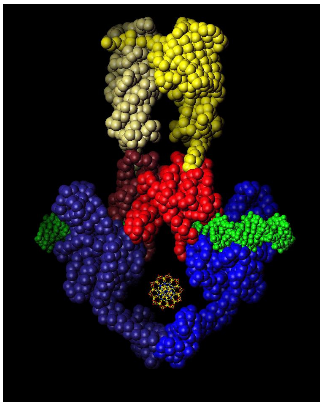
Class 24: P=NP?

Remaining Exam 2 comments now posted.

PS6 (the last one) is due Thursday, April 24.

David Evans
http://www.cs.virginia.edu/evans

cs302: Theory of Computation
University of Virginia Computer Science



Protein model, Berger Lab UC Berkeley

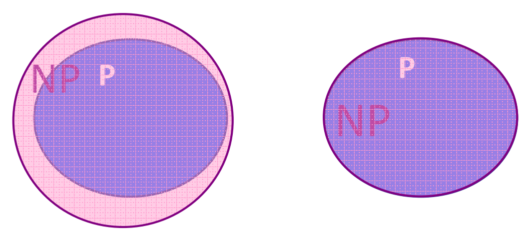
Final Exam

- Scheduled by registrar:
  - **Saturday**, May 3, **9am**-noon (exam is scheduled for 3 hours, but will be designed to take ≤ 1.5 hours)
- No notes or books allowed
  - My sense from grading Exam 2 is people used their notes as a crutch, not helpfully
  - Enables “easier” questions and more partial credit
- In class next Tuesday, I will hand out a “preview” of some of the exam questions and possibly discuss them

Final Exam Topics

- **Everything** covered through this Thursday:
  - Exams 1 and 2 and comments
  - Problem Sets 1-6 and comments
  - Lectures 1-25
  - Sipser, Chapters 0-5, 7
  - Additional Readings: Aaronson (spring break), one of the NP-completeness papers
- Roughly ⅓ Exam 1 material, ⅓ Exam 2 material, ⅓ since Exam 2 (but many individual questions will combine material from multiple parts)

P = NP ?



Option 1: $\mathbf{P} \subset \mathbf{NP}$

Option 2: $\mathbf{P} = \mathbf{NP}$

Theological Question

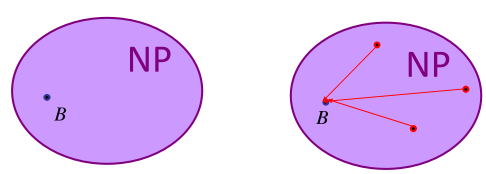
If God exists (and is omnipotent), can she compute anything regular people cannot compute?

Yes: $\mathrm{P} \subset \mathrm{NP}$
Being able to always guess right when given a decision makes you more powerful than having to try both.

No: $\mathrm{P} = \mathrm{NP}$
Being able to always guess right when given a decision does not make you more powerful than having to try both.

NP-Complete

A language B is in NP-complete if:

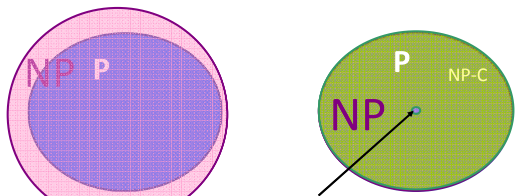


1. $B \in \mathbf{NP}$

2. There is a polynomial-time reduction from every problem $A \in \mathbf{NP}$ to B.

Is NP-Complete a Ring or a Circle?

NP-Complete

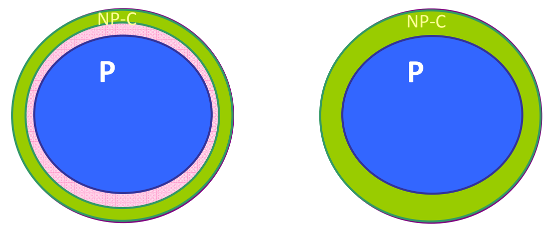


Tautology problems
A = { }; A = Σ*

Option 1: $\mathbf{P} \subset \mathbf{NP}$

Option 2: $\mathbf{P} = \mathbf{NP}$
= **NP**-Complete ∪ Tautology

NP-Complete



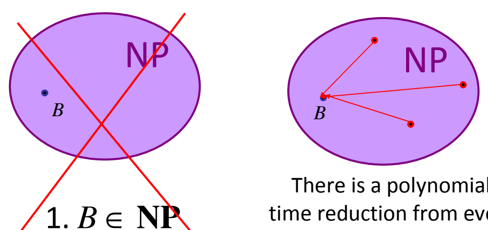
Option 1a: $\mathbf{P} \subset \mathbf{NP}$,
$\mathbf{NP\text{-}C} \cup \mathbf{P} \subset \mathbf{NP}$

Option 1b: $\mathbf{P} \subset \mathbf{NP}$,
$\mathbf{NP\text{-}C} \cup \mathbf{P} = \mathbf{NP}$

Either is possible

NP-~~Complete~~ Hard

A language B is in NP-complete if:



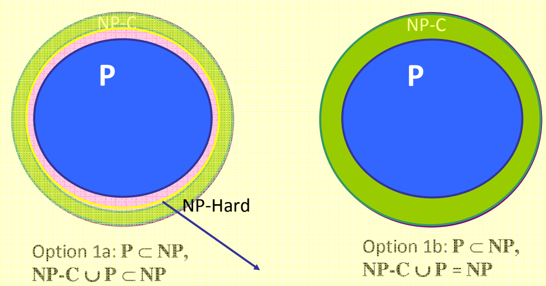
~~1. $B \in \mathbf{NP}$~~

Not *necessary* for NP-Hard

There is a polynomial-time reduction from every problem $A \in \mathbf{NP}$ to B.

What does NP-Hard look like?

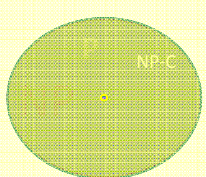
NP-Hard (if $\mathbf{P} \subset \mathbf{NP}$)



Option 1a: $\mathbf{P} \subset \mathbf{NP}$,
$\mathbf{NP\text{-}C} \cup \mathbf{P} \subset \mathbf{NP}$

Option 1b: $\mathbf{P} \subset \mathbf{NP}$,
$\mathbf{NP\text{-}C} \cup \mathbf{P} = \mathbf{NP}$

NP-Hard (if $\mathbf{P} = \mathbf{NP}$)



Option 2: $\mathbf{P} = \mathbf{NP}$
≈ **NP**-Complete

NP-Hard = All Problems - {A = { }; A = Σ*}

NP-Hardness Recap

- If P = NP:
  - To show a problem is NP-Hard: show for some input it outputs “true”, and for some input it outputs “false”
- If P ⊂ NP:
  - To show a problem is NP-Hard: show that there is a polynomial-time reduction from some known NP-Complete problem to it
  - Showing a problem is NP-Hard means there is no polynomial time solution for it

Games and NP-Hardness

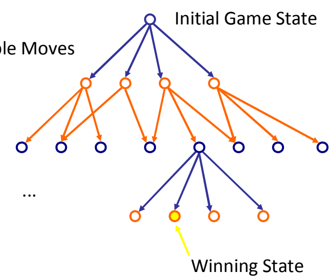
Papers from Last Class

- (Generalized) Cracker Barrel Puzzle is NP-Complete
- (Generalized) March Madness is NP-Hard
  - Is it NP-Complete also?
- (Generalized) Minesweeper Consistency is NP-Complete
- … ?

Are these special cases, or is there something about “interesting” games that makes them NP-Hard?

What makes a “game” a game?

All “Interesting” Games?



What is a “winning path”?

Recall: Class NP

A language is in NP if and only if it is decided by some nondeterministic polynomial time Turing Machine

A language is in NP if and only if it has a corresponding polynomial time verifier

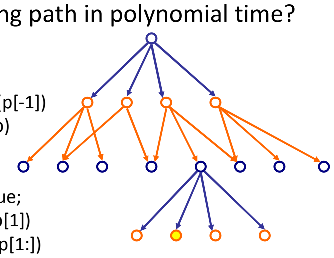
That is, there is a **certificate** that can prove a string is in the language which can be **checked** in polynomial time.

Game Certificate

- Given a path through a game, can you check if it is a valid winning path in polynomial time?

```
def verify(Path p):
  return isInitialState(p[0])
         && isWinningState(p[-1])
         && allMovesValid(p)

def allMovesValid(Path p):
  if (p.length <= 1) return true;
  return isValidMove(p[0], p[1])
         && allMovesValid(p[1:])
```



(One-Player) Games in NP

How could a game be outside NP?

- The maximum number of moves is polynomial in the size of the game (e.g., Hex, Sokoban)
- There is a polynomial-time procedure for checking a move (state, state pair) is valid ?
- There is a polynomial-time procedure for checking a position is a winner ?

Games in P

- The number of possible moves or the number of moves you need to lookahead to pick the right move, does not scale with the size of the game

There is a polynomial-time function from the game state to the correct move: don't need to consider deep paths to select the right move

NP-Complete One-Player Games

- In NP: polynomial-time certificate
- Polynomial-time reduction from 3SAT (or any other NPC problem) to the game

Essentially: no way to know if a move is correct without looking ahead all the way to the end.

All "fun" one-player games are NP-Complete:
Games inside P are too easy (once you solve them always win)
Games outside NP are too hard

But…we actually play finite versions of these games (in **TIME(1)**)

Reduction Proofs

Reducing Reduction Proofs

- Conjecture: A has some property Y.
- Proof by reduction **from** B to A:
  - Assume A has Y. Then, we know there is an M that decides A.
  - We already know B does not have property Y.
  - Show how to build S that solves B using M.
- Since we know B does not have Y, but having S would imply B has Y, S cannot exist. Therefore, M cannot exist, and A does not have Y.

Undecidability Proofs

- Conjecture: A has some property Y.
- Proof by reduction **from** B to A:
  - Assume A has Y. Then, we know an M exists.
  - We already know B does not have property Y.
  - Show how to build S that solves B using M.
- Since we know B does not have Y, but having S would imply B has Y, S cannot exist. Therefore, M cannot exist, and A does not have Y.

Undecidability:
Y = "can be decided by a TM"
B = a known undecidable problem (e.g., A_{TM}, $HALT_{TM}$, EQ_{TM}, …)
M = "a TM that decides A"

NP-Hardness Proofs

- Conjecture: A has some property Y.
- Proof by reduction **from** B to A:
  - Assume A has Y. Then, we know an M exists.
  - We already know B does not have property Y.
  - Show how to build S that solves B using M.
- Since we know B does not have Y, but having S would imply B has Y, S cannot exist. Therefore, M cannot exist, and A does not have Y.

NP-Hardness:
Y = "is NP-Hard"
B = a known NP-Hard problem (e.g., *3-SAT*, *SUBSET-SUM*, …)
M = "a TM that decides A in polynomial-time"

The Hard Part

- Conjecture: A has some property Y.
- Proof by reduction **from** B to A:
  - Assume A has Y. Then, we know an M exists.
  - We already know B does not have property Y.
  - Show how to build S that solves B using M.
- Since we know B does not have Y, but having S would imply B has Y, S cannot exist. Therefore, M cannot exist, and A does not have Y.

Example

- Suppose we know A_{TM} is undecidable, but do not yet know if EQ_{TM} is.

EQ_{TM} = { <A, B> | A and B are TMs where $L(A) = L(B)$ }
A_{TM} = { <M, w> | M is TM, w is string, w in $L(M)$ }

Conjecture: EQ_{TM} is undecidable.

What do we need to do to prove conjecture?

Reduce from A_{TM} to EQ_{TM}: show that a solver for EQ_{TM} could be used to solve A_{TM}.

Pitfall #1: Make sure you do reduction in right direction.
Showing how to solve *B* using M_A, shows *A* is as hard as *B*.

Building Solvers

B = A_{TM} = { <M, w> | M is TM, w is string, w in $L(M)$ }
A = EQ_{TM} = { <M_1, M_2> | M_1 and M_2 are TMs where $L(M_1) = L(M_2)$ }

Conjecture: EQ_{TM} is undecidable.

Reduce from EQ_{TM} to A_{TM}: show that M_{EQ}, a solver for EQ_{TM} can be used to solve A_{TM}.

M_B(<M, w>): machine that decides A_{TM}
Simulate M_{EQ} on <M_1, M_2>:
M_1 = a TM that simulates M running on w
M_2 = a TM that always accepts
If it accepts, accept; if it rejects, reject.

Pitfall #2: Get the inputs to the solver to match correctly.
To solve *B* using M_A, must transform inputs to *B* into inputs to *A*.

Legal Transformations

- Undecidability proofs: your transformation can do anything a TM can do, but must be guaranteed to terminate
  - E.g., cannot include, "simulate M and if it halts, accept"
- NP-Hardness proofs: your transformation must finish in polynomial time
  - E.g., cannot include, "do an exponential search to find the answer, and output that"

Example: *KNAPSACK* Problems

- You have a collection of items, each has a value and weight
- How to optimally fill a knapsack with as many items as you can carry

Scheduling: weight = time,
one deadline for all tasks
Budget allocation: weight = cost

General *KNAPSACK* Problem

- **Input:** a set of n items
  $\{<\text{name}_0, \text{value}_0, \text{weight}_0>, \ldots,$
  $<\text{name}_{n-1}, \text{value}_{n-1}, \text{weight}_{n-1}>\}$
  and *maxweight*
- **Output:** a subset of the input items such that the sum of the weights of all items in the output set is ≤ *maxweight* and there is no subset with weight sum ≤ *maxweight* with a greater value sum

Note: it is not a decision problem. Can we make it one?


```
def knapsack (items, maxweight):
  best = {}
  bestvalue = 0
  for s in allPossibleSubsets (items):
     value = 0
     weight = 0
     for item in s:
        value += item.value
        weight += item.weight
     if weight <= maxweight:
        if value > bestvalue:
           best = s
           bestvalue = value
  return best
```

2^n subsets

$\Theta(n)$ for each one

Running time $\in \Theta(n2^n)$

Does this prove it is not in **P**?

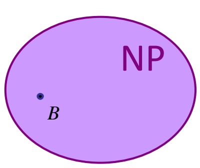
No!

To prove it is not in P, we would need to show the **best** possible algorithm that solves it is not polynomial time.

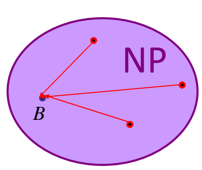
Is KNAPSACK NP-Complete?

NP-Complete

A language B is in NP-complete if:



1. $B \in$ **NP**



2. There is a polynomial-time reduction from every problem $A \in$ **NP** to B.

KNAPSACK in NP

- Certificate: subset of items
- Test in P: add up the weights of those items, check it is less than *maxweight*

For the non-decision problem: ask for certificates for all values 1, 2, …, *maxweight*.

KNAPSACK in NP-Complete

- Reduction from *SUBSET-SUM* to *KNAPSACK*:

  $SUBSET\text{-}SUM = \{ \langle S, t \rangle \mid S = \{x_1, \ldots, x_k\}$ and for some $\{y_1, \ldots, y_l\} \subseteq S, \Sigma y_i = t \}$

  Transform input to match KNAPSACK:

  Input: a set of n items

  $\{\langle name_0, value_0, weight_0 \rangle, \ldots, \langle name_{n-1}, value_{n-1}, weight_{n-1} \rangle\}$

  and *maxweight*

Input Transformation

$SUBSET\text{-}SUM$ $(\langle S, t \rangle)$: $S = \{x_1, \ldots, x_k\}$

do something using

$KNAPSACK$ ($\langle\{\langle$"x1", $x_1, x_1\rangle, \ldots,$

$\langle$"xk", $x_k, x_k\rangle\}, t\rangle$)

KNAPSACK **Input:** a set of n items

$\langle\{\langle name_0, value_0, weight_0 \rangle, \ldots, \langle name_{n-1}, value_{n-1}, weight_{n-1} \rangle\},$

$maxweight\rangle$

Output Transformation

$SUBSET\text{-}SUM$ $(\langle S, t \rangle)$: $S = \{x_1, \ldots, x_k\}$

accept iff

t = Σ ($KNAPSACK$ ($\langle\{\langle$"x1", $x_1, x_1\rangle, \ldots,$

$\langle$"xk", $x_k, x_k\rangle\}, t\rangle$)))

KNAPSACK Output: a subset of the input items such that the sum of the weights of all items in the output set is ≤ *maxweight* and there is no subset with weight sum ≤ *maxweight* with a greater value sum

"Solving" NP-Hard Problems

- What do we do when solving an important problem requires solving an NP-Complete problem?

  a. Give up.
  b. Hope P = NP.
  c. Solve a different problem.
  d. Settle for an "incorrect" answer.

Approximation Algorithms

Sometimes it is better to produce an incorrect answer quickly, than wait (longer than the lifetime of the universe) for a correct answer.

A good approximation algorithm:
1. Runs in Polynomial Time
2. Produces answer within some known bound of best answer

Greedy Algorithms

- Make locally optimal decisions
- For NP-Hard problems: cannot guarantee you find the best answer this way

Greedy Knapsack Algorithm

```
def knapsack_greedy (items, maxweight):
  result = []
  weight = 0
  while True:
    # try to add the best item
    weightleft = maxweight - weight
    bestitem = None
    for item in items:
      if item.weight <= weightleft \
        and (bestitem == None \
             or item.value > bestitem.value):
        bestitem = item
    if bestitem == None: break
    else:
      result.append (bestitem)
      weight += bestitem.weight
  return result
```

Running Time
$\in \Theta(n^2)$

Is Greedy Algorithm Correct?

No.

Proof by counterexample:

Consider input

items = {<“gold”, 100, 1 >,
<“platinum”, 110, 3>
<“silver”, 80, 2 >}

maxweight = 3

Greedy algorithm picks {<“platinum”>}
value = 110, but {<“gold”>, “silver”>}
has weight <= 3 and value = 180

The Moral

Life is (NP-) Hard, but probably not (NP-)Complete…

Thursday: Karsten Nohl will talk about interesting theory problems in breaking cryptosystems