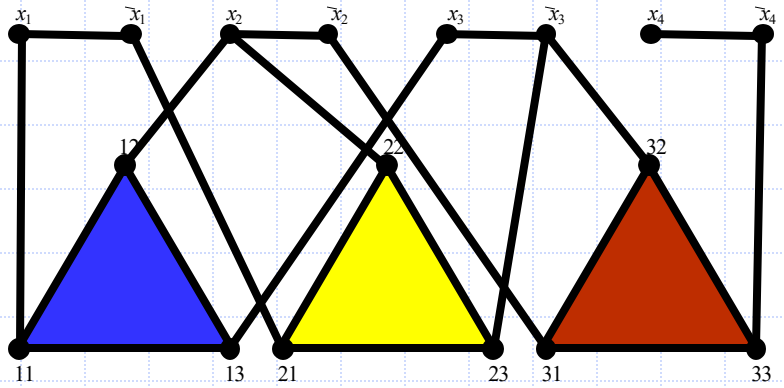
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

NP-Complete Problems



Search Problems

- Note that almost all of what we have done thus far has involved searches
  - Shortest path
  - Minimum spanning tree
  - Maximum flow
- In most cases, the number of possibilities has been exponential, and yet our solutions were polynomial time

Search Problems

- Note that almost all of what we have done thus far has involved searches
  - Shortest path – number of paths?
  - Minimum spanning tree – number of spanning trees (for n vertices n^{n-2})
  - Maximum flow – number of flows?
- Poly algorithms means a great success!

Search Problems

- All great successes
- Now we meet the failures
  - Best known solutions are exponential
  - And often not much better than exhaustive search
  - Yet some seem surprisingly simple…

SATISFIABILITY (SAT)

$$(x \vee y \vee z)\ (x \vee \overline{y})\ (y \vee \overline{z})\ (z \vee \overline{x})\ (\overline{x} \vee \overline{y} \vee \overline{z}).$$

This is a *Boolean formula in conjunctive normal form (CNF)*. It is a collection of *clauses* (the parentheses), each consisting of the disjunction (logical *or*, denoted $\vee$) of several *literals*, where a literal is either a Boolean variable (such as x) or the negation of one (such as $\overline{x}$). A *satisfying truth assignment* is an assignment of `false` or `true` to each variable so that every clause contains a literal whose value is `true`. The SAT problem is the following: given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

SAT is a search problem

- What property do we want of such search problems?

SAT is a search problem

- What property do we want of such search problems?

A *search problem* is specified by an algorithm C that takes two inputs, an instance I and a proposed solution S, and runs in time polynomial in $|I|$. We say S is a solution to I if and only if $C(I, S) = \texttt{true}$.

SAT

- SAT is a hard problem
- Over 50 years of research has failed to find a solution that is better than exponential
- It can be difficult to distinguish hard problems from not-so-hard problems

SAT

- SAT is a hard problem
- Over 50 years of research has failed to find a solution that is better than exponential
- It can be difficult to distinguish hard problems from not-so-hard problems

2SAT

In the 2SAT problem, you are given a set of *clauses*, where each clause is the disjunction (OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value `true` or `false` to each of the variables so that *all* clauses are satisfied – that is, there is at least one true literal in each clause. For example, here's an instance of 2SAT:

$$(x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (x_1 \vee x_2) \wedge (\overline{x}_3 \vee x_4) \wedge (\overline{x}_1 \vee x_4).$$

- 2SAT is not a hard problem
  - It amounts to finding the strongly connected components of a carefully constructed directed graph
  - And can be solved in linear time

3SAT

- 3SAT is just what you expect
- And turns out to be a hard problem

Two Problems

- Given a graph G, find a path through the graph that hits each **edge** exactly once, or report no such path exists
  - Method of Euler solves this in poly time
- Given a graph G, find a path through the graph that hits each **vertex** exactly once, or report no such path exists
  - This problem is hard

Two Problems

- Given a graph G, find a path through the graph that hits each **edge** exactly once, or report no such path exists
- Given a graph G, find a path through the graph that hits each **vertex** exactly once, or report no such path exists
  - HAMILTIONIAN CYCLE
    - Though Hamilton only rediscovered it. Originally discovered by Rudrata

TSP

- Our old friend
- Is this a search problem?

TSP

- Our old friend
- Is this a search problem?

TSP

- Our old friend
- Is this a search problem?
- Can we make it one?

TSP

- Our old friend
- Is this a search problem?
- Can we make it one?
- Is this any different than the "optimization" version of TSP?
- Are search problems in general any different than "optimization" version?

More hard problems

- BALANCED CUT: Given a graph with n vertices and budget b, partition the vertices into two sets S and T such that both have at least n/3 vertices and such that there are at most b edges between S and T
- INTEGER LINEAR PROGRAMMING (ILP)
  - Solutions must be integers

More hard problems

There is a particularly clean special case of ILP that is very hard in and of itself: the goal is to find a vector $\mathbf{x}$ of 0's and 1's satisfying $\mathbf{Ax} = \mathbf{1}$, where $\mathbf{A}$ is an $m \times n$ matrix with $0-1$ entries and $\mathbf{1}$ is the m-vector of all 1's. It should be apparent from the reductions in Section 7.1.4 that this is indeed a special case of ILP. We call it ZERO-ONE EQUATIONS (ZOE).

- INDEPENDENT SET (budget g vertices)
- SET COVER (budget b subsets)
- VERTEX COVER (budget b vertices)
- SUBSET SUM (sum T)

Hard Problems and Easy Problems

Hard problems (**NP**-complete)	Easy problems (in **P**)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

Revisit and Old Friend

In the *longest increasing subsequence* problem, the input is a sequence of numbers $a_1, \ldots, a_n$. A *subsequence* is any subset of these numbers taken in order, of the form $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and an *increasing* subsequence is one in which the numbers are getting strictly larger. The task is to find the increasing subsequence of greatest length. For instance, the longest increasing subsequence of $5, 2, 8, 6, 3, 6, 9, 7$ is $2, 3, 6, 9$:

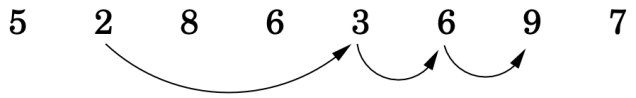
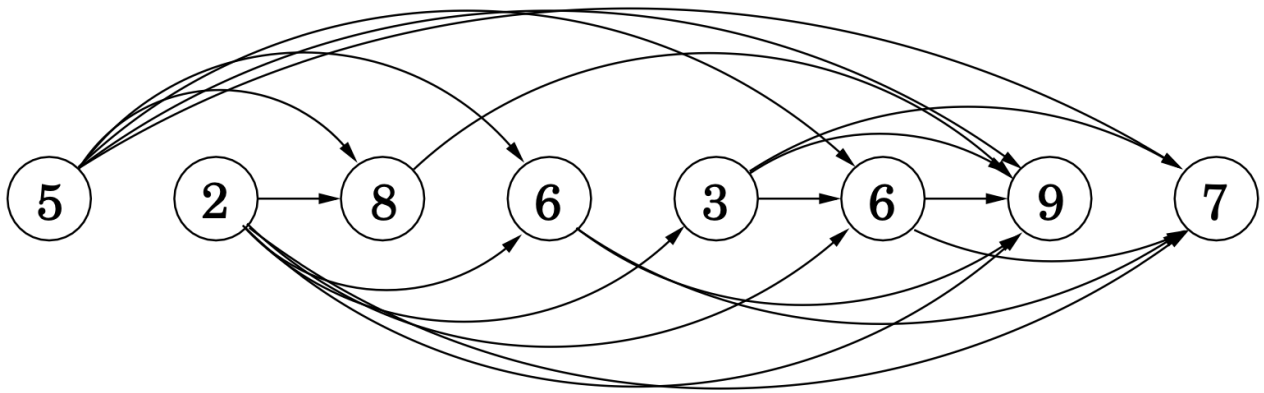


Figure 6.2 The dag of increasing subsequences.



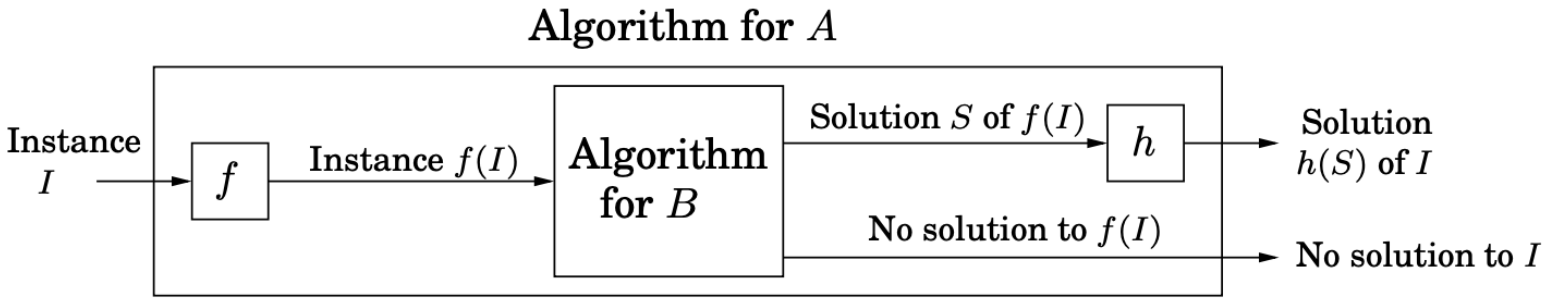
How are these related?

Question:

- When should two problems be considered the same?
  - Let's call the two search problems A and B

Question:

- When should two problems be considered the same?
  - Let's call the two problems A and B



Reductions

- A ***reduction*** from search algorithm A to search algorithm B is a polynomial time algorithm f that transforms any instance I of A into an instance f(I) of B, together with a polynomial time algorithm h that maps any solution S of f(I) into a solution h(S) of I.

Hard Problems and Easy Problems

Hard problems (**NP**-complete)	Easy problems (in **P**)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

It turns out that all of the hard problems on this list (and thousands more) are really the same problem!
Any of them can be reduced to any other!

The complexity class NP

It's time to introduce some important concepts. We know what a search problem is: its defining characteristic is that any proposed solution can be quickly checked for correctness, in the sense that there is an efficient checking algorithm C that takes as input the given instance I (the data specifying the problem to be solved), as well as the proposed solution S, and outputs `true` if and only if S really is a solution to instance I. Moreover the running time of $C(I, S)$ is bounded by a polynomial in $|I|$, the length of the instance. *We denote the class of all search problems by* **NP**.

The complexity class P

It's time to introduce some important concepts. We know what a search problem is: its defining characteristic is that any proposed solution can be quickly checked for correctness, in the sense that there is an efficient checking algorithm C that takes as input the given instance I (the data specifying the problem to be solved), as well as the proposed solution S, and outputs `true` if and only if S really is a solution to instance I. Moreover the running time of $C(I, S)$ is bounded by a polynomial in $|I|$, the length of the instance. *We denote the class of all search problems by* **NP**.

We've seen many examples of **NP** search problems that are solvable in polynomial time. In such cases, there is an algorithm that takes as input an instance I and has a running time polynomial in $|I|$. If I has a solution, the algorithm returns such a solution; and if I has no solution, the algorithm correctly reports so. *The class of all search problems that can be solved in polynomial time is denoted* **P**. Hence, all the search problems on the right-hand side of the table are in **P**.

NP-Complete

It's time to introduce some important concepts. We know what a search problem is: its defining characteristic is that any proposed solution can be quickly checked for correctness, in the sense that there is an efficient checking algorithm C that takes as input the given instance I (the data specifying the problem to be solved), as well as the proposed solution S, and outputs `true` if and only if S really is a solution to instance I. Moreover the running time of $C(I, S)$ is bounded by a polynomial in $|I|$, the length of the instance. *We denote the class of all search problems by* **NP**.

We've seen many examples of **NP** search problems that are solvable in polynomial time. In such cases, there is an algorithm that takes as input an instance I and has a running time polynomial in $|I|$. If I has a solution, the algorithm returns such a solution; and if I has no solution, the algorithm correctly reports so. *The class of all search problems that can be solved in polynomial time is denoted* **P**. Hence, all the search problems on the right-hand side of the table are in **P**.

A search problem is **NP***-complete if all other search problems reduce to it.*

NP-Complete

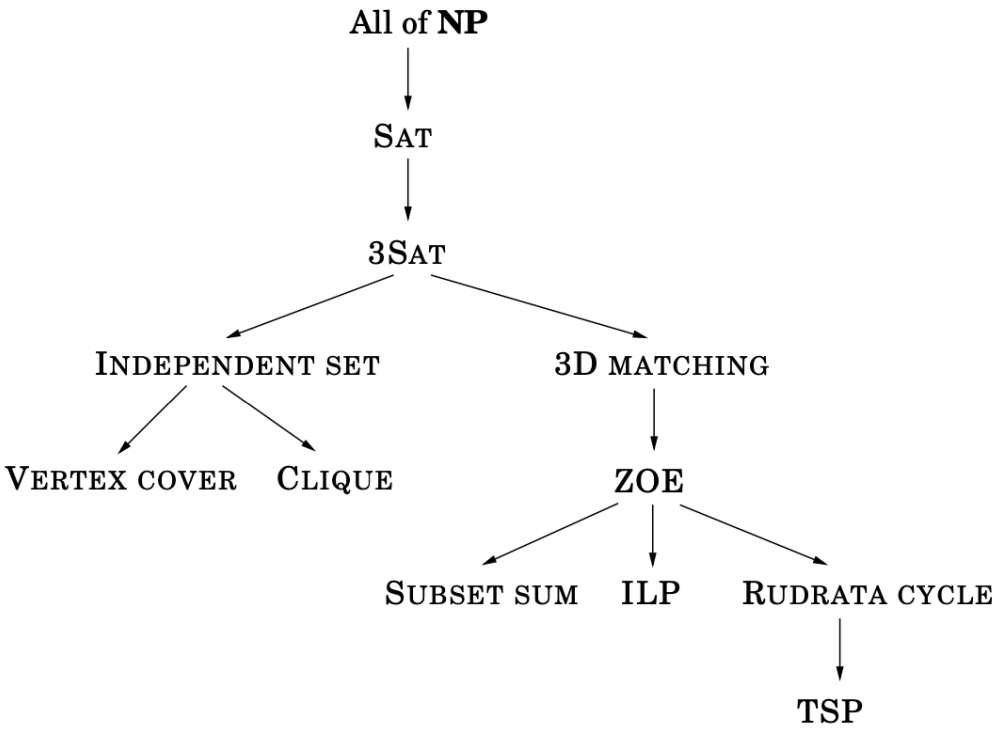
A search problem is **NP***-complete if all other search problems reduce to it.*

Think about how remarkable this is. For a problem to be NP-complete, it must be useful for solving *every* search problem in the world!

Not all hard problems are NP-complete

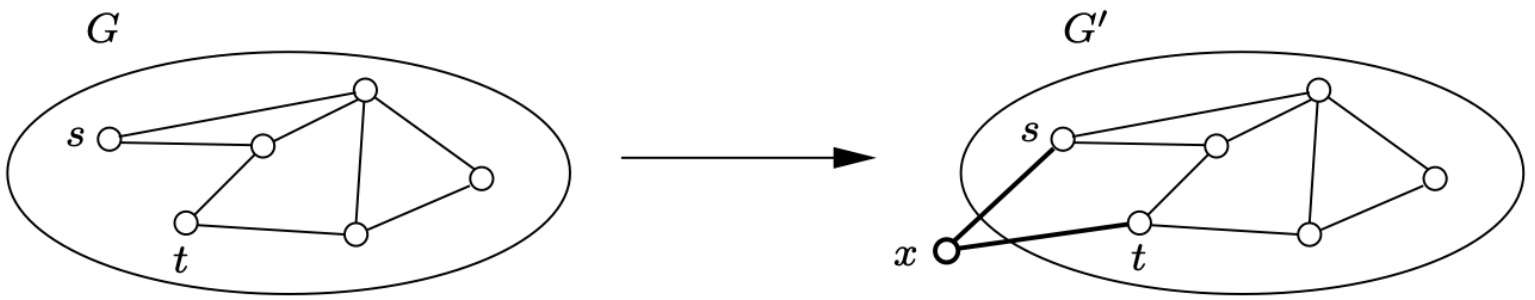
- Factoring is a hard problem
  - Many cryptographic primitives rely on this fact
- But not NP-complete
  - It's difficulty is different than NP-complete hard problems
    - "Or indicate no solution exists"
    - Can be solved using quantum computation

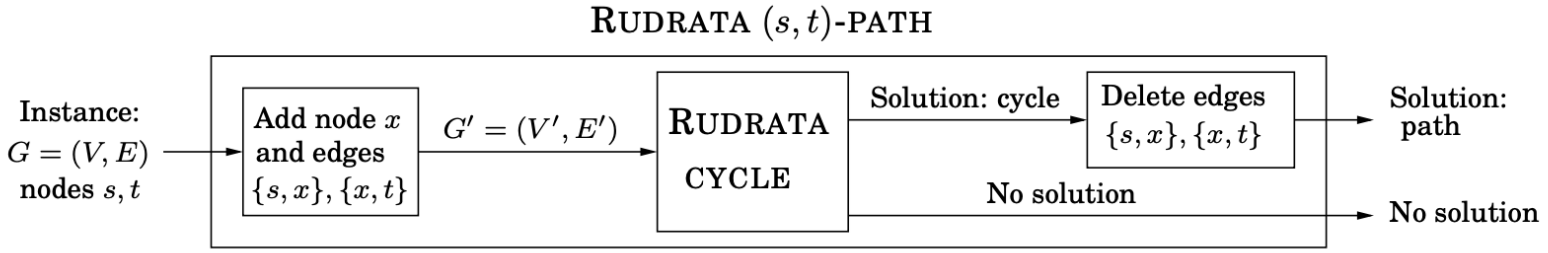
Figure 8.7 Reductions between search problems.



Reductions

- Hamiltonian (s,t)-path -> Hamiltonian Cycle
- Recall the problems
- You do the reduction
  - And remember what you need to prove!


G
s
t
G′
s
x
t


RUDRATA (s,t)-PATH
Instance:
$G=(V,E)$
nodes s,t
Add node x
and edges
$\{s,x\},\{x,t\}$
$G'=(V',E')$
RUDRATA
CYCLE
Solution: cycle
Delete edges
$\{s,x\},\{x,t\}$
Solution:
path
No solution
No solution

Reductions

3SAT⟶INDEPENDENT SET

One can hardly think of two more different problems. In 3SAT the input is a set of clauses, each with three or fewer literals, for example

$$(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y}),$$

First, state both problems.

Reductions

3SAT⟶INDEPENDENT SET

One can hardly think of two more different problems. In 3SAT the input is a set of clauses, each with three or fewer literals, for example

$$(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y}),$$

Now, how do you start to attack this?

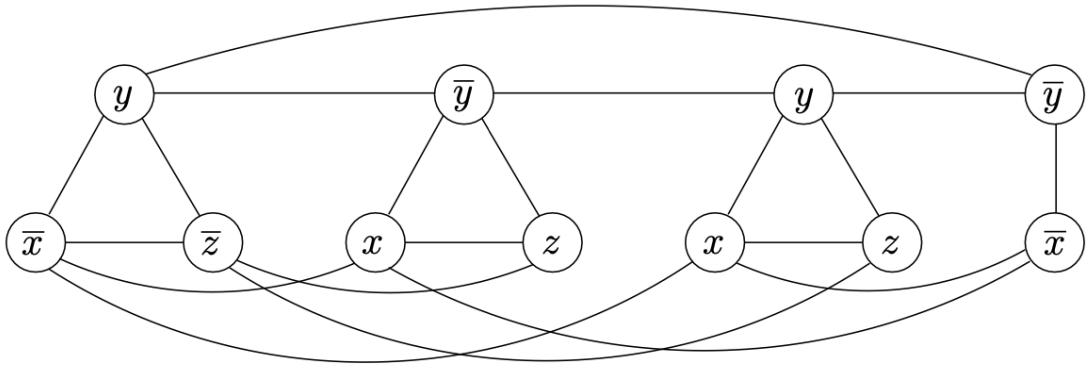
Reductions

3SAT⟶INDEPENDENT SET

One can hardly think of two more different problems. In 3SAT the input is a set of clauses, each with three or fewer literals, for example

$$(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y}),$$

Figure 8.8 The graph corresponding to $(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y})$.



Reductions

- SAT->3SAT
- This is a special, and common type of problem
  - Reducing from a general problem to a special case of itself
  - Note what that says about the relative difficulty of the special case

Reductions

- SAT->3SAT

$$(a_1 \vee a_2 \vee y_1)\,(\overline{y}_1 \vee a_3 \vee y_2)\,(\overline{y}_2 \vee a_4 \vee y_3)\,\cdots\,(\overline{y}_{k-3} \vee a_{k-1} \vee a_k),$$

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\,(\overline{y}_1 \vee a_3 \vee y_2)\,\cdots\,(\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Prove this iff

Reductions

- Independent set -> vertex cover
- State the problems
  - And remember what these are: a vertex cover, an independent set

Reductions

- Independent set -> clique
- State the problems
  - And remember what a clique is
  - Remember also that these problems have integer parameters

Maximum Bipartite Matching

- In the maximum bipartite matching problem, we are given a connected undirected graph with the following properties:
  - The vertices of G are partitioned into two sets, X and Y.
  - Every edge of G has one endpoint in X and the other endpoint in Y.
- Such a graph is called a **bipartite graph**.
- A **matching** in G is a set of edges that have no endpoints in common—such a set "pairs" up vertices in X with vertices in Y so that each vertex has at most one "partner" in the other set.
- The maximum bipartite matching problem is to find a matching with the greatest number of edges.

Maximum Bipartite Matching

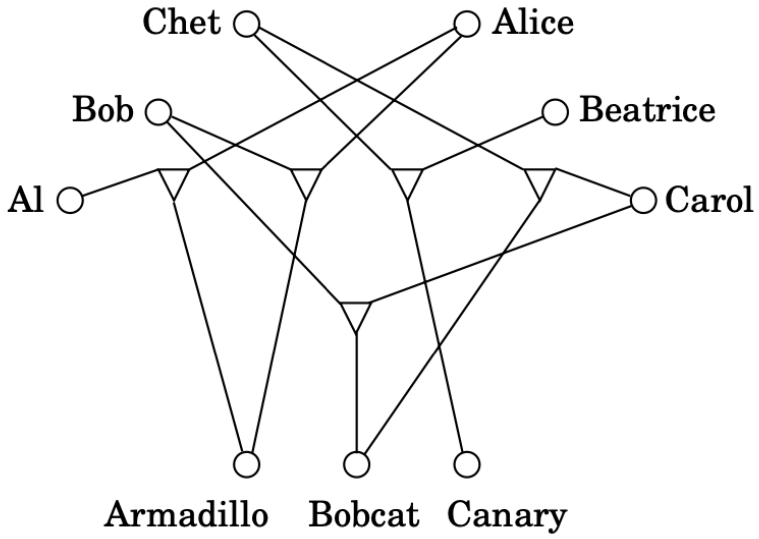
- This is an "easy" probem
- How do we solve it?

3D Matching Problem

- Now given n women, n men, and n pets!
- Compatibilities specified by triples
  - (woman, man, pet)
- Goal: find n *disjoint* triples
  - That is, triples such that each woman, each man, and each pet, are in exactly one triple
  - Note, there may be (and often are) many more than n triples

3D Matching

Figure 8.4 A more elaborate matchmaking scenario. Each triple is shown as a triangular-shaped node joining boy, girl, and pet.



It turns out that this is a hard problem

Reductions

- 3D Matching -> ZOE
- State the problems
  - And remember what a clique is
  - Remember also that these problems have integer parameters

ZOE

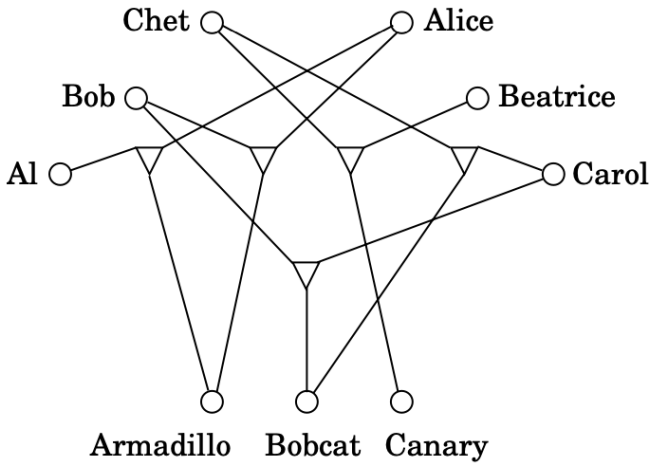
Recall that in ZOE we are given an $m \times n$ matrix $\mathbf{A}$ with $0-1$ entries, and we must find a $0-1$ vector $\mathbf{x} = (x_1, \ldots, x_n)$ such that the m equations

$$\mathbf{Ax} = \mathbf{1}$$

are satisfied, where by $\mathbf{1}$ we denote the column vector of all 1's. How can we express the 3D MATCHING problem in this framework?

Indeed, how can we? That's for you to figure out!

Reduction: 3D Matching->ZOE



$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

The five columns of **A** correspond to the five triples, while the nine rows are for Al, Bob, Chet, Alice, Beatrice, Carol, Armadillo, Bobcat, and Canary, respectively.

Aside: The Halting Problem

- What we want:
  - `terminates(p,x)`
    - p is file containing a program
    - x is a file containing the input
  - returns true or false, depending on whether program p terminates when run on input x

Halting Problem (cont.)

- Suppose we had such a program. We could use it as a subroutine for the following program:

```
function paradox(z:file)
1: if terminates(z,z) goto 1
```

Halting Problem (cont.)

- Note what **`paradox`** does
  - Terminates iff program z does not terminate given its own code as input

```
function paradox(z:file)
1: if terminates(z,z) goto 1
```

Halting Problem (cont.)

- Note what **`paradox`** does
  - Terminates iff program z does not terminate given its own code as input

```
function paradox(z:file)
1: if terminates(z,z) goto 1
```

What happens if we put this program in a file named **`paradox`** and execute **`paradox(paradox)`**?

Reductions

- ZOE-SUBSET SUM
- State the problems

Reductions

- ZOE-SUBSET SUM
- State the problems

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix},$$

Problem: CIRCUIT SAT

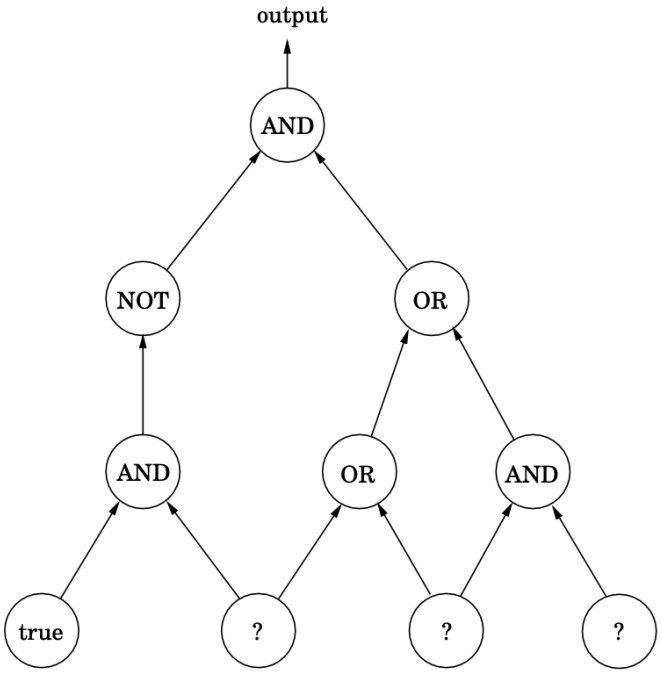
- Given a boolean circuit
- A DAG whose vertices are *gates* of 5 different types
  - AND gates and OR gates with indegree 2
  - NOT gates with indegree 1
  - *Known input* gates have no incoming edges and are labeled **`true`** or **`false`**
  - *Unknown input* gates have no incoming edges and are labeled "**`?`**"

Problem: CIRCUIT SAT

- One of the sinks of the DAG is labeled **`output`**
- Given an assignment of values to the unknown gates, we can evaluate the circuit in topological order
- The search: Given such a circuit, find an assignment for the unknown values such that the output is true, or report that no such assignment exists

CIRCUIT SAT

Figure 8.13 An instance of CIRCUIT SAT.



Reductions

- Any problem in NP -> SAT
- We do this by reducing any problem in NP->CIRCUIT SAT
- And CIRCUIT SAT -> SAT

Question

- SAT -> CIRCUIT SAT?

Reduction

- CIRCUIT SAT -> SAT
- What is the issue here?

Reduction

- CIRCUIT SAT -> SAT
- What is the issue here?