COMP225: Algorithms and Data Structures

Intractable Problems

Mark Dras
Mark.Dras@mq.edu.au
E6A380

Outline

- Definitions
- SAT
- Reductions
- Vertex Cover
- Miscellanea

What's the Point?

- In general, aim to find an efficient algorithm to solve a problem
- Therefore, useful to know if a particular problem has no efficient solution
  - means you don't waste time looking for a solution that doesn't exist
  - instead, you can concentrate on developing good heuristics or an approximate solution

What's a Problem?

- Examples

given: a weighted graph G

determine: the minimal spanning tree

given: a weighted graph G and integer k

determine: does there exist a path containing all vertices of cost $\leq$ k?

What's a Problem?

- We'll just be looking at decision problems (the second type)
- Most problems can be rephrased as decision problems
  - the second one is a version of the Travelling Salesman Problem

P & NP

- A problem is said to be polynomial (in class P) if it can be *solved* in time that is bounded by a polynomial in its size
  - all the sorting algorithms from this unit have been (bounded by *a*) polynomial
  - e.g. worst case complexity is $O(n^2)$ for bubble sort
    - that is, there's some polynomial $f(n) = an^2 + bn + c$ that gives the running time for the algorithm

P & NP

- P also includes things like O(n log n) for heapsort
  - not strictly polynomial, but bounded by some polynomial $an^2 + bn + c$
- What's not in P then?
  - exponential functions
  - factorial functions
  - ...

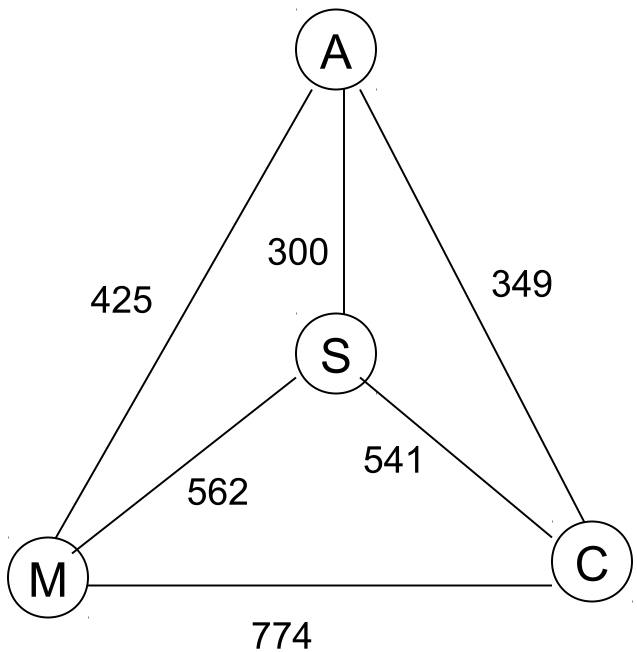
P & NP

- NP stands for nondeterministically polynomial
  - as a rough description, a problem in NP can have a proposed solution *verified* in polynomial time
  - all problems in P can have solutions verified in polynomial time
    - it is no harder to verify a solution than it is to calculate it
    - e.g. it can be verified in polynomial time whether an array is sorted (complexity?)

NP

- However, there are many problems where no polynomial algorithm is known for finding a solution, but where the solution can be checked in polynomial time
  - e.g. Travelling Salesman (TSP) decision problem

NP: TSP Decision



is there a TSP tour of less than 2000?

optimal: 1877 (A-C-S-M-A)

greedy: 2040 (A-S-C-M-A)

NP: TSP Decision

- Only way to find tour <k for some arbitrary k is to enumerate every possibility

a-c-s-m-a	a-c-m-s-a
a-m-c-s-a	a-m-s-c-a
a-s-c-m-a	a-s-m-c-a

- For a fully connected graph of n nodes, there are (n-1)! solutions
  – not polynomial
- Related problem: Hamiltonian circuit
  – no weights on edges: just see if there’s a circuit taking in all vertices

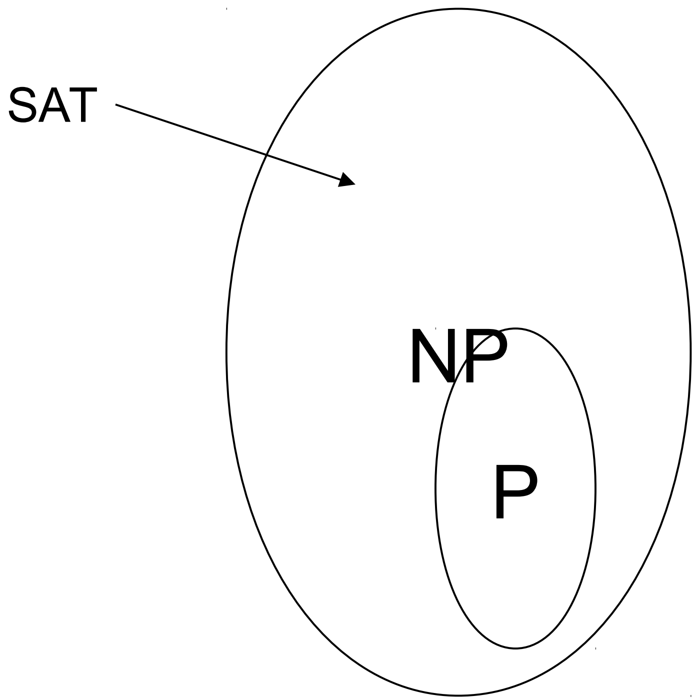
NP

- However, TSP decision problem can be verified in polynomial time
  - traverse the path of the proposed solution, add up cost, compare with k: O(n)
- General TSP problem (find optimal tour) follows from TSP decision:
  - can verify for some k
  - pick some range for k, carry out binary search until optimal size is determined

NP

- Aim
  - there are some problems where it is "almost certain" that no polynomial time algorithm exists to solve it
  - we'll look at one such problem, **satisfiability** (SAT)
  - then, to demonstrate that for other problems it's "almost certain" that no polynomial time algorithm exists, we'll show that the new problem is at least as hard as SAT
    - this process is called a reduction

P & NP



- Current big question in computer science: does P = NP?

Outline

- Definitions
- SAT
- Reductions
- Vertex Cover
- Miscellanea

SAT Definition

- Given: a set of clauses in conjunctive normal form
- Determine: is there a truth assignment to the Boolean variables such that every clause is simultaneously satisfied?

SAT Example

- Satisfy all of the following clauses:

$$x_1 \vee x_2 \vee x_3 \vee x_4$$

$$\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4$$

$$\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4$$

$$x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4$$

SAT Example

- Possible solution:
  - $x_1=1, x_2=0, x_3=1, x_4=1$
- Need to enumerate all possibilities (exhaustive search):
  - $x_1=0, x_2=0, x_3=0, x_4=0$
  - $x_1=1, x_2=0, x_3=0, x_4=0$
  - $x_1=0, x_2=1, x_3=0, x_4=0$
  - $x_1=1, x_2=1, x_3=0, x_4=0$
  - ...
- Enumerating 2^n possibilities

SAT: Exercise

- Satisfy all of the following clauses:

$x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$

$\neg x_1 \vee x_3 \vee \neg x_4 \vee \neg x_5$

$x_1 \vee \neg x_2 \vee x_3 \vee x_5$

$\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4 \vee \neg x_5$

$x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$

$\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4 \vee \neg x_5$

SAT

- Used in:
  - integrated circuit design
  - computer-aided design
  - computer networking
  - scheduling
  - robotics
  - machine vision
  - ...

SAT

- An obvious algorithm
  - try all combinations
  - for each combination, verify it
    - set n variables to 0 or 1, then OR them together
    - do this for m clauses
  - this algorithm will then be $O(mn2^n)$

SAT

// represent n variables by a class BinArray (basically, array of 0, 1)

```
public class BinArray {
  private int MAXSIZE;
  private int[] a;

  public BinArray(int n) {
    MAXSIZE = n;
    a = new int[MAXSIZE];
    this.init();
  }
  public void init() {
    for (int i = 0; i < a.length; i++)
    a[i] = 0;
  }
  public void print() {
    for (int i = 0; i < a.length; i++)
    System.out.print(a[i] + " ");
    System.out.println();
  }
```

SAT

```java
public void inc() {
  int i = 0;
  boolean carry = true;
  a[0]++;
  while (carry) {
    if (a[i] > 1) {
      a[i] -= 2;
      a[i+1]++;
    }
    else
      carry = false;
    i++;
  }
}

public boolean isMin() {
  for (int i = 0; i < a.length; i++)
    if (a[i] == 1)
      return false;
  return true;
}
```

```java
public boolean isMax() {
  for (int i = 0; i < a.lngth; i++)
    if (a[i] == 0)
      return false;
  return true;
}

public int get(int j) {
  return a[j];
}

}
```

SAT

// represent m constraints by a 2D array int constraints[m][n]
// constraints[i][j] can be 1 (when variable j exists in constraint i)
// 0 (when variable j is not in constraint i)
// -1 (when neg of variable j in constraint i)
e.g.

```
int[][] cons = {{1, -1, 1, -1},
                {0, 1, 1, -1},
                {-1, 0, -1, 1},
                {0, 1, 1, -1},
                {1, 1, 1, 0},
                {-1, 1, 1, 0},
                {1, 0, 1, 1},
                {-1, -1, 0, 1}};
```

SAT

```
public static boolean sat(int cons[][])
{
  boolean result;  // whether we've found a solution
  if (cons.length > 0) { // there are some constraints
    result = false;
    BinArray vars = new BinArray(cons[0].length);
    vars.init();
    result = verify(vars, cons);
    while (!result && !vars.isMax()) { // enumerate all candidate solutions
      vars.inc();
      result = verify(vars, cons);
    }
    vars.print();
  }
  else
    result = true;
  return result;
}
```

SAT

```java
public static boolean verify(BinArray vars, int cons[][])
{
  boolean indivResult;
  boolean result = true;
  int term;
  for (int i = 0; i < cons.length; i++) { // check all clauses
    indivResult = false;
    for (int j = 0; j < cons[i].length; j++) { // check single clause
      if (cons[i][j] == 0)
        term = 0;
      else if (cons[i][j] == 1)
        term = vars.get(j);
      else
        term = 1 - vars.get(j);
      indivResult = indivResult || (term == 1); // clause T if one term is T
    }
    result = result && indivResult; // result T if all clauses T
  }
  return result;
}
```

SAT and NP

- Is it known that there is no polynomial algorithm to solve SAT?
  - no
  - but lots of smart people have worked on the problem for a long time
  - and if SAT is in P (i.e. has a polynomial solution), lots of weird things would be true
- SAT is in NP though
  - a solution can be verified in polynomial time: O(mn)

NP-Completeness

- SAT is a fundamental problem in NP that's (very probably) not in P
  - it is strongly believed that the problem is inherently computationally intractable
- To show that another problem is inherently computationally intractable, show that it is at least as hard as SAT
  - this is **NP-hard**
  - **NP-complete** is if it is NP-hard and also in NP

Outline

- Definitions
- SAT
- Reductions
- Vertex Cover
- Miscellanea

Problem Reduction

- To show that a problem is at least as hard as SAT (or any other), you need to show that every instance of SAT can be transformed into an instance of this other problem
  - direction often seems counter-intuitive
- Reduction has to be by a polynomial process
- As an example, define a simpler version of SAT, and reduce SAT to that

3-SAT

- Like SAT, but each clause has exactly 3 literals
- To show that 3-SAT is computationally intractable, show that every instance of SAT can be written as 3-SAT
- Suppose a clause in SAT contains k literals

3-SAT

- If k = 1 (that is, the clause consists of a single literal), make up two new variables, v_1 and v_2
  - then, four new 3-literal clauses:

    $x_1 \vee v_1 \vee v_2$

    $x_1 \vee v_1 \vee \neg v_2$

    $x_1 \vee \neg v_1 \vee v_2$

    $x_1 \vee \neg v_1 \vee \neg v_2$
  - only way all can be true is if x_1 is true

3-SAT

- If k = 2 (that is, the clause consists of two literals), make up one new variables v_1
  - then, two new 3-literal clauses:

    $x_1 \vee x_2 \vee v_1$

    $x_1 \vee x_2 \vee \neg v_1$
  - only way both can be true is if x_1 or x_2 is true

3-SAT

- If k > 3, make up n-3 new variables, v_1, v_2, …, v_{n-3}
  - then, n-2 new 3-literal clauses:

    $x_1 \vee x_2 \vee \neg v_1$

    $v_1 \vee x_3 \vee \neg v_2$

    $v_2 \vee x_4 \vee \neg v_3$

    …

    $v_{n-3} \vee x_{n-1} \vee x_n$
  - same satisfiability as SAT

3-SAT

- Conclusion is that 3-SAT is at least as hard as SAT
  - therefore 3-SAT is NP-hard
- Also can obviously verify in polynomial time: O(m)
  - therefore NP-complete
- 3-SAT is used in a lot of proofs of computational intractability

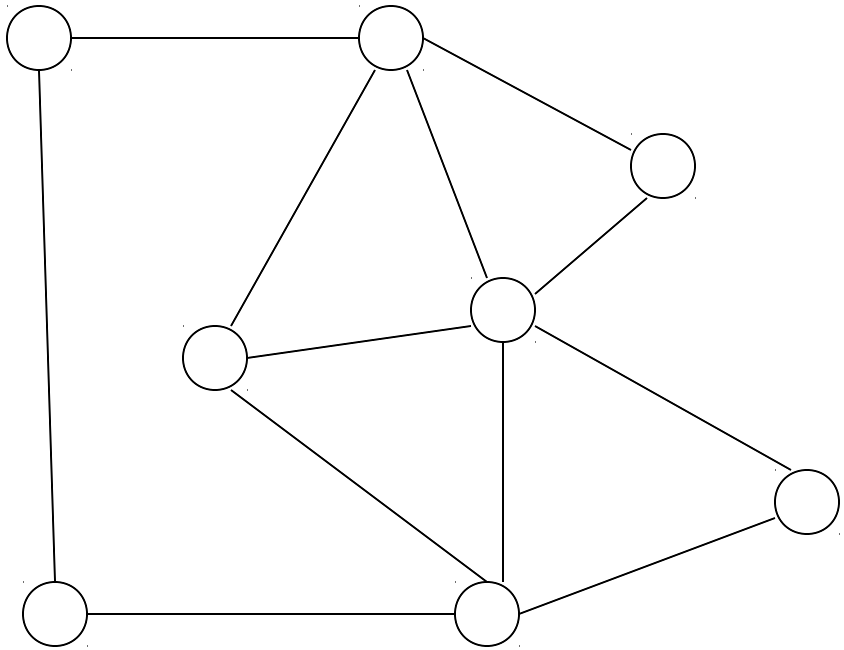
Outline

- Definitions
- SAT
- Reductions
- Vertex Cover
- Miscellanea

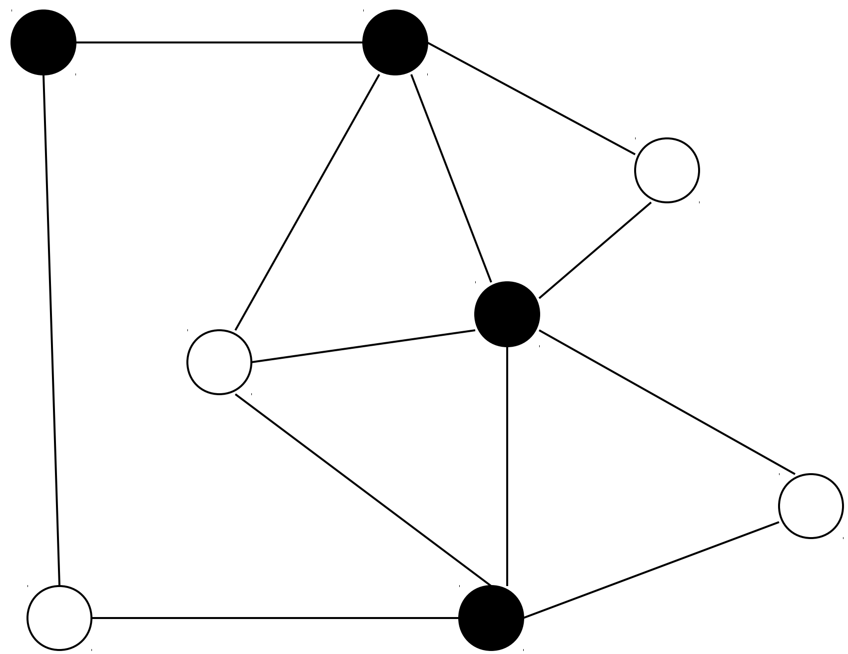
Vertex Cover

- Given: graph $G = (V,E)$ and integer $k \leq |V|$
- Determine: is there a subset S of at most k vertices such that every $e \in E$ has at least one vertex in S?
- Easy to find some vertex cover
  - just choose all vertices
  - hard to find the minimal one

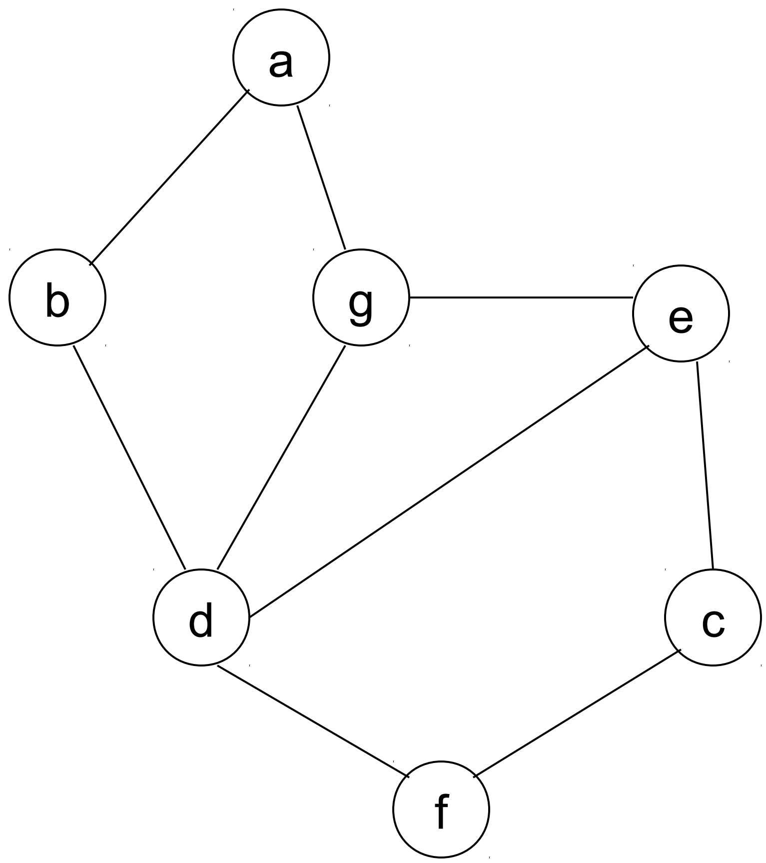
Vertex Cover



Vertex Cover



Vertex Cover: exercise



Vertex Cover

- Want to show that vertex cover is hard by a reduction of 3-SAT to vertex cover
  - show that every instance of variables satisfaction in 3-SAT corresponds to a vertex cover in some graph

Vertex Cover

- First, translate the variables in 3-SAT:
  - for each variable v_i, create two vertices v_i, v_i'
  - connect these by an edge
  - any vertex cover must have one of v_i, v_i'
- Second, translate the clauses
  - for each clause, create three new vertices (one corresponding to each variable)
  - connect these by edges to form triangles
  - any vertex cover must have two of these vertices

Vertex Cover

for clauses: $v_1 \vee \neg v_3 \vee \neg v_4$, $\neg v_1 \vee v_2 \vee \neg v_4$

stage 1

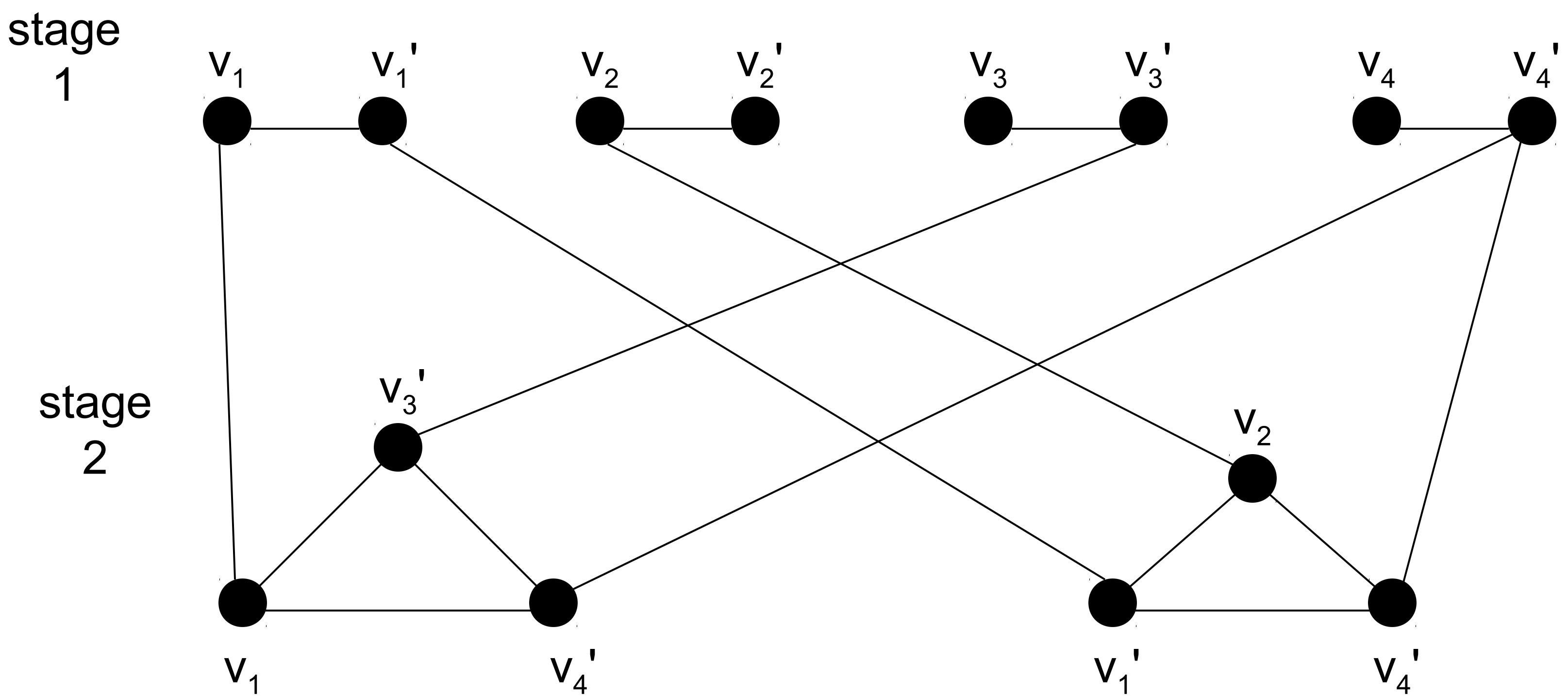
stage 2

Vertex Cover

- Finally, connect vertices that are labelled the same
- From a 3-SAT instance with n variables and c clauses, this gives a graph with 2n+3c vertices
  - has been specifically designed to have a vertex cover with n+2c nodes if and only if the original 3-SAT expression is satisfiable

Vertex Cover

for clauses: $v_1 \vee \neg v_3 \vee \neg v_4$, $\neg v_1 \vee v_2 \vee \neg v_4$



Vertex Cover

- Each solution to the 3-SAT needs a corresponding vertex cover
- A solution to the 3-SAT clauses is $v_1 = v_2 = T$, $v_3 = v_4 = F$
- To construct a vertex cover
  - in stage 1, choose the vertex corresponding to T/F variable values (i.e. v_1, v_2, v_3’, v_4’)
  - in stage 2, choose the vertices in the triangles **opposite** one selected in stage 1 (i.e. v_3’, v_4’ in triangle #1; v_1’, v_4’, in triangle #2)

Vertex Cover

- To complete reduction, need to show that
  - every satisfying truth assignment gives a vertex cover
  - every vertex cover gives a satisfying truth assignment
- Check for yourself
- Important: make sure you do the reduction in the correct direction!

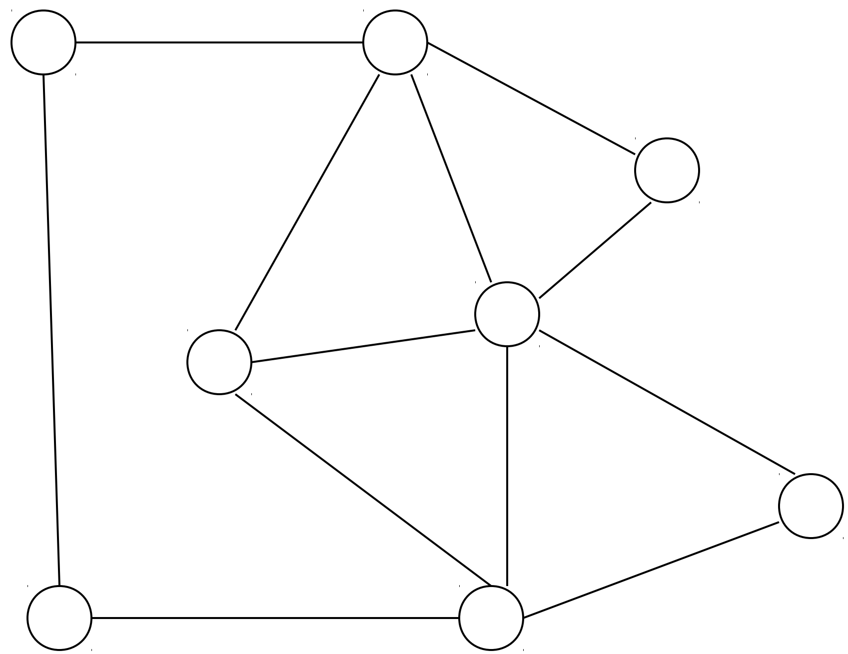
Reductions So Far

- Have shown that if SAT has no polynomial solution, then neither does 3-SAT
- Have shown that if 3-SAT has no polynomial solution, then neither does vertex cover
- Now show that if vertex cover has no polynomial solution, then neither does independent set

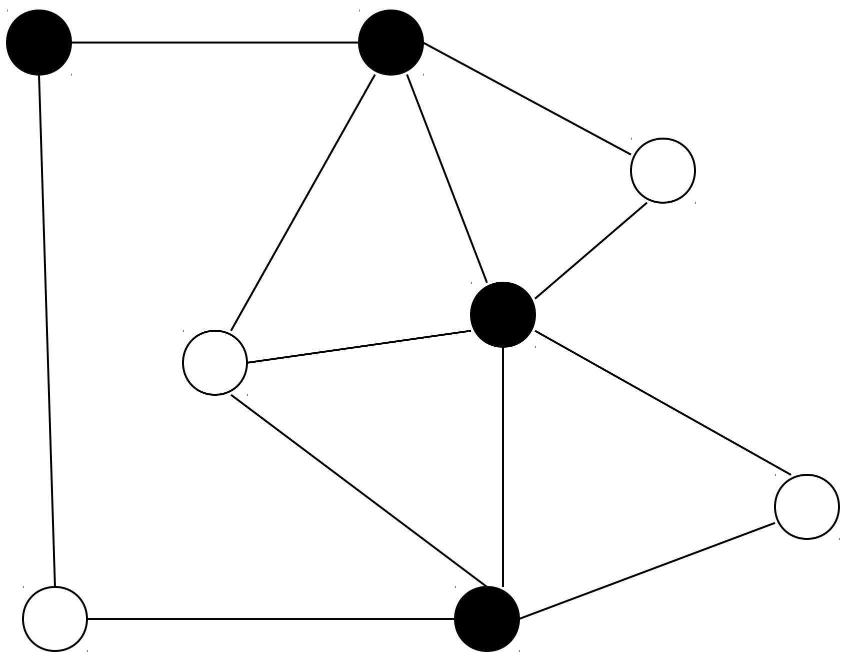
Independent Set

- A set of vertices S of a graph G is independent if there are no edges (x,y) where $x \in S$ and $y \in S$
  - i.e. vertices in the independent set are not connected
- Given: a graph G and integer $k \leq |V|$
- Determine: is there an independent set of k vertices in G

Independent Set



Vertex Cover



Independent Set

- The above definition is the decision problem version of the *maximum maximal independence* problem
  - find the largest independent set S for graph G
- Notice that this is just the complement of the vertex cover problem
- Reduction is much simpler than previously

Independent Set

- Convert every instance of vertex cover to independent set

```
VertexCover(G,k)
  G' = G
  k' = |V| - k
  return answer to IndependentSet(G',k')
```

- i.e. if you could solve independent set, you could solve vertex cover

Outline

- Definitions
- SAT
- Reductions
- Vertex Cover
- Miscellanea

The Aim of All This

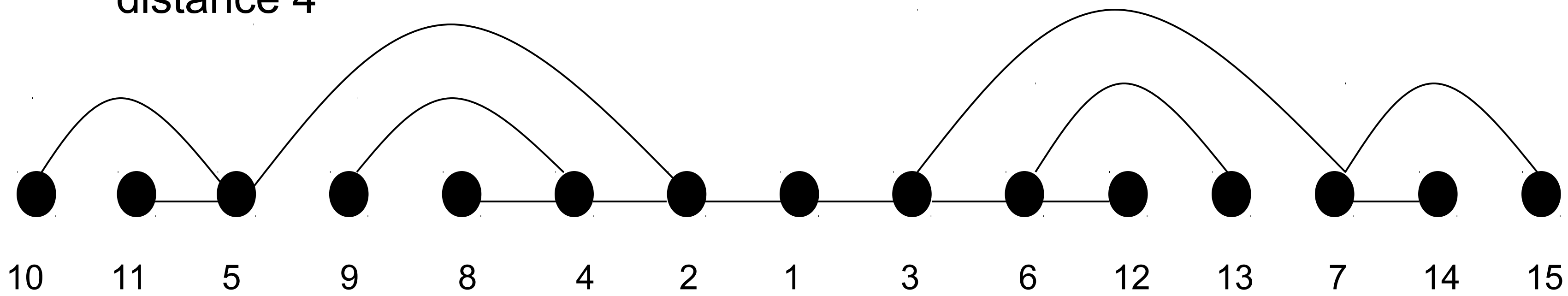
- If you're confronted with a new problem, want to work out whether there's an efficient algorithm
  - most important thing is to recognise whether it feels like an NP-complete problem, work out which one (as in the reduction process), then look it up in a catalogue

New Problem

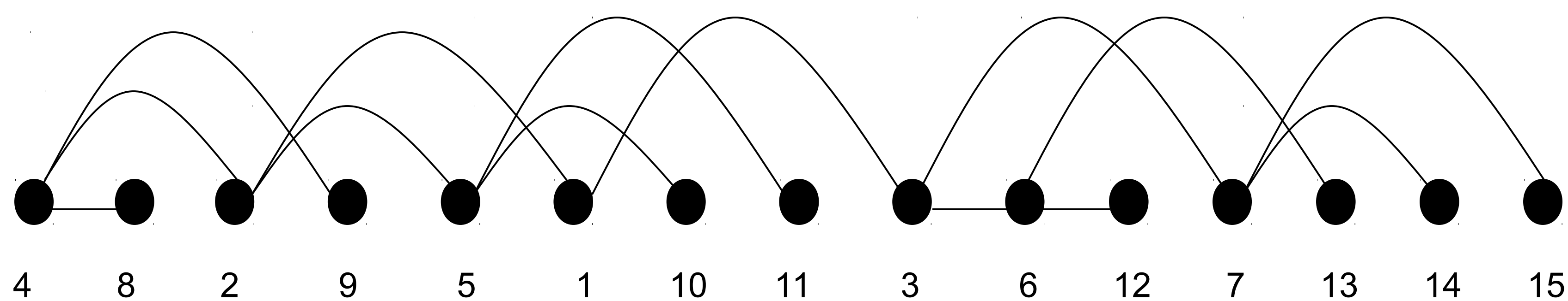
- Say you have a hypertext application
  - need to store large objects (like images) on some linear storage device
  - from each image there is a set of possible images that can be visited next (i.e. the hyperlinks)
  - to minimise search time, want to place linked images next to each other on the device, or as close as possible
    - minimise length of longest connected distance

New Problem

max
distance 4



max
distance 3



New Problem

- The structure looks like previous ones
  - what we want to find is a minimal maximum
  - involves a permutation
  - suggests maybe it's NP-complete
- Is it?

What to Do about NP-Complete Problems

- Know that problem has no polynomial time algorithm in worst case
- Three solutions
  - find algorithms fast in the average case
  - use heuristic methods (e.g. greedy algorithms, genetic algorithms)
    - no guarantee about solution at all
  - use approximation algorithms
    - give guarantee, but guaranteed to be sub-optimal

Approximating Vertex Cover

- Simple algorithm

```
VertexCover(G=(V,E))
   while (E≠∅)
      select an arbitrary edge (u,v)∈E
      add both u and v to the vertex cover
      delete all edges from E that are incident
            on either u or v
```

Approximating Vertex Cover

- Guaranteed to produce a cover at most twice the size of the optimal cover
- Cf. greedy algorithm
  - always select and delete vertex with highest remaining degree
  - can go astray giving a cover O(log n) times the optimal

Feasible Solutions

- Combine heuristics with approximate algorithms
  - like diversification of investments: stock market and bank account
  - run one heuristic and one approximate
    - approximate gives guaranteed lower bound in case heuristic produces bad solution
- Preprocessing good too