

# Announcements

- Exam 3 Solutions Online
- Final Exam March 19<sup>th</sup>

# Last Time

- NP-Hard/Complete Problems are as hard as any other problems in NP.
- It is believed that this means that there are no polynomial time algorithms for such problems.

# Dealing With NP-Completeness (Ch 9)

- Backtracking/Branch and Bound
- Heuristic Search
- Approximation Algorithms

# Identifying NP-Complete Problems

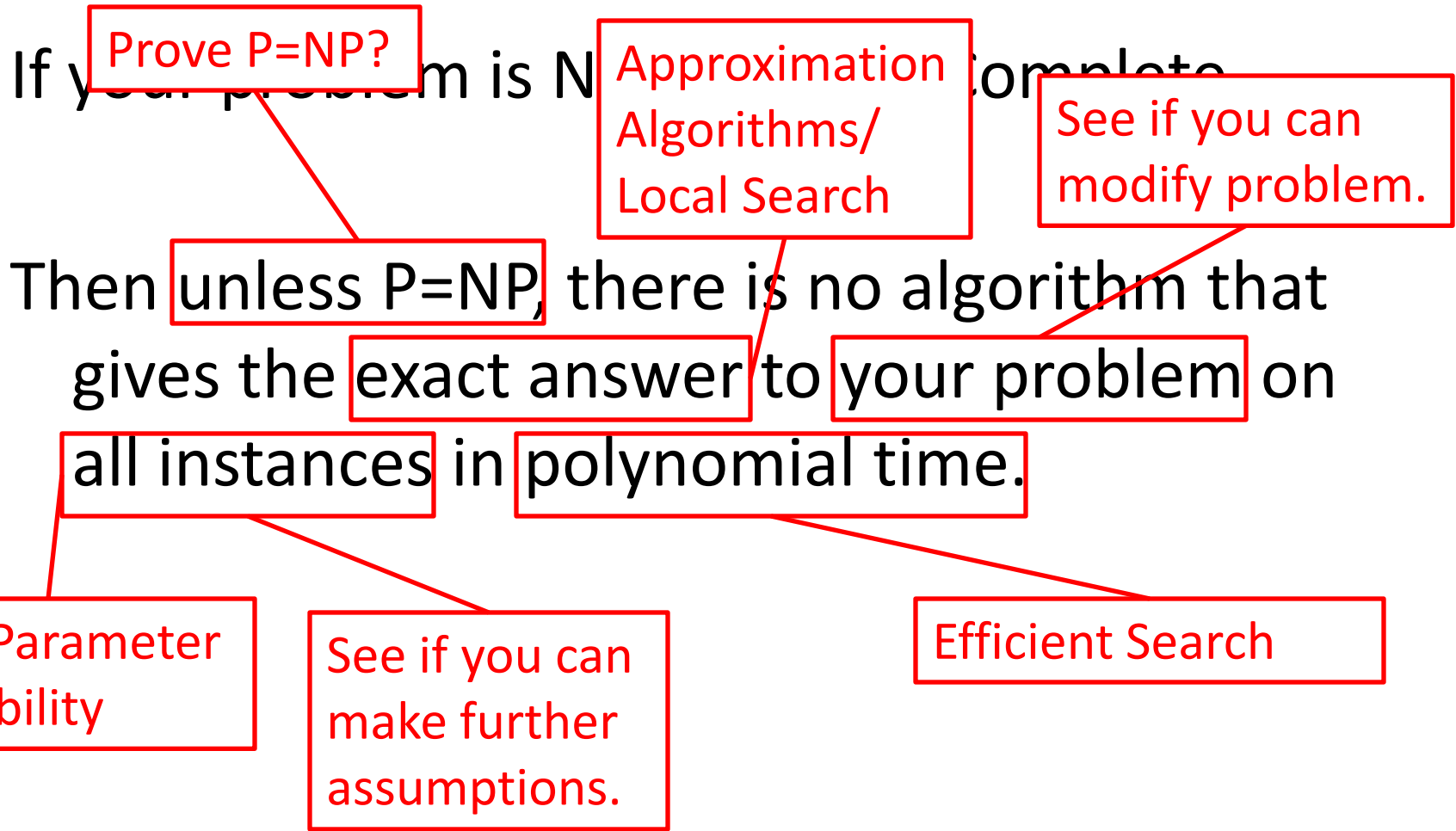
When given a problem to solve, it is important to determine if it is NP-Complete.

If it is, then you have very good evidence that you won't find a polynomial time solution. So you have an excuse for not having a better algorithm.

Unfortunately, this doesn't solve your original problem. Even if it's NP-Complete you still need to solve it anyway.

# Bad News

What are the loopholes here?



# Sudoku

Consider the logic puzzle Sudoku (or any similar logic puzzle).

Fill a 9x9 grid of numbers with 1-9 so that:

- Each row has all numbers
- Each column has all numbers
- Each of the main 3x3 sub squares has all numbers
- Some entries are pre-filled

# NP-Hard

Suitable generalizations of Sudoku are NP-Hard.

- So in general, you cannot do much better than brute force search.
- True brute force search would consider  $9^{81} \approx 2 \cdot 10^{77}$  possibilities.
- In practice, people can solve them while waiting for the dentist.
  - How?

# Deductions

One way to progress is so make deductions.

- Use the rules to show that some square can only be filled out in one way.
- Use that information to help fill out more squares.
- Hopefully, you can keep going until the entire problem is solved.



# Example

Consider 3-SAT:  $(x) \wedge (\bar{x} \vee y) \wedge (\bar{x} \vee \bar{y} \vee z)$ .

First clause implies  $x = \text{True}$ .

Plugging in and simplifying gives:  $(y) \wedge (\bar{y} \vee z)$ .

First clause implies  $y = \text{True}$ .

Plugging in and simplifying gives:  $(z)$ .

So we must have  $x = y = z = \text{True}$ , which is a solution.

# Getting Stuck

Deductions are very useful when you can make them, but for hard problems, you will often get stuck quickly and be unable to make more deductions.

How do you get out?

Option 1: Stronger deductive rules.

# Sudoku Inference Rules

More complicated deduction rules allow you to go further without getting stuck. Common Sudoku rules include:

- 1) Find a square that only one number can fill.
- 2) Find a region with only one place for a given number.
- 3) Find a pair of squares in the same row that must contain two numbers (which then cannot appear elsewhere in that row).
- 4) Find a rectangle whose corners must contain 2 copies of a number. That number cannot appear elsewhere in those rows/columns.
- 5) Find 3 rows & 3 columns whose intersections must contain 3 copies of a number. That number cannot appear elsewhere in those rows and columns.

# Still Stuck?

What if your complicated set of inference rules is still not enough?

There is a general strategy that can always be made to work.

Guess and check.

# Guess and Check

- Make a guess for some entry.
- Try to solve the resulting puzzle (perhaps doing more guessing).
- If you find a solution, great!
- If not, you have deduced that your original guess was wrong.

# Example

$$(x \vee y) \wedge (y \vee z) \wedge (z \vee x) \wedge (\bar{x} \vee \bar{y}) \wedge (\bar{y} \vee \bar{z}) \wedge (\bar{z} \vee \bar{x})$$

Guess  $x = \text{True}$ :  $(y \vee z) \wedge (\bar{y}) \wedge (\bar{y} \vee \bar{z}) \wedge (\bar{z})$

2<sup>nd</sup> clause:  $y = \text{False}$

4<sup>th</sup> clause:  $z = \text{False}$

Contradicts 1<sup>st</sup> clause.

So must have  $x = \text{False}$ :  $(y) \wedge (y \vee z) \wedge (z) \wedge (\bar{y} \vee \bar{z})$

1<sup>st</sup> clause:  $y = \text{True}$

3<sup>rd</sup> clause:  $z = \text{True}$

Contradicts 4<sup>th</sup> clause.

**No Solutions!**

# Backtracking

You can combine guess and check nicely with deductions. In fact, a deduction can be thought of as just guessing the wrong way to fill things in and then concluding that it doesn't work.

This brings us to the general algorithm of Backtracking. This takes some search problem  $P$  with some space  $S$  that needs to be searched.

# Backtracking

Backtracking( $P, S$ )

If you can deduce unsolveable

Return 'no solutions'

Split  $S$  into parts  $S_1, S_2, \dots$

For each  $i$ ,

Run Backtracking( $P, S_i$ )

Return any solutions found



# Splitting

How do you split  $S$  into parts?

- Pick variable  $x_i$  and set  $x_i = \text{True}$ , or  $x_i = \text{False}$
- Try all possible numbers in a square in Sudoku
- Try all possible edges in Hamiltonian Cycle

Which variable do we guess?

- Often helps to pick a variable that shows up a lot. Then guessing its value will make later deductions easier.

# Runtime

These problems are still NP-Hard. Worst case, backtracking will still take exponential time. But it is usually much better than brute force.

SAT Solvers can use these ideas to solve problems with hundreds of variables, many many more than would be practical by brute force.

# Optimization Version

Backtracking works well for decision/search problems (where a potential solution works or doesn't work), but not so well for optimization problems (where many solutions work, but you need to find the best one).

If most solutions work, how do you weed out bad paths?

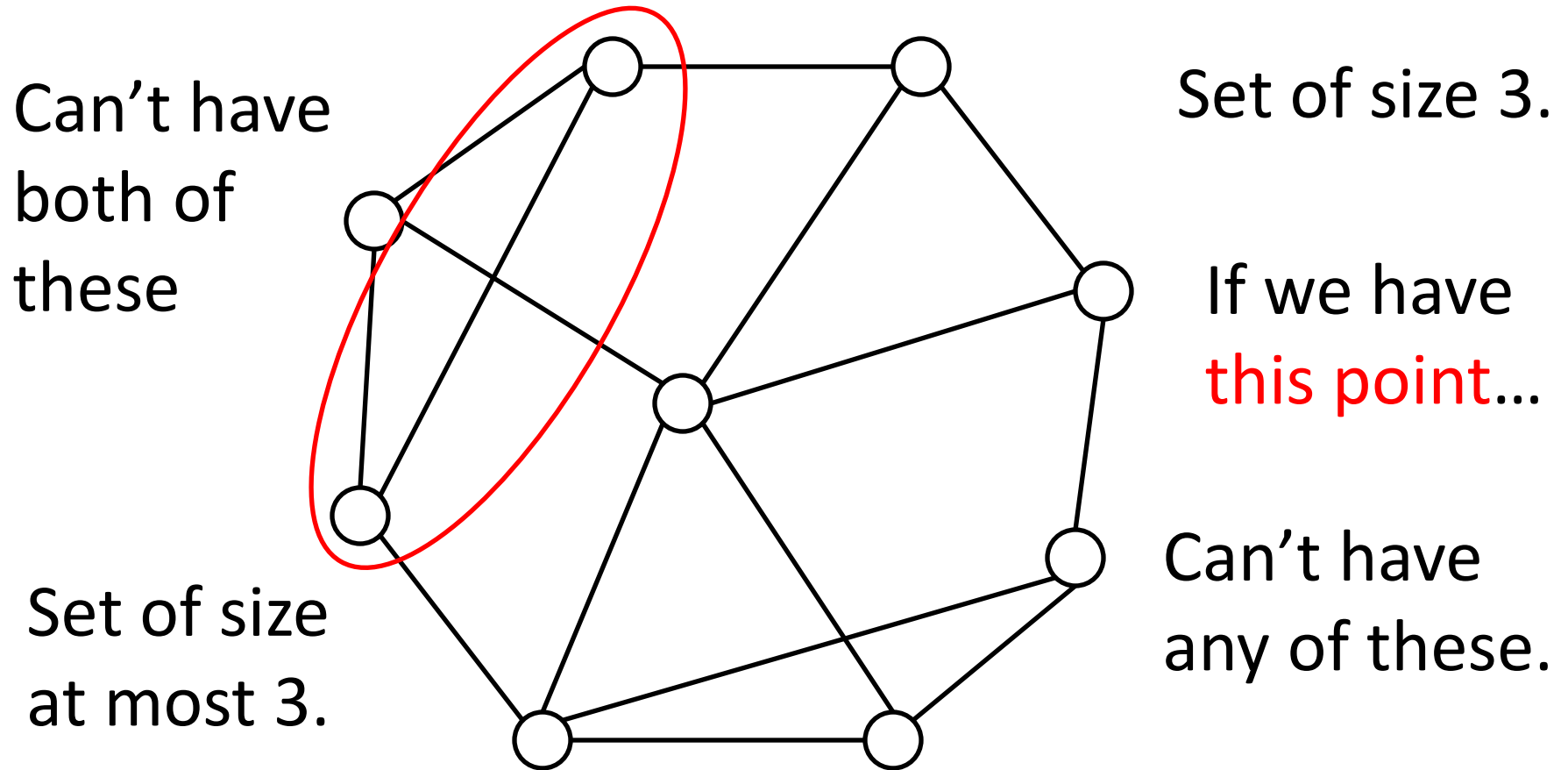
# Branch & Bound

To get rid of bad paths do two things:

- 1) Keep track of the best solution you have found so far.
- 2) Try to prove upper bounds on your subproblems.

If an upper bound is smaller than your best solution so far, it cannot contain the optimum.

# Example: Maximum Independent Set



# Branch and Bound

```
BranchAndBound(Best, S)
  If UpperBound(S) ≤ Best
    Return 'no improvement'
  If S a full solution
    Return value of S
  Split S into S1, S2, ...
  For each Si
    New ← BranchAndBound(Best, Si)
    Best = Max(New, Best)
  Return Best
```

# Local Search

Many optimization problems have a structure where solutions “nearby” a good solution will likely also be good.

This leads to a natural algorithmic idea:

- Find an OK solution
- Search nearby for better solutions
- Repeat

# Local Search

LocalSearch( $f$ )

\ \ Try to maximize  $f(x)$

$x \leftarrow$  Random initial point

Try all  $y$  close to  $x$

If  $f(y) > f(x)$  for some  $y$

$x \leftarrow y$

Repeat

Else Return  $x$