Final Exam Review

CSE 101

Winter 2023

Exam Details

- Wednesday 11:30-2:30
- Randomized assigned seats
- You may use 12 one-sided pages of notes
- No textbook or electronic aids
- No need to provide proofs unless asked for
- 6 Questions in 3 hours

This Review

- Brief outline of **new** topics that might show up on the exam
  - For a review of previous topics, see the old exam review videos
- To see anything in more depth use other review options.

Other Review Options

- Lecture podcasts / slides
- Textbook
- OH questions
- Old exams from problem archive

Topics

- NP Completeness
  - NP Decision/Optimization problems
  - CSAT/3SAT/MIS/ZoE/ SubsetSum/Knapsack/ HamCycle/TSP
  - Reductions
  - NP Complete/Hard problems
- Dealing with NP Completeness
  - Ways of getting around P≠NP
  - Backtracking / Branch & Bound
  - Local search
  - Approximation Algorithms
  - MAXCUT/Vertex cover

NP-Completeness (Ch 8)

- NP-Problems
- Reductions
- NP-Completeness & NP-Hardness
- SAT
- Hamiltonian Cycle
- Zero-One Equations
- Knapsack

Brute Force Algorithms

For almost every problem we have seen there has been a (usually bad) naïve algorithm that just considers every possible answer and returns the best one.

- Is there a path from s to t in G?
- What is the longest common subsequence?
- What is the closest pair of points?
- Does G have a topological ordering?

NP

Such problems are said to be in Nondeterministic Polynomial time (NP).

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

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

Optimization vs. Decision

Note that these are not too different.

- Every decision problem can be phrased as an optimization problem (objective has value 1 if the object satisfies the condition and 0 otherwise).
- Every optimization problem has a decision form (can we find an example whose objective is more than x).

SAT

Problem: Formula-SAT

Given a logical formula in a number of Boolean variables, is there an assignment to the variables that causes the formula to be true?

$$(x \vee y) \wedge (y \vee z) \wedge (z \vee x) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$$
$$x = \text{True}, y = \text{True}, z = \text{False}$$

$$(x \vee y) \wedge (y \vee z) \wedge (z \vee x) \wedge (\bar{x} \vee \bar{y}) \wedge (\bar{y} \vee \bar{z}) \wedge (\bar{z} \vee \bar{x})$$
No satisfying assignment.

Hamiltonian Cycle (in text as Rudruta Path)

Given an undirected graph G is there a cycle that visits every vertex exactly once?

Brute Force Search

- Every NP problem has a brute force search algorithm.
- Throughout this class we have looked at problems with algorithms that substantially improve on brute force search.
- Does every NP problem have a better-than-brute-force algorithm?

P vs. NP

$1,000,000 Question: Is P = NP?

Is it the case that every problem in NP has a polynomial time algorithm?

- If yes, every NP problem has a reasonably efficient solution.
- If not, some NP problems are fundamentally difficult

Most computer scientists believe P ≠ NP.
(But proving anything is very very hard)

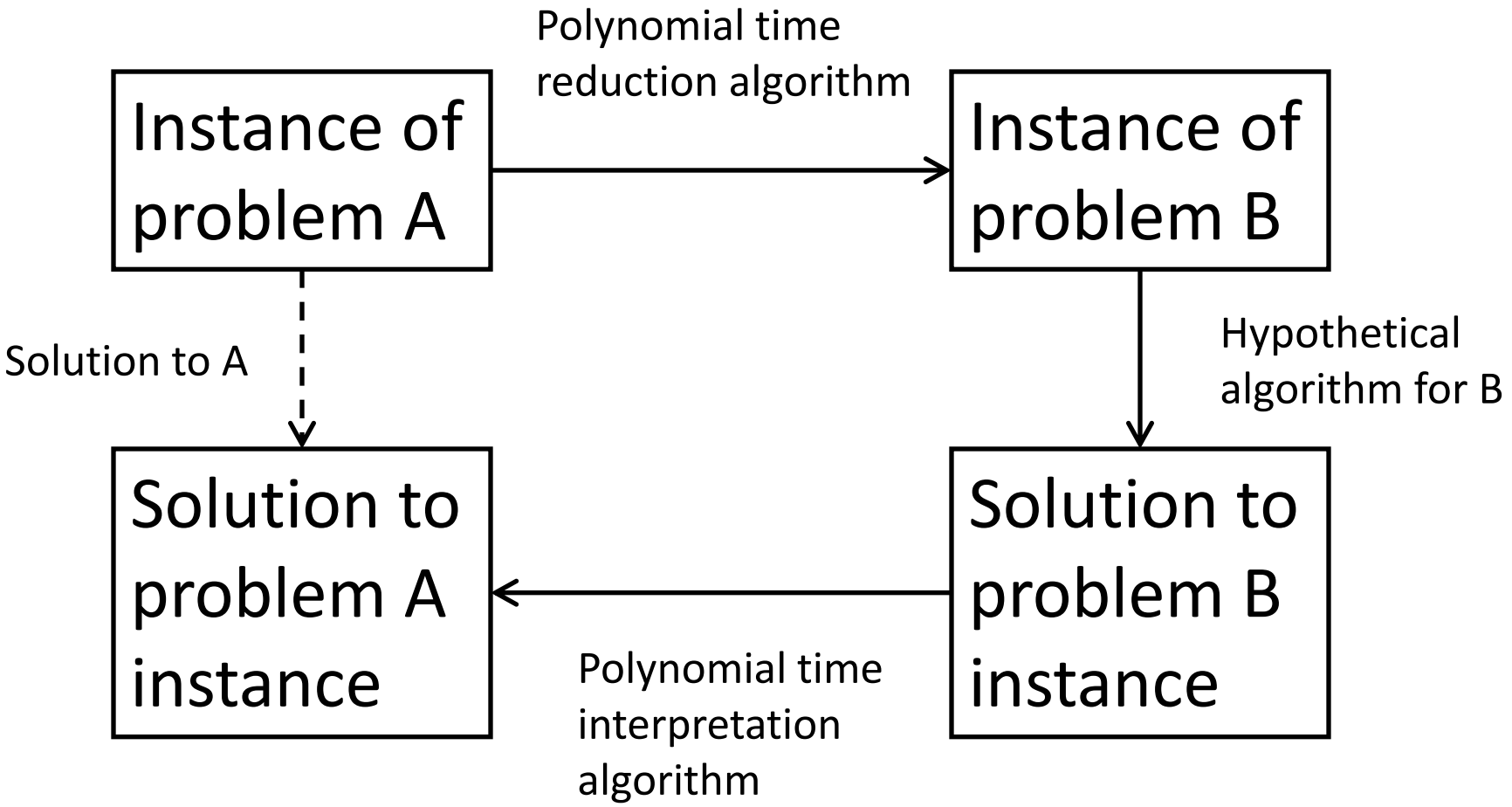
Reductions

Reductions are a method for proving that one problem is at least as hard as another.

How do we do this?

We show that if there is an algorithm for solving B, then we can use this algorithm to solve A. Therefore, A is no harder than B.

Reduction A → B



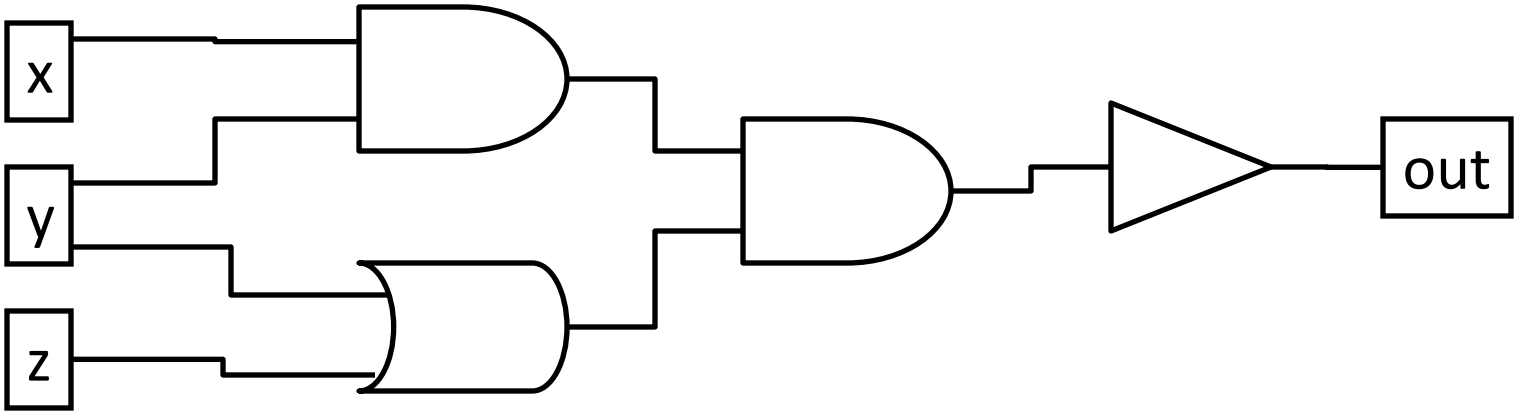
Reduction A → B

If we have algorithms for reduction and interpretation:

- Given an algorithm to solve B, we can turn it into an algorithm to solve A.
- This means that A might be easier to solve than B, but cannot be harder.

Circuit SAT

Problem: Given a circuit C with several Boolean inputs and one Boolean output, determine if there is a set of inputs that give output 1.



Important Reduction:

Any NP decision problem → Circuit SAT

NP-Complete

Circuit-SAT is our first example of an NP-Complete problem. That is a problem in NP that is at least as hard as any other problem in NP.

- **Good news:** If we find a polynomial time algorithm for Circuit-SAT, we have a polynomial time algorithm for all NP problems!
- **Bad news:** If any problem in NP is hard, Circuit-SAT is hard.

Note: Decision problems can be NP-Complete. For optimization problems, it is called NP-Hard.

3-SAT

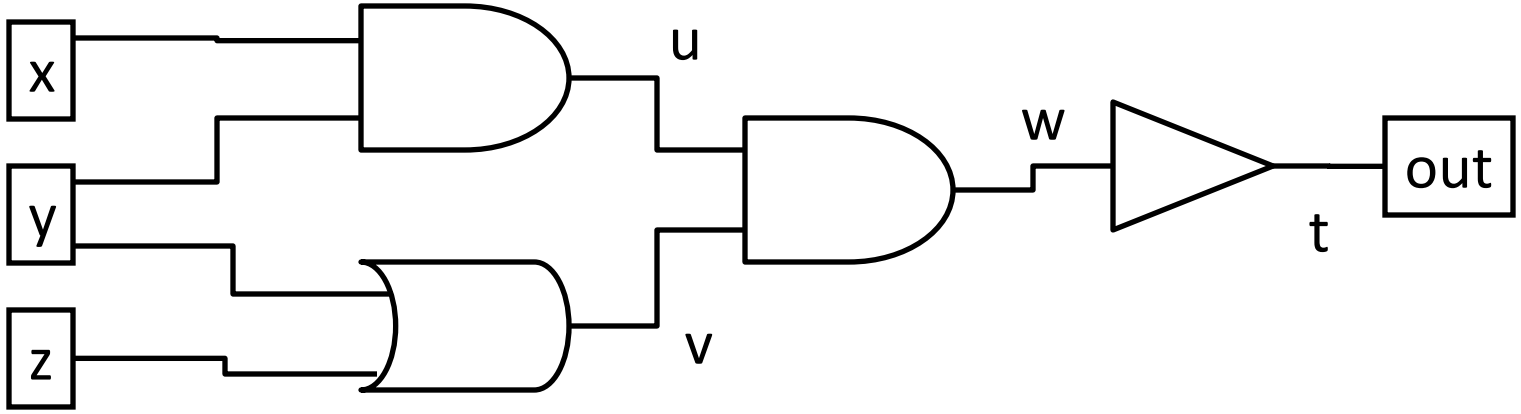
3-SAT is a special case of formula-SAT where the formula is an AND of clauses and each clause is an OR of at most 3 variables or their negations.

Example:

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

Circuit-SAT → 3-SAT

- Start with circuit



- Create variable for each wire
- Create formula with clause for each gate and output

$$(v \iff y \vee z) \wedge (u \iff x \wedge y) \wedge (w \iff u \wedge v) \wedge (t \iff \bar{w}) \wedge (t)$$

These Aren't 3-SAT Clauses

We have 3-variable clauses that aren't 3-SAT clauses. Write it in terms of them.

- Write truth table
- Each 3-SAT clause sets one output to false.

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

$$= (z \iff x \vee y)$$

x	y	z	$z \iff x \vee y$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Another Look at 3-SAT

Lemma: A 3-SAT instance is satisfiable if and only if it is possible to select one term from each clause without selecting both a variable and its negation.

Example:

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

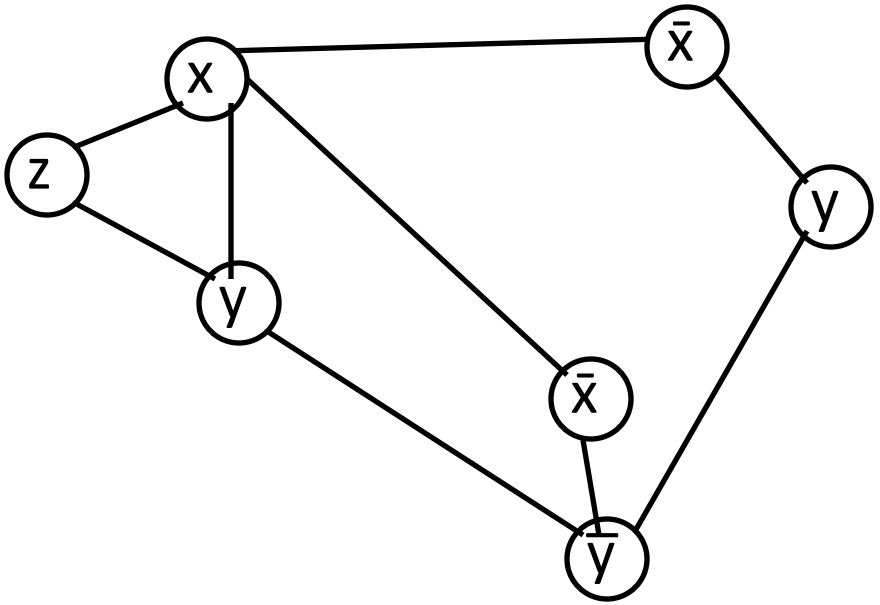
3-SAT → Maximum Independent Set

Want to encode this select one term from each clause as a graph.

- Create one vertex for each term in each clause.
- Edges between terms in same clause.
- Edges between contradictory terms.

Example:

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



Zero-One Equations

Problem: Given a matrix A with only 0 and 1 as entries and b a vector of 1s, determine whether or not there is an x with 0 and 1 entries so that

$$Ax = b.$$

Example

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

x_1 x_2 x_3 x_4 x_5 x_6 x_7

One term per clause:

$x_1 + x_2 + x_3 = 1$

$x_4 + x_5 = 1$

$x_6 + x_7 = 1$

Replace
$a + b \leq 1$ with
$a + b + c = 1$

No Contradictions:

$x_1 + x_4 + x_8 = 1$

$x_1 + x_7 + x_9 = 1$

$x_2 + x_6 + x_{10} = 1$

$x_5 + x_6 + x_{11} = 1$

General Construction

- Create one variable per term
- For each clause, create one equation
- For each pair of contradictory term, create an equation with those two and a new variable

Another Way of Looking at ZOE

Recall if $A = [v_1\ v_2\ v_3\ \ldots\ v_n]$,

$Ax = x_1 v_1 + x_2 v_2 + x_3 v_3 + \ldots + x_n v_n$.

Example:

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

$$\begin{aligned} &x_1 * [\ 1\ 0\ 0\ 1\ ] + \\ &x_2 * [\ 0\ 0\ 1\ 1\ ] + \\ &x_3 * [\ 1\ 1\ 1\ 0\ ] \\ &\text{----------} \\ &= \quad [\ 1\ 1\ 1\ 1\ ] \end{aligned}$$

What if we treated these as numbers rather than vectors?

Subset Sum

Problem: Given a set S of numbers and a target number C, is there a subset $T \subseteq S$ whose elements sum to C.

Alternatively: Can we find $x_y \in \{0,1\}$ so that

$$\sum_{y \in S} x_y y = C.$$

Reduction: ZOE → Subset Sum.

Subset Sum → Knapsack

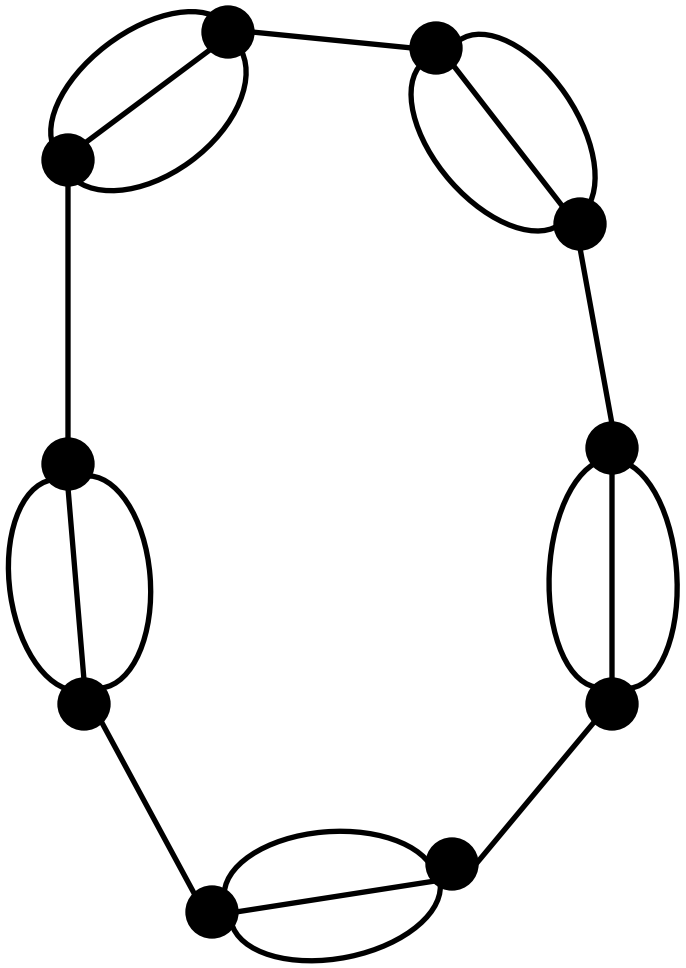
- Create Knapsack problem where for each item Value(item) = Weight(item).
- Maximizing value is the same as maximizing weight (without going over capacity).
- We can achieve value = capacity if and only if there is a subset of the items with total weight equal to capacity.

ZoE → Hamiltonian Cycle Strategy

Often in order to show that a problem is NP-Complete, you want to be able to show that it can simulate logic somehow.

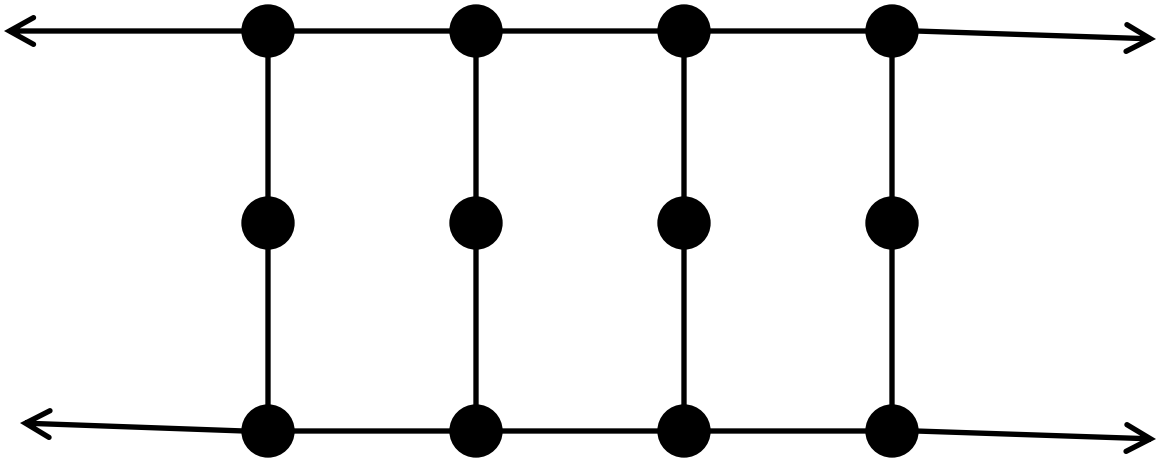
This is a bit difficult for Hamiltonian Cycle as most graphs have too many options, so we will want to find specific graphs with clear, binary choices.

Strategy



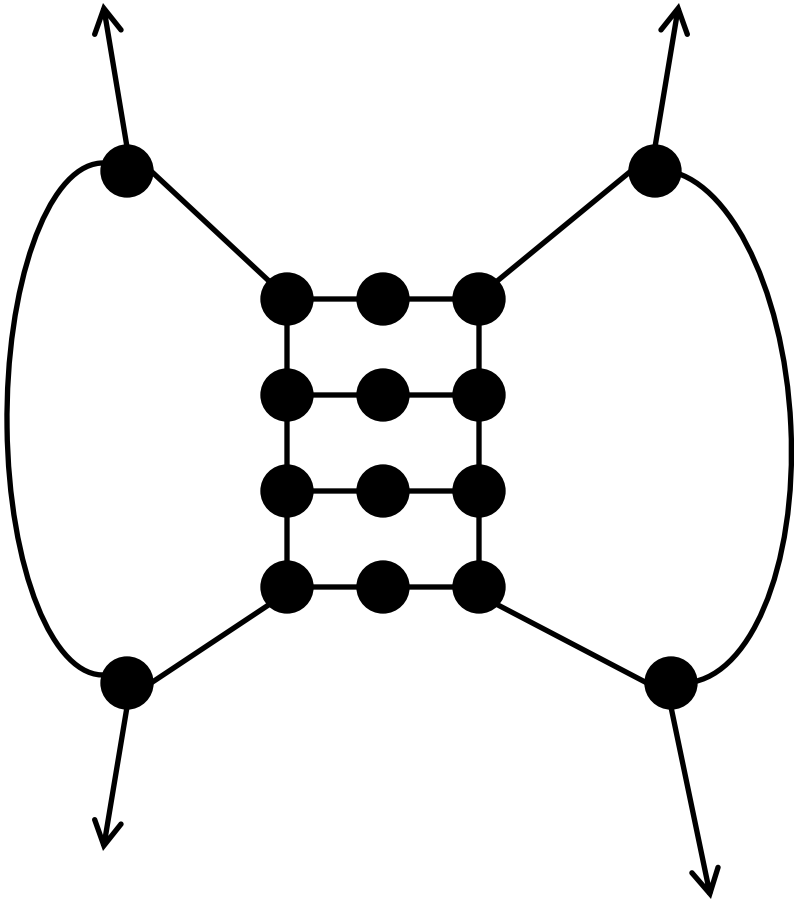
- Start with a cycle
- Double up some edges
- Cycle must pick one edge from each pair.
    - This provides a nice set of binary variables
- Need a way to add restrictions so that we can't just use <u>any</u> choices.

Gadget



- Must use these edges.
- Two ways to fill out.

Gadget Use



- Hook gadget up between a pair of edges.
- Hamiltonian Cycle must use exactly one of the connected edges.
- This allows us to force logic upon our choices.

Full Construction

Choices:

- For each variable, choose either 0 or 1.
- For each equation, choose one variable.

Constraints:

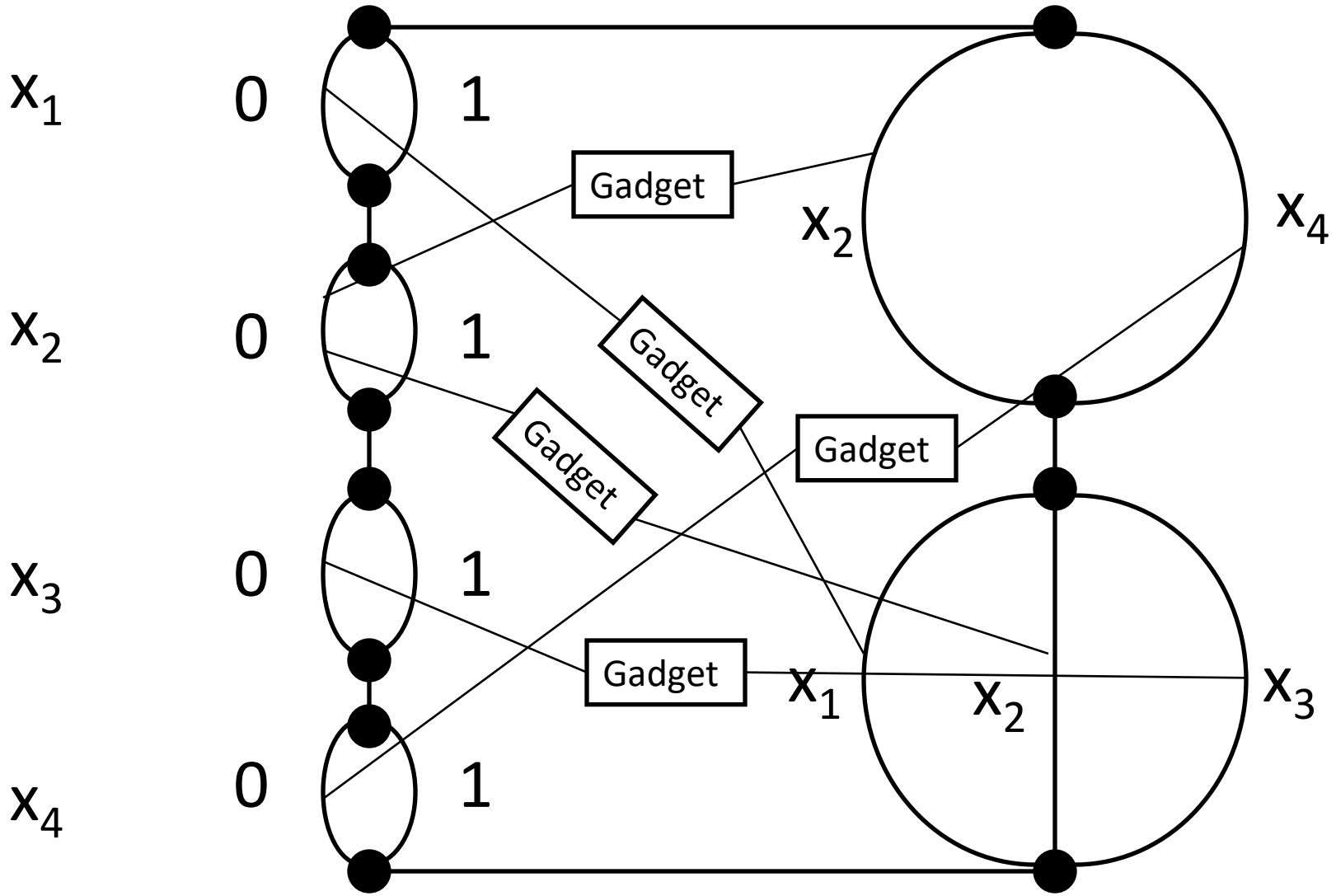
- For each variable that appears in an equation, exactly one of the following should be selected:
  - That variable in that equation
  - That variable equal to 0

$x_1=1\ x_2=0\ x_3=0\ x_4=1$

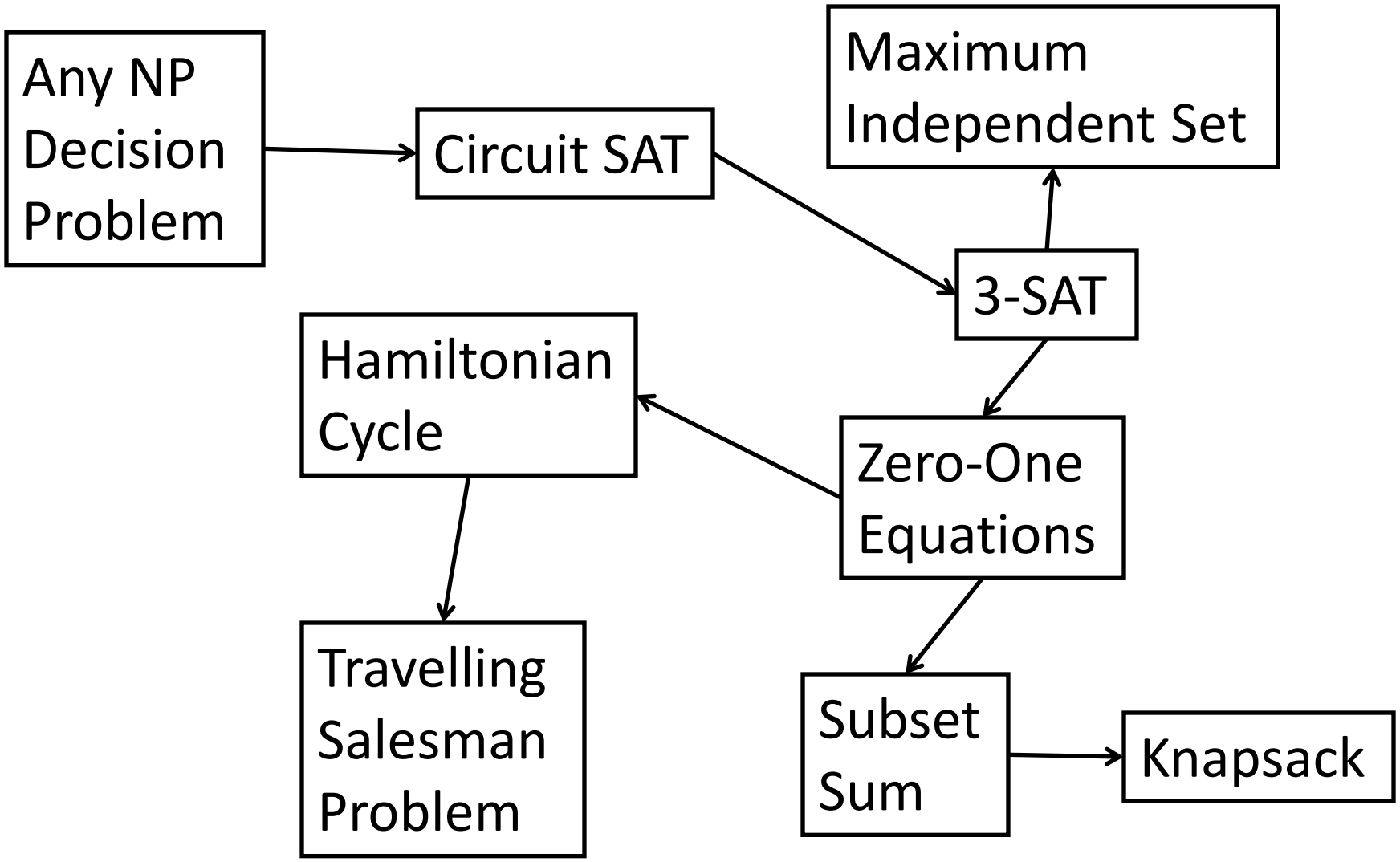
Example

x1+x2+x3=1

x2+x4=1



Reduction Summary



Dealing With NP-Completeness (Ch 9)

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

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.

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

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 S1,S2,…
   For each i,
      Run Backtracking(P,Si)
   Return any solutions found
```

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.

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 ← Random initial point
  Try all y close to x
    If f(y) > f(x) for some y
      x ← y
      Repeat
    Else Return x
```

MAXCUT

Problem: Given a graph G find a way to color the vertices of G black and white so that as many edges as possible have endpoints of different colors.

How to Get Unstuck

- Randomized Restart
  - If you try many starting points, hopefully, you will find one that finds you the true maximum.
- Expand Search Area
  - Look for changes to 2 or 3 vertices rather than 1.
    - Larger area means harder to get stuck
    - Larger area also takes more work per step
- Still no guarantee of finding the actual maximum in polynomial time.

Simulated Annealing

- At the start of algorithm take big random steps.
  - Hopefully, this will get you onto the right “hill”.
- As the algorithm progresses, the “temperature” decreases and the algorithm starts to fine tune more precisely.
- Works well in practice on a number of problems.

MAXCUT Minimal Value

Look back at local search for MAXCUT.

- Swap a vertex if most of its neighbors are the same color.
- At the end of the algorithm most of a vertices neighbors are the opposite color.
- At the end of the algorithm at least half of the edges are cut.
- Get cut of size at least $|E|/2$, but optimum at most $|E|$.

Approximation Algorithms

An α-approximation algorithm to an optimization problem is a (generally polynomial time) algorithm that is guaranteed to produce a solution within an α-factor of the best solution.

Our local search algorithm for MAXCUT is a 2-approximation algorithm.

Often approximation algorithms can produce good enough solutions.

Vertex Cover

Problem (Vertex Cover): Given a graph G find a set S of vertices so that every edge of G contains a vertex of S and so that |S| is as small as possible.

Greedy Algorithm

```
GreedyVertexCover(G)
   S ← {}
   While(S doesn't cover G)
      (u,v) ← some uncovered edge
      Add u and v to S
   Return S
```

Analysis

Algorithm finds k edges and 2k vertices.

- Edges are vertex-disjoint.
- Any cover must have at least one vertex on each of these edges.
- Optimum cover has size at least k.
- We have a 2-approximation.

Knapsack

Even though general knapsack is NP-Hard, we have a polynomial time algorithm if all weights are small integers (or more generally small integer multiples of some common value).

Since everything can be rounded to small integers, we have an algorithm idea.

Small Values

Actually rounding the weights doesn't quite work. It gives you sets which almost fit in the sack.

Instead, we want to round the values of the items and for this, we need a new algorithm.

Dynamic Program

Let $\text{Lightest}_{\leq k}(V)$ be the weight of the lightest collection of the first k items with total value V.

We have a recursion

$\text{Lightest}_{\leq k}(V) = \min\{\text{Lightest}_{\leq k-1}(V), \text{Wt}(k) + \text{Lightest}_{\leq k-1}(V - \text{Val}(k))\}$

This gives a DP.

#subprobs = [Total Value][#items]

Time/Subproblem = O(1).

Approximation Algorithm

1) Throw away items that don't fit in sac.
2) Let V_0 be highest value of item.
3) Round each item's value to closest multiple of δV_0.
4) Run the small integer values DP.

Runtime: Values integer multiples of δV_0. Total value at most $[\#items]V_0 = ([\#items]/\delta)\ \delta V_0$.

Total runtime $O([\#items]^2/\delta)$.

Approximation Analysis

Optimal value at least V_0.

Rounding changes the value of any set of items by at most [#items]δV_0.

The solution we find is at least as good as the optimal after round.

Our solution is within [#items]δV_0 of optimal.

Combining

Let $\delta = \varepsilon/[\#\text{items}]$.

$OPT \geq V_0$.

Our solution is at most εV_0 worse.

Have a $(1+\varepsilon)$-approximation algorithm.

<u>Runtime</u> = $O([\#\text{items}]^3/\varepsilon)$

For any $\varepsilon > 0$, have a $(1+\varepsilon)$-approximation in polynomial time. (known as a PTAS).