computer Science

Studies the powers of machines.

Fundamental Question CS asks:

What is “computable” by machines?

Turing (along with Church and Godel) was the person who made computation something we can mathematically study.
It’s interesting that as humans in our quest to understand what machines can do, we have been studying an abstract machine that in essence imitates a human being.
Central Dogma of Computer Science:

**Whatever is computable is computable by the Turing machine.**

There is no proof of this claim.
“All arguments which can be given are bound to be, fundamentally appeals to intuition, and for this reason rather unsatisfactory mathematically.

The arguments which I shall use are of three kinds.

(a) A direct appeal to intuition.

(b) A proof of the equivalence of two definitions [referring to Godel’s definition].

(c) Giving examples of large classes of numbers which are computable.”
When we say there is an algorithm computing shortest paths of a graph in $O(m\log(n))$ times we really mean:

There is a Turing Machine that computes the shortest paths of a graph in $O(m\log(n))$ operations.
Turing Machine Is The Most Powerful Machine

CS tries to understand the limits of TM.
We limit/extend TM and try to understand what can be computed by it.

- Limit the # times it’s head is allowed to move left/right and it changes states to a polynomial. => poly-time algs
- What if the machine had access to a random source => randomized algorithms.
- What if there were multiple heads on the tape => parallel algorithms
- Limit the length of its tape. => space-efficient algs
- What if the head was only allowed to move right => streaming algorithms
- ...

...
Outline For Today

1. Beyond Intractability: Undecidability
2. What to do with NP-complete Problems
3. Important Things to Remember In the Final Exam
4. Algorithmic Topics Beyond CS 341
Your Problem is NP-complete. Now What?

- Option 1: Focus to special-case inputs.
  - Ex: Independent Set is NP-complete.
  - Focusing on line graphs, had a $O(n)$ DP alg.

- Option 2: Find an approximate answer.

- Option 3: Be exponential time but better than brute-force search.
  - 0-1 Knapscak $O(nW)$ runtime DP algorihm.

- Option 4: Heuristics: fast algorithms that are not always correct (or even approximate)

- Option 5: Mix some of these options
Options 1&3: Restrict Input & Be exponential but better than brute force search (0/1 Knapsack)

- **Input:** n items
  - values for items $v_1, \ldots, v_n \geq 0$
  - sizes for items $w_1, \ldots, w_n \geq 0$
  - knapsack capacity $W \geq 0$

- **Output:** subset $S \subseteq \{1, 2, \ldots, n\}$ items s.t.

$$\max \sum_{i \in S} v_i \quad \text{ Recall Fact: 0/1 Knapsack is } \text{ NP-complete}$$

$$s.t. \sum_{i \in S} w_i \leq W$$
Restrict \( w_i \) and \( W \) to be Integers

◆ **Input:**

- n items
- values for items \( v_1, \ldots, v_n \geq 0 \)
- sizes for items \( w_1, \ldots, w_n \geq 0 \) & \( w_i \) are **INTEGERS**
- knapsack capacity \( W \geq 0 \) & \( W \) is an **INTEGER**

◆ **Output:** subset \( S \subseteq \{1, 2, \ldots, n\} \) items s.t.

\[
\begin{align*}
\text{max } & \sum_{i \in S} v_i \\
\text{s.t. } & \sum_{i \in S} w_i \leq W
\end{align*}
\]
0-1 Knapsack with integral \( w_i \) and W DP Alg

\[ K(i, c) \text{: opt. knapsack for the first } i \text{ items and cap } c. \]

\[
K(i, c) = \max \begin{cases} 
K(i-1, c) \\ 
K(i-1, c - w_i) + v_i 
\end{cases}
\]

\textbf{procedure} DP-Knapsack-1(n, W):
Base Cases: \( A[0][i] = 0 \)
for \( i = 1, 2, \ldots, n: \)
\quad for \( c = 1, \ldots, W: \)
\quad \quad \( A[i][c] = \max\{A[i-1][c], A[i-1][c-w_i]+v_i\} \)
return \( A[n][W] \)
Options 1& 3: “Good” Exponential Runtime on restricted input

Runtime: $O(nW)$

Brute-Force Search: $\Omega(2^n)$

Observation: This is polynomial in $n$ and $W$.

Q: Is Knapsack then tractable?!

A: No! B/c we’re still exponential in input size.

Input size $n + \log(W)$ and $W$ is exponential in $\log(W)$. 

Outline For Today

1. Beyond Intractability: Undecidability
2. What to do with NP-complete Problems
3. Important Things to Remember In the Final Exam
4. Algorithmic Topics Beyond CS 341
Final Information

- **Date:** April 15th
- **Practice final will be posted on Learn** (no solutions most likely, ask us)
- **Same format as the practice final**
- **Coverage:** Everything including Undecidability
◆ Get the indices in your DP algs right!

procedure DP:

\[
\text{for } i = 1 \ldots n \\
\text{for } j = 1 \ldots n \\
\]

A[i][j+1] is not yet computed, so cannot access it

Usually this is not a problem at all, but make sure you are accessing already solved problem.
Know the runtimes of every fundamental algorithms we learned
Things To Remember In The Final and After (3)

◆ Remember the math tools:
  ◆ Big-Oh notation

◆ Recurrences:
  ◆ Master Method
  ◆ Substitution Method
  ◆ Recursion Tree Method

◆ Basic math sequences we learned
  ◆ …
2 Things to Remember in NP-completeness Proofs

Suppose you’ll prove problem X is NP-Complete

1. First you have to show that X is in NP:

2. Get the direction of your reduction correct:
   
   You reduce a known NP-complete problem Y to X

   Not X to Y.
If you believe your algorithm is wrong, practice on skills to construct very very simple counter examples
Outline For Today

1. Beyond Intractability: Undecidability
2. What to do with NP-complete Problems
3. Important Things to Remember In the Final Exam
4. Algorithmic Topics Beyond CS 341
Randomization: Max Cut Problem

- **Input:** Undirected $G(V, E)$
- **Output:** cut $(X, Y)$ with max # of crossing edges

*Fact: Max-Cut is Intractable (NP-complete)*

$\Rightarrow$ can’t hope to solve it exactly within reasonable (poly-time) amount of time

*Another Fact: Min-Cut is poly-time*

*Takeaway: Don’t be deceived by how similar two problems look like! One may be tractable and the other intractable!*
procedure Approximate-MaxCut(G(V,E)):
  let L, R be 2 empty sets
  for each vertex v:
    toss a fair coin
    if heads: put v into L
    if tails: put v into R

This is a $\frac{1}{2}$ approximation to OPT.

Take Away 1: Solving max-cut exactly is impossible but we have a very simple linear time approximate randomized algorithm!

Take Away 2: Randomized Algorithms can be very elegant.
Optimization Problem of following structure:

maximize $x_1 + x_2$

subject to

$x_1 + 2x_2 \leq 1$

$2x_1 + x_2 \leq 1$

$x_1 \geq 0$

$x_2 \geq 0$
Geometric Interpretation

Constraint 2: $2x_1 + x_2 \leq 1$

**Opt Solution**

Constraint 1: $x_1 + 2x_2 \leq 1$

Feasible Solution Set
Linear Programming Applications

- Tons! Lots and lots of problems can be solved or approximated with LP!
  - Vertex Cover
  - Set Cover
  - Load Balancing
  - Lots of problems in manufacturing/operations research/finance, etc..

- Covered in CS 466

**Invented By George Dantzig.**
Algorithms/Theory Courses Beyond CS 341

- CS 466: Approximation & Randomized Algorithms
- CS 365/CS 664: Complexity Theory
- CS 467: Intro to Quantum Information Processing
- CS 482: Comp. Techniques in Biological Sequence Analysis

For students who like theory: watch out for the 600 and 800 level courses thought by theory faculty:
Ian Munro, Eric Blais, Jeffrey Shallit, ...
Fundamental (& Fast) Algorithms to Tractable Problems

- MergeSort
- Strassen’s MM
- BFS/DFS
- Dijkstra’s SSSP
- Kosaraju’s SCC
- Kruskal’s MST
- Floyd Warshall APSP
- Topological Sort
- ...

Common Algorithm Design Paradigms

- Divide-and-Conquer
- Greedy
- Dynamic Programming

Mathematical Tools to Analyze Algorithms

- Big-oh notation
- Recursion Tree
- Master method
- Substitution method
- Exchange Arguments
- Greedy-stays-ahead Arguments

Intractable Problems

- P vs NP
- Poly-time Reductions
- Undecidability

Other (Last Lecture)

- Randomized/Online/Parallel Algorithms
Before/After CS 341

1. Fundamental Algorithms
2. Fundamental Algorithm Design Paradigms
3. Tractability/Intractability

Math Techniques for Algorithm Analysis
Acknowledgements

◆ Trevor Brown, Doug Stinson
◆ TAs: Vedat Alev, Kaleb Alway, Aseem Baranwal, Stavros Birmilis, Zhengkun Chen, Nathaniel Harms, Tiasa Mondol, Azin Nazari, Akshay Ramachandran, Hong Zhou
Thank you!

Good luck in the final!