Announcements

- No HW this week
- 3-4pm discussion section today over zoom
- Final exam next week
  - Wednesday at 11:30
  - In this classroom
  - Randomized seating
  - Comprehensive (with slight emphasis on new material in Chapters 8 and 9)
  - 6 Questions in 3 hours
  - 12 one-sided pages of notes

Last Time

- NP Problems and NP completeness

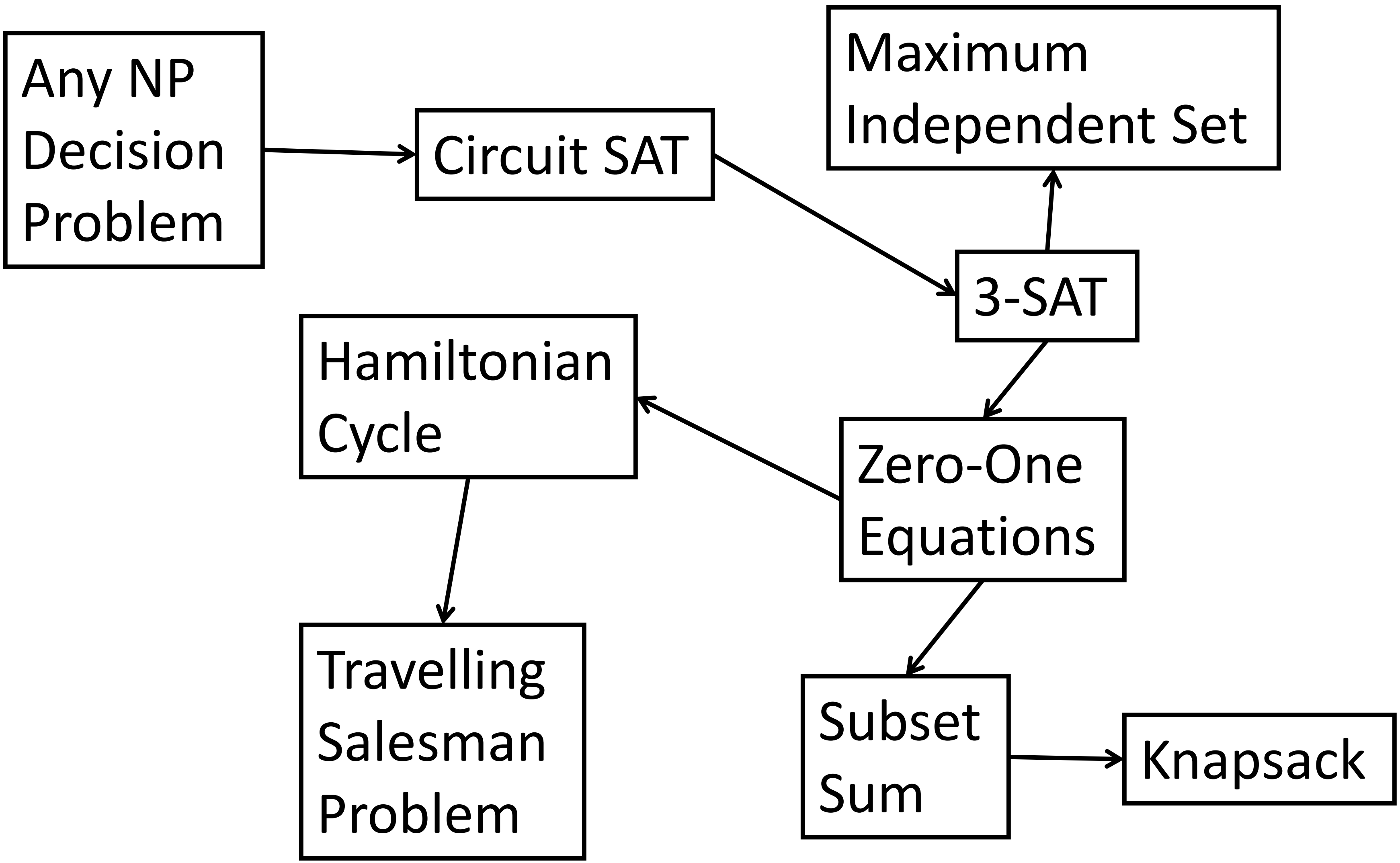
NP

Such problems are said to be in <u>Nondeterministic Polynomial</u> time (NP).

<u>NP-Decision</u> problems ask if there is some object that satisfies a polynomial time-checkable property.

<u>NP-Optimization</u> problems ask for the object that maximizes (or minimizes) some polynomial time-computable objective.

Reduction Summary



NP Complete

The problems on the last slide are all NP-Complete/Hard. This means that if *any* problem in NP is fundamentally hard (a widely held belief), then these particular problems are.

Today

- Dealing with NP Completeness
- Basic Methods
- Backtracking / Branch and Bound

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.

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.

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

If your problem is NP-Hard/NP-Complete…

Then unless P=NP, there is no algorithm that gives the exact answer to your problem on all instances in polynomial time.

Bad News

What are the loopholes here?

If your problem is NP-Hard/NP-Complete…

Then unless P=NP, there is no algorithm that gives the exact answer to your problem on all instances in polynomial time.

Bad News

What are the loopholes here?

Prove P=NP?

If your problem is NP-Hard/NP-Complete…

Then unless P=NP, there is no algorithm that gives the exact answer to your problem on all instances in polynomial time.

Bad News

What are the loopholes here?

Prove P=NP?

If your problem is NP-Complete…

Approximation Algorithms/ Local Search

Then $\boxed{\text{unless P=NP}}$, there is no algorithm that gives the $\boxed{\text{exact answer}}$ to your problem on all instances in polynomial time.

Bad News

What are the loopholes here?

Prove P=NP?

Approximation Algorithms/ Local Search

See if you can modify problem.

If your problem is NP-Complete

Then unless P=NP, there is no algorithm that gives the exact answer to your problem on all instances in polynomial time.

Bad News

What are the loopholes here?

If your problem is NP-Complete

Then unless P=NP, there is no algorithm that gives the exact answer to your problem on all instances in polynomial time.

Prove P=NP?

Approximation Algorithms/ Local Search

See if you can modify problem.

Fixed Parameter Tractability

See if you can make further assumptions.

Bad News

What are the loopholes here?

If your problem is NP-Complete

Then unless P=NP, there is no algorithm that gives the exact answer to your problem on all instances in polynomial time.

Prove P=NP?

Approximation Algorithms/ Local Search

See if you can modify problem.

Fixed Parameter Tractability

See if you can make further assumptions.

Efficient Search

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.

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.

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 <u>deductions</u>.

- Use the rules to show that some square can only be filled out in one way.

Deductions

One way to progress is so make <u>deductions</u>.

- 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.

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

Example

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

First clause implies x = True.

Example

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

First clause implies x = True.

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

Example

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

First clause implies x = True.

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

First clause implies y = True.

Example

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

First clause implies x = True.

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

First clause implies y = True.

Plugging in and simplifying gives: (z).

Example

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

First clause implies x = True.

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

First clause implies y = True.

Plugging in and simplifying gives: (z).

So we must have x = y = z = 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.

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?

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?

Still Stuck?

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

There is a general strategy that can <u>always</u> be made to work.

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

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

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

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

2nd clause: y = False
4th clause: z = False
Contradicts 1st clause.

Example

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

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

Example

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

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

1^{st} clause: y = True
3^{rd} clause: z = True
Contradicts 4^{th} clause.

Example

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

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

1st clause: y = True
3rd clause: z = True
Contradicts 4th 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.

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,…
  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 = True, or x_i = False
- Try all possible numbers in a square in Sudoku
- Try all possible edges in Hamiltonian Cycle

Splitting

How do you split S into parts?

- Pick variable x_i and set x_i = True, or x_i = 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 it's 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 <u>much</u> better than brute force.

Runtime

These problems are still NP-Hard. Worst case, backtracking will still take exponential time. But it is usually <u>much</u> 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 <u>best</u> one).

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:

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.

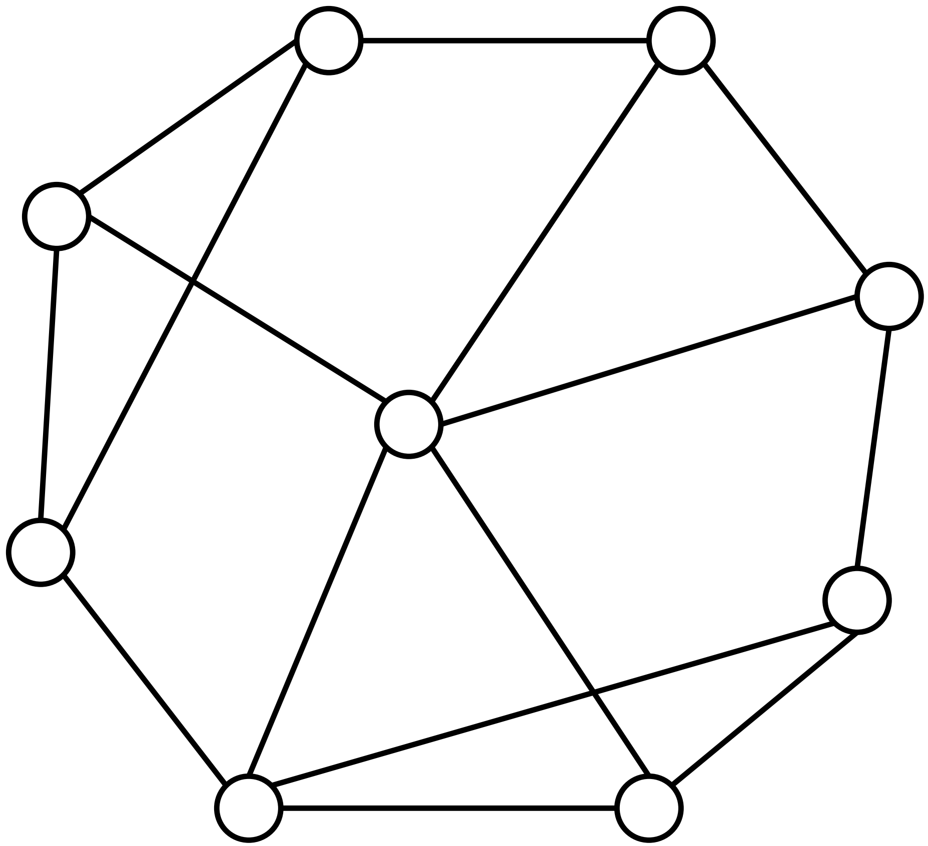
Branch & Bound

To get rid of bad paths do two things:

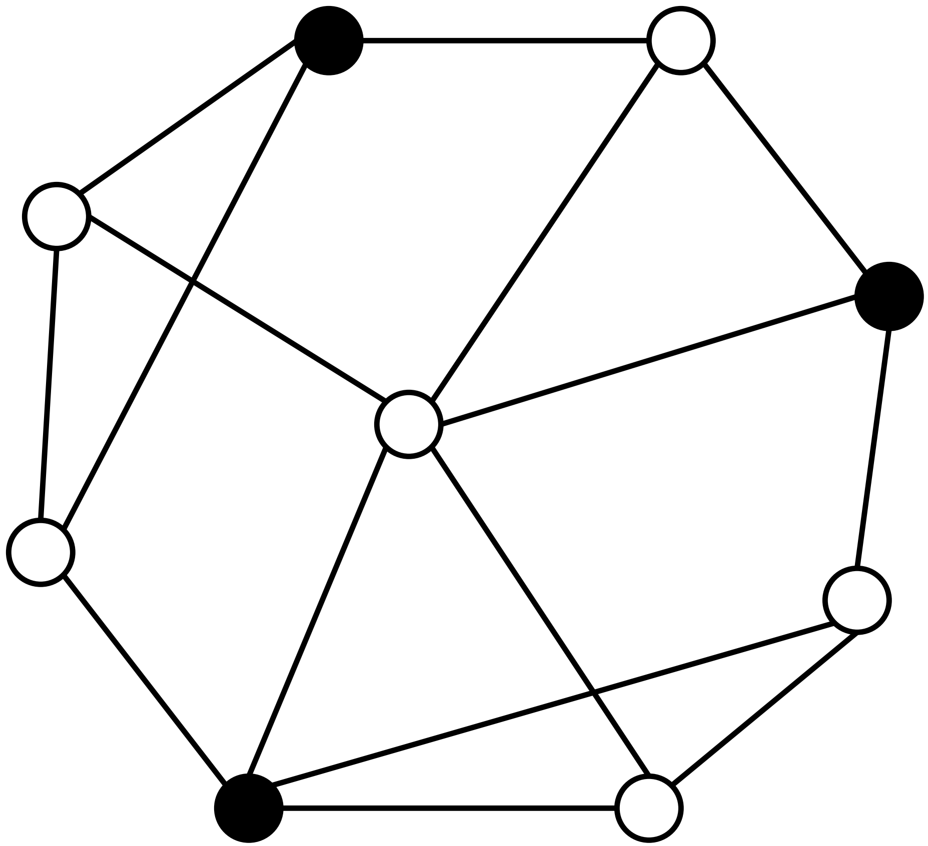
1) Keep track of the best solution you have found so far.
2) Try to prove upper bounds on your subproblems.

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

Example: Maximum Independent Set

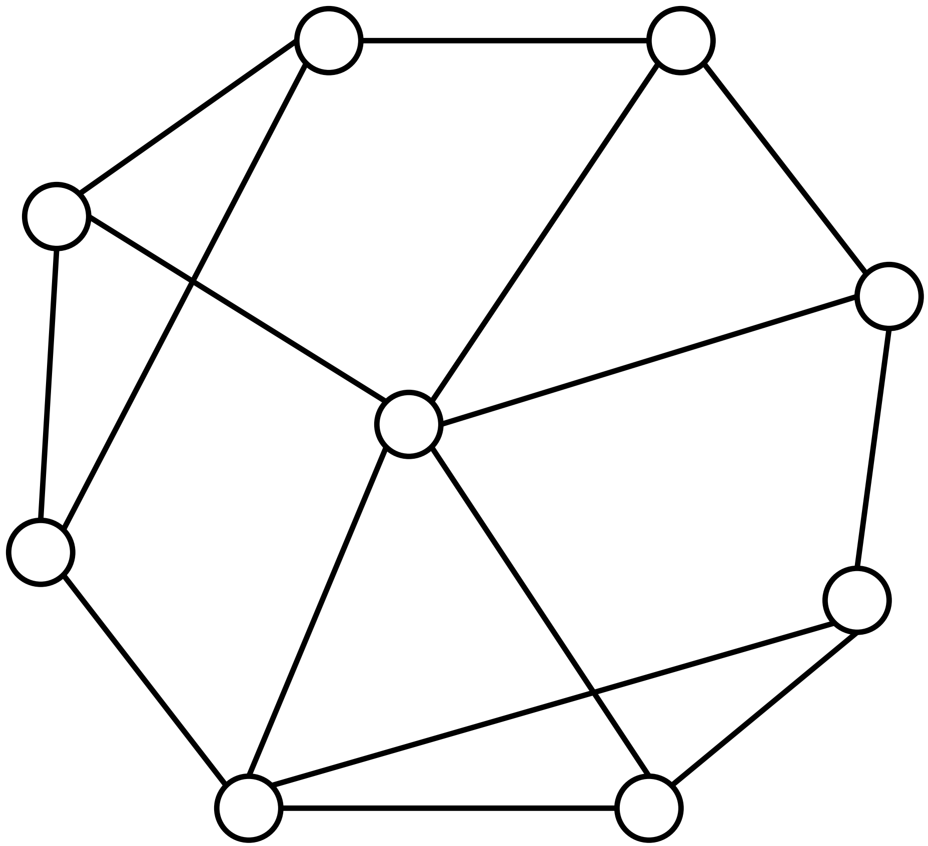


Example: Maximum Independent Set



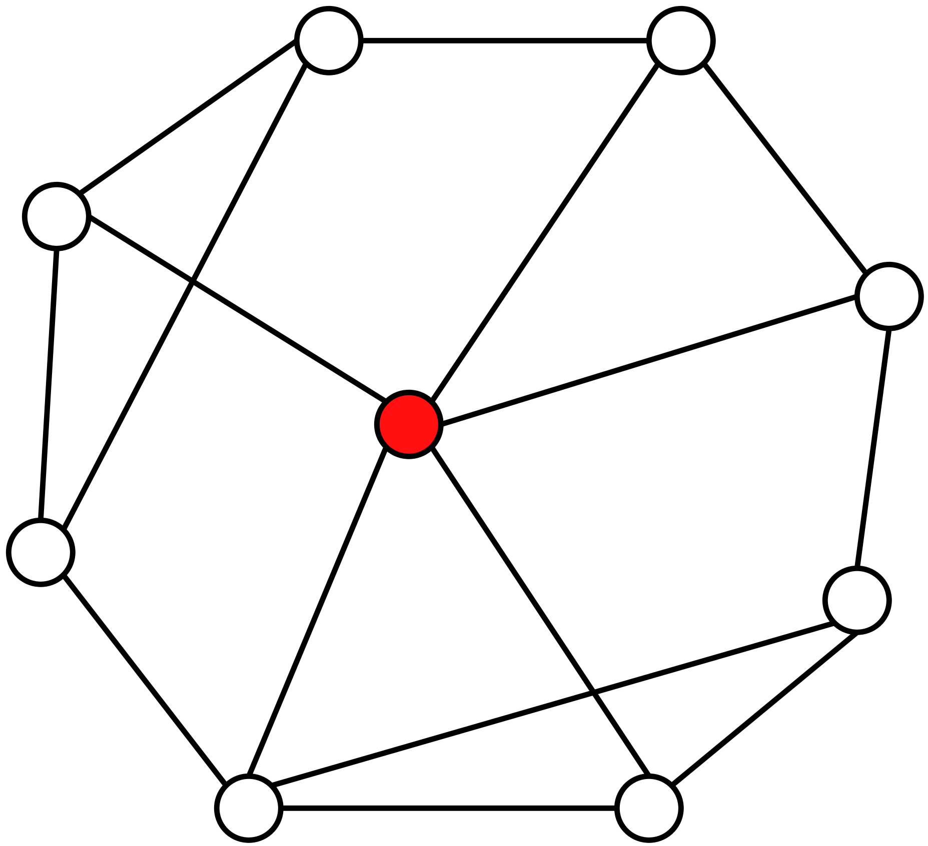
Set of size 3.

Example: Maximum Independent Set



Set of size 3.

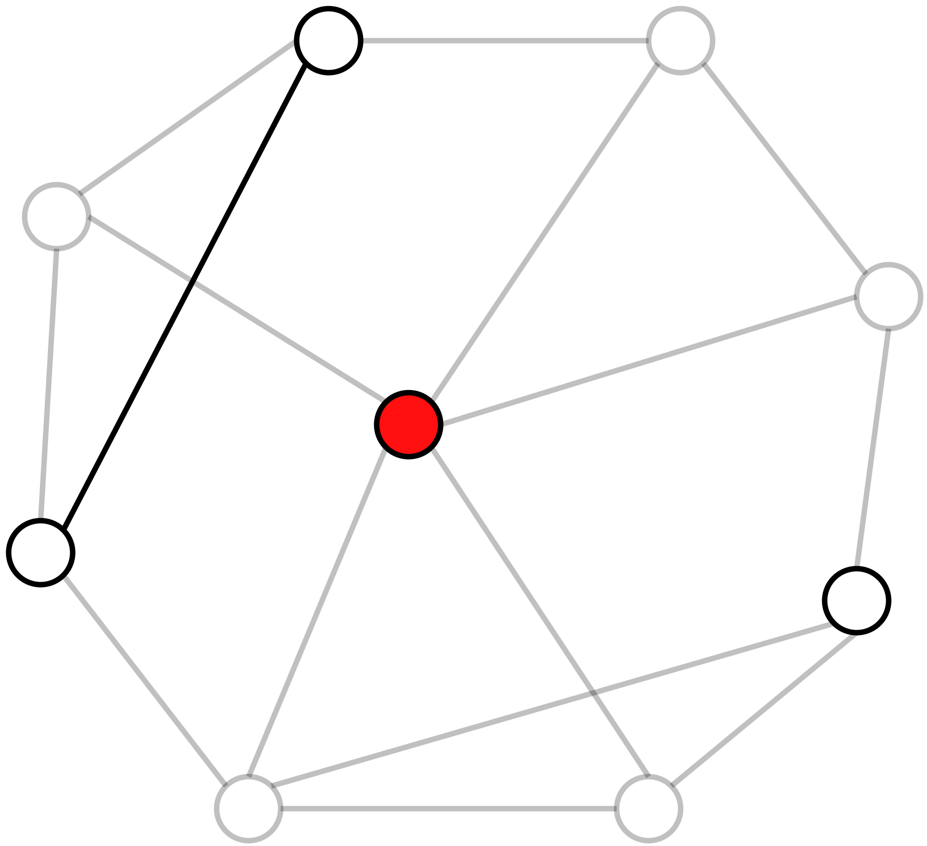
Example: Maximum Independent Set



Set of size 3.

If we have this point...

Example: Maximum Independent Set



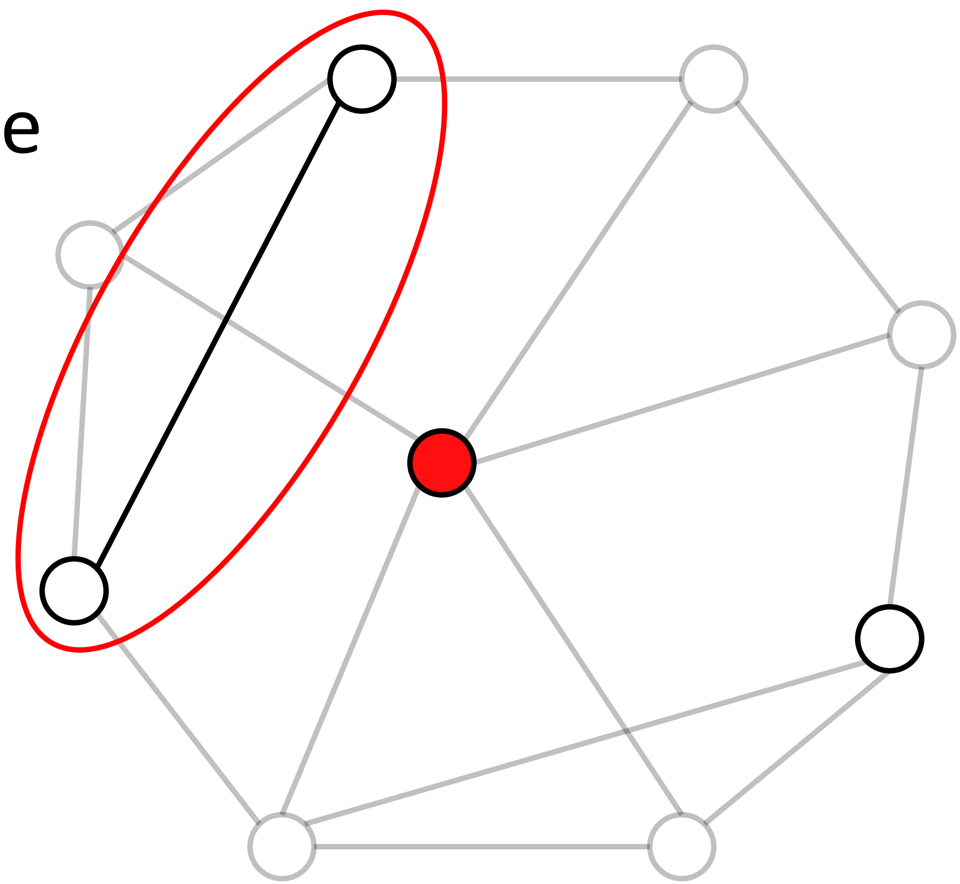
Set of size 3.

If we have this point…

Can't have any of these.

Example: Maximum Independent Set

Can't have both of these



Set of size 3.

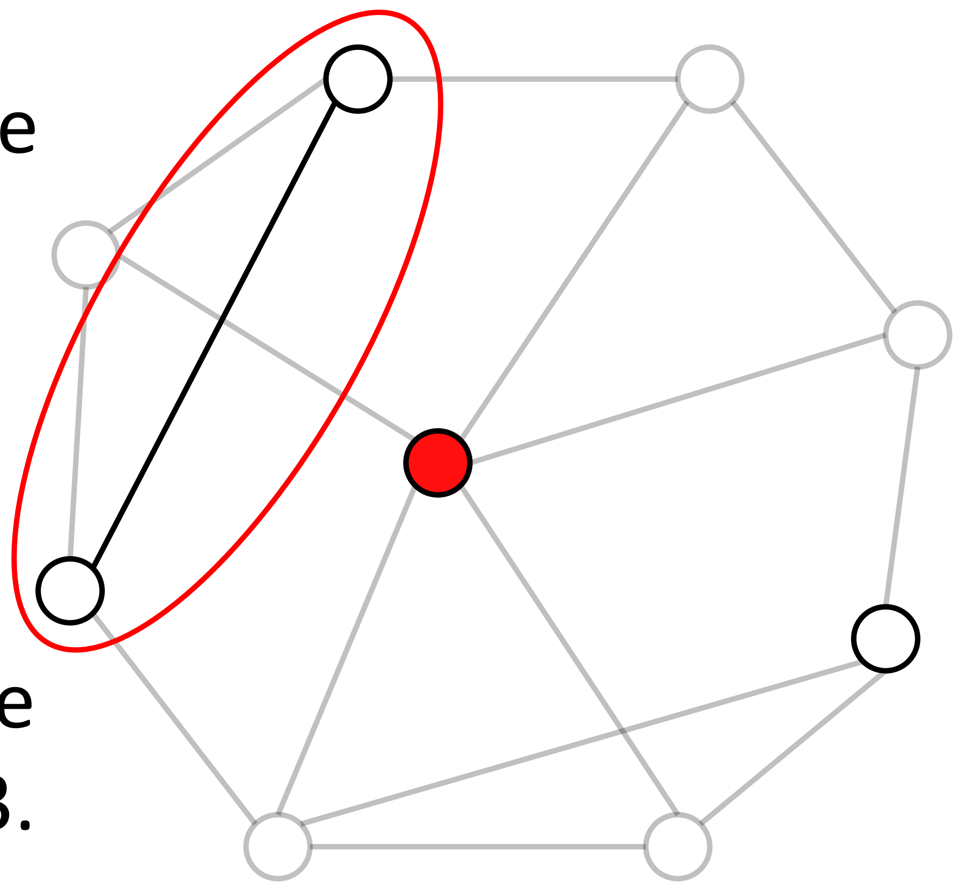
If we have this point…

Can't have any of these.

Example: Maximum Independent Set

Can’t have both of these

Set of size at most 3.



Set of size 3.

If we have this point…

Can’t have any of these.

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
```