NP PROBLEMS, BACKTRACKING AND APPROXIMATE ALGORITHMS

DECIDABILITY

Consider a decision problem whose answer is yes or no

Problems for which there exist algorithms to find the answer **are decidable**

Problems for which no algorithms exist to find the answer **are undecidable**

What is an example of an undecidable problem?

- **Halting Problem**

TRACTABLE

Decidable problems that can be solved in polynomial time; $T(n)= O(n^k)$ are tractable

Others that take more than polynomial time are intractable

EXPRESSING AS DECISION PROBLEMS

We can express many problems, particularly optimization ones as decision problems

Instead of saying find the minimum value, we rephrase it as is there a solution whose size is less than equal to n

CIRCUIT-SAT

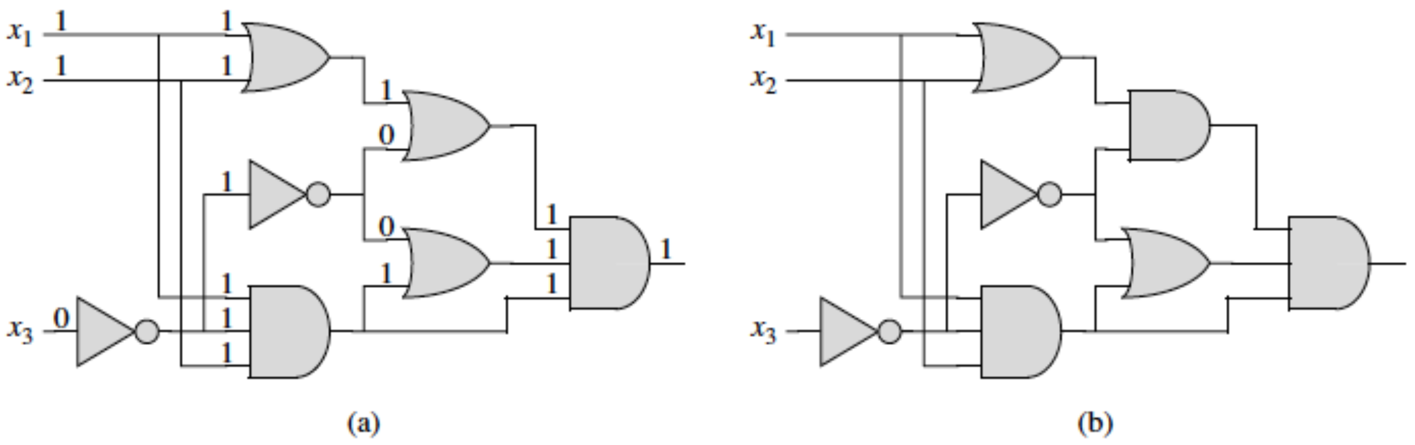


Figure 34.8 Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

SOLVING CIRCUIT-SAT

Two steps

- 1. Guess the answer
- 2. Verify the answer

For all possible combination of the inputs, check whether the answer satisfies the circuit.

Checking the answer is polynomial time.

How many possible combinations to check ?

CHARACTERISTICS OF NP PROBLEMS

Verification can be done in polynomial time

Number of instances to check is larger than polynomial

If we could check all the instances non-deterministically, then problem is polynomial

NP=Non-deterministic Polynomial

NP PROBLEMS

The upper bound is larger than polynomial time

Is the lower bound polynomial ?

If yes then we can show P=NP

Answer is no known yet!

NP COMPLETE PROBLEM

Consider a set of NP problems : S

Let x is an element (a problem) in S

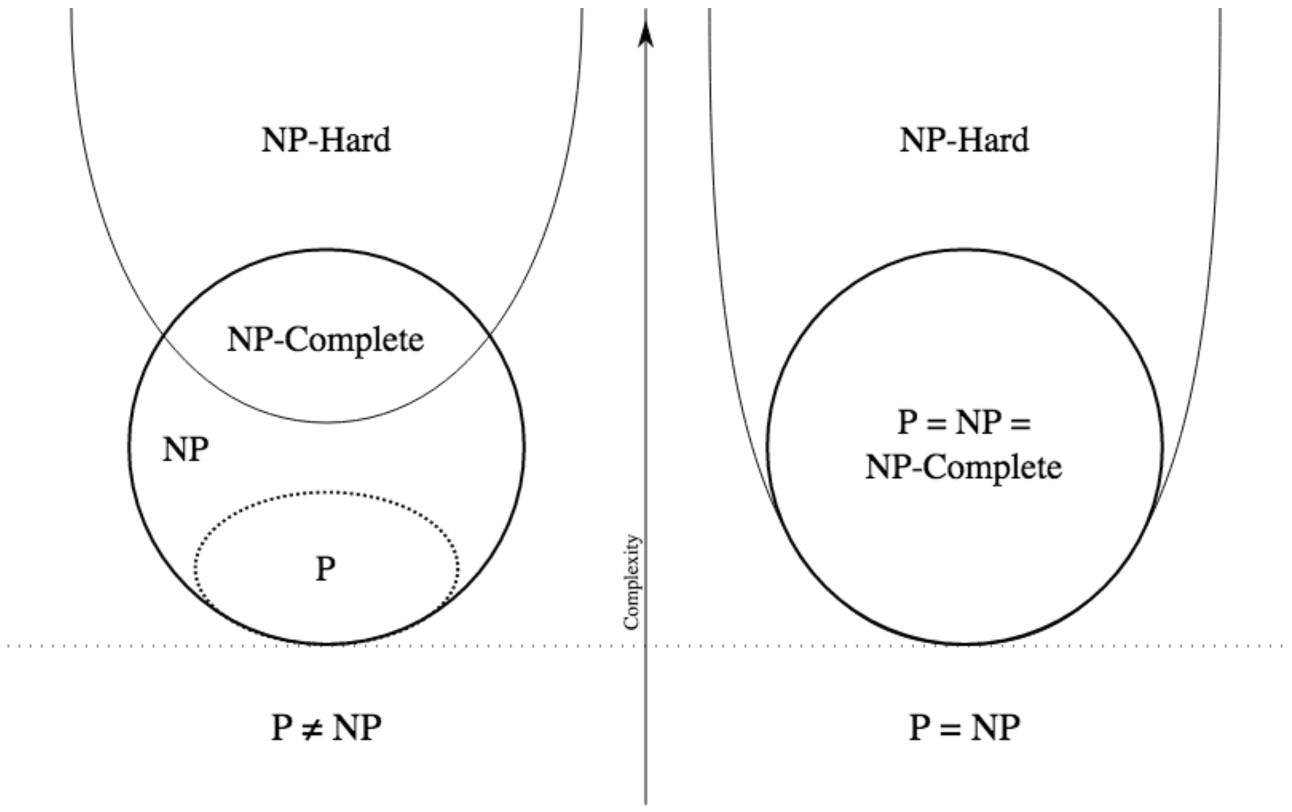
There is a polynomial time algorithm to transform any element in S to x

Then the problems in set S are NP-complete

So if we can solve one problem in the NP-complete set in polynomial time, we can solve all problems in the NP-complete set in polynomial time.

NP, NP-COMPLETE, NP-HARD

NP-hard: The problem is at least as hard as NP
Every problem in NP can be reduced to a problem in NP-hard set



NP-complete Problems

$$\begin{tikzcd}
\text{Stephen Cook 1971} & & & & \text{SAT} \arrow[d] & \\
\text{Richard Karp 1972} & & & & \text{3SAT} \arrow[dll] \arrow[dr] & \\
 & & \text{Independent Set} \arrow[dl] \arrow[dr] & & & \text{Hamiltonian Cycle} \arrow[d] \\
 & \text{Vertex Cover} \arrow[d] & & \text{CLIQUE} & & \text{Traveling-Salesman Problem(TSP)} \\
 & \text{Subset-Sum} & & & & \text{......} \\
 & \text{......} & & & &
\end{tikzcd}$$

About 1000 NP-complete problems have been discovered since.

SAT=>3 SAT

Circuit Satisfiability (SAT):

Can be written as set of Boolean Clauses of any length

- (x1 OR x2) AND (x3 OR x4 OR x5) AND (x6 OR x7 OR x8 OR x9)

3SAT:

All Clauses should be of length 3

SAT=> 3SAT

Convert clauses of size < 3

- (x1OR x2) => (x1 OR x1 OR x2)

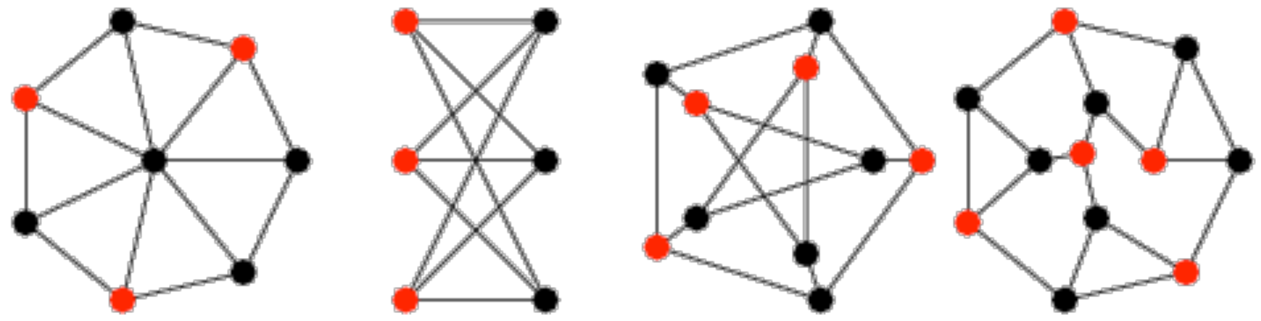
Convert clauses of size >3

- (x6 OR x7 OR x8 OR x9)
  - => (x6 Or x7 OR z1) AND (!z1 OR x8 OR x9)
- (x1 OR x2 OR x3 …. OR xk)
  - => (x1 OR x2 OR z1) AND (!z1 OR x3 OR z2) AND … (!zk OR x(k-1) OR xk)

INDEPENDENT SET

An independent set in a graph is a set of vertices no two of which are adjacent to each other

Does a given graph have an independent set of size >= K



3 SAT TO INDEPENDENT SET

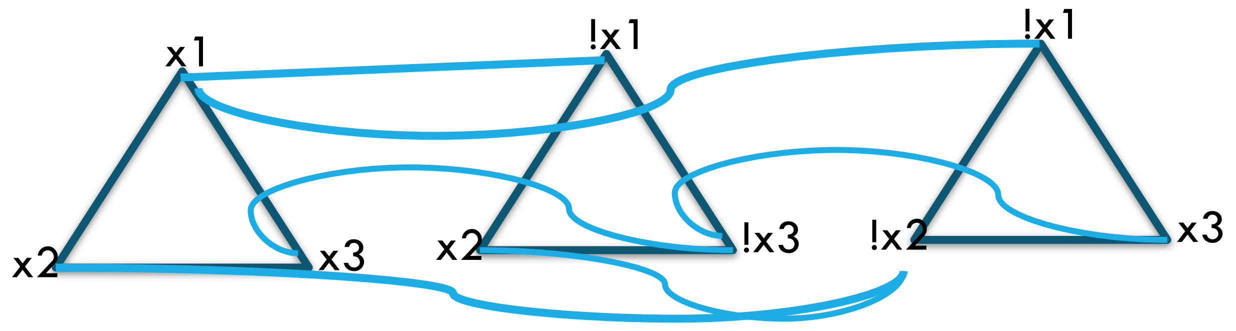
3SAT instance with K clauses

- (x1 OR x2 OR x3) AND (!x1 OR x2 OR !x3) AND (!x1 OR !x2 OR x3)

Create a triangle for each clause

Across triangles connect to negations of the same variables

The graph will have an independent set of size K if the given 3SAT formula is satisfiable



INDEPENDENT SET TO MAX CLIQUE

Is there a clique of size >=K in a graph

Take a graph

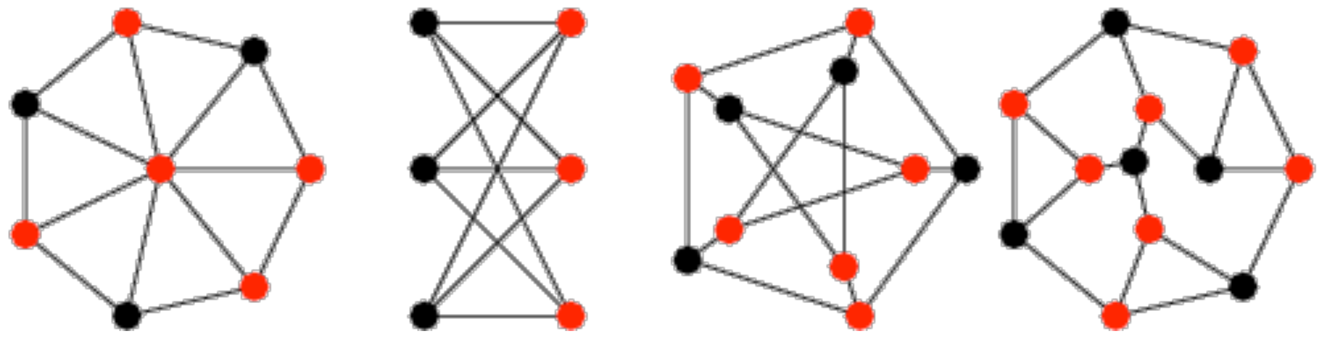
Take the complement of the graph (connect vertices that were not connected; remove connections between vertices that were connected)

Find the independent set of the complement graph

The vertices that form the independent set in the complement graph form the clique in the original graph

Independent set in the original graph=max clique in the complement graph

VERTEX COVER



A *vertex cover* of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it.

If K is an independent set in G (V,E), then V-K is a vertex cover in G’

MAX CLIQUE TO VERTEX COVER

Given a graph G, with V vertices, let the max clique be formed of the set of vertices C

Now create the complement of the graph, say G'.

Any edge in G' will have at least one end point not in C.

Thus the size of the independent set is at least |V|-|C|

APPROXIMATION ALGORITHMS

We can find a close near-optimal solution to many NP-complete problems with polynomial time algorithms.

Approximation algorithms provide performance ratios of the obtained solution (C) to the optimal solution (C*)

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) .$$

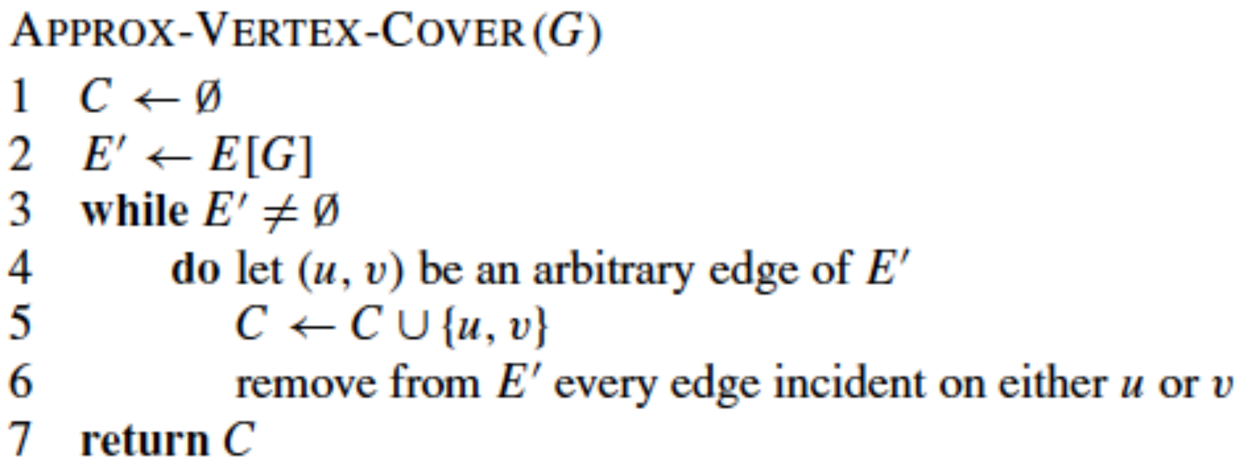
APPROXIMATION ALGORITHMS VS HEURISTICS

Approximation algorithms provide an approximation ratio of how close the solution is

Heuristics do not provide any such ratio

Not all NP-complete problems can be approximated within a polynomial factor—e.g. Independent Set problem

APPROXIMATION ALGORITHM FOR VERTEX COVER



```
APPROX-VERTEX-COVER($G$)
$C \leftarrow \emptyset$
$E' \leftarrow E[G]$
while $E' \neq \emptyset$
        do let $(u, v)$ be an arbitrary edge of $E'$
            $C \leftarrow C \cup \{u, v\}$
            remove from $E'$ every edge incident on either $u$ or $v$
return $C$
```

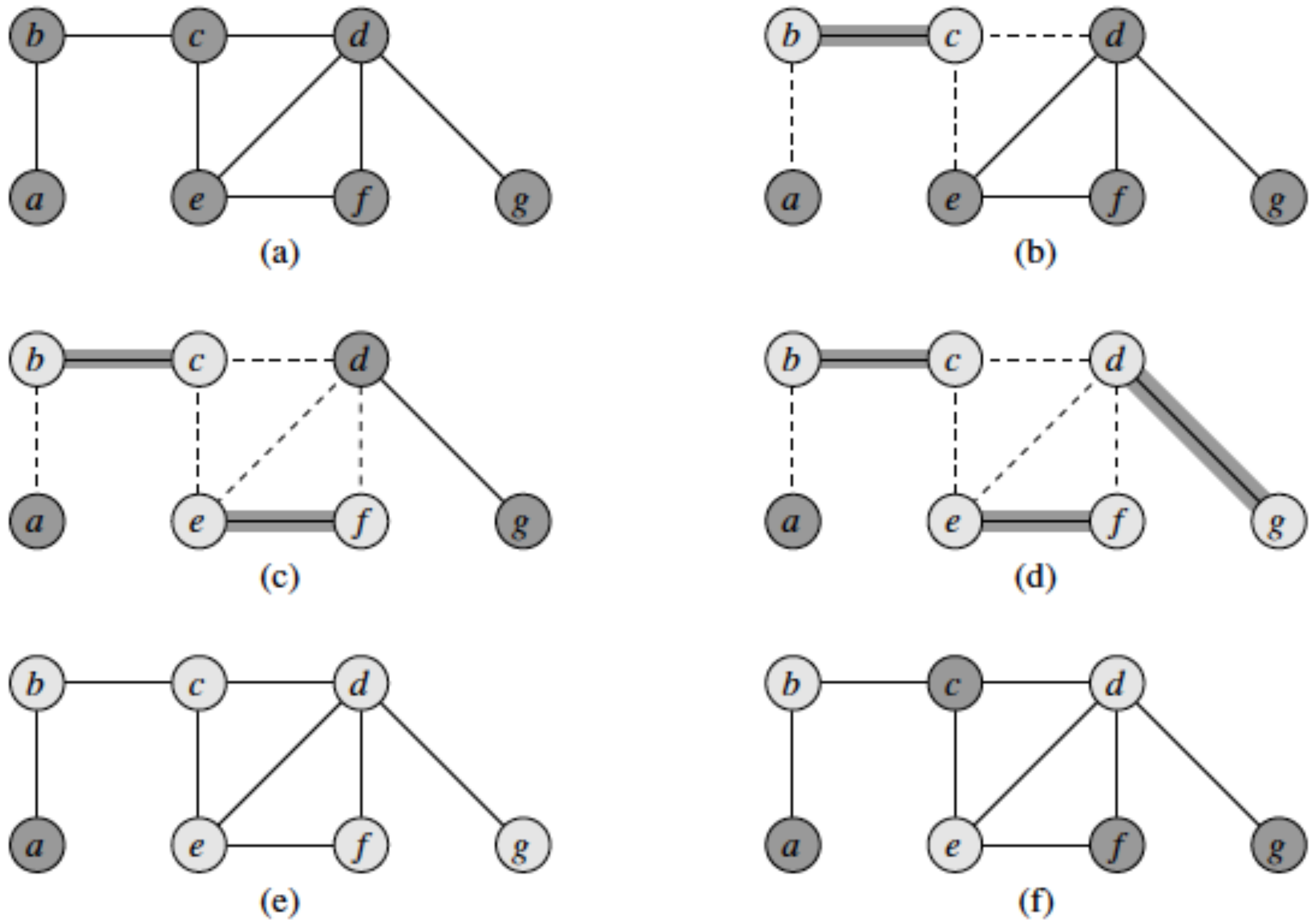


Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C. **(c)** Edge (e, f) is chosen; vertices e and f are added to C. **(d)** Edge (d, g) is chosen; vertices d and g are added to C. **(e)** The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g. **(f)** The optimal vertex cover for this problem contains only three vertices: b, d, and e.

APPROXIMATION FACTOR

Vertex cover is a polynomial time 2-approximation algorithm

Let A be the set of edges selected by the algorithm

Then no two edges in A share an end-point

The optimal solution C* includes at least one end point to each edge

Therefore no two edges in A are covered by the same vertex in C*; |C*| >= |A|

In the algorithm we pick the vertex cover C, by taking the two end points of A; |C|=2|A|

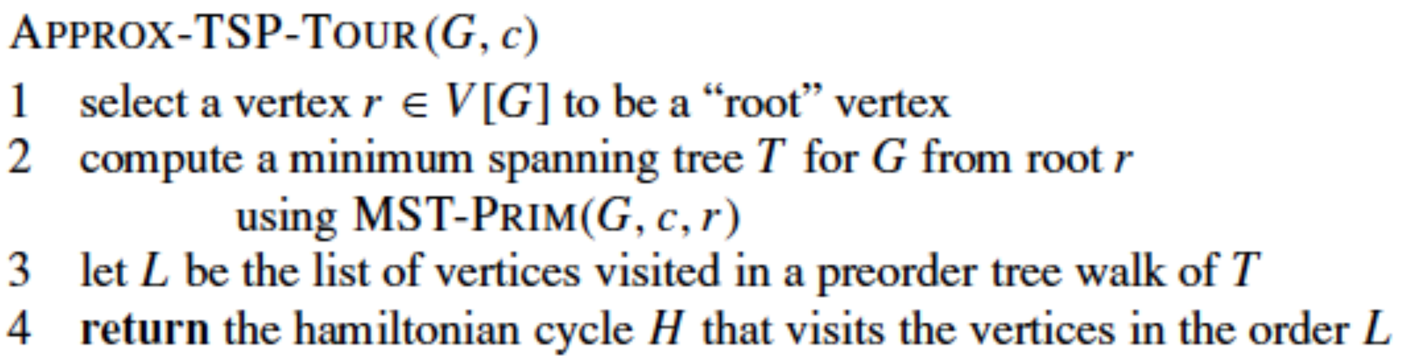
Together |C| <= 2|C*|

TRAVELLING SALESMAN PROBLEM

complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$, and we must find a hamiltonian cycle (a tour) of G with minimum cost. As an extension of our notation, let $c(A)$ denote the total cost of the edges in the subset $A \subseteq E$: $$c(A) = \sum_{(u,v)\in A} c(u, v) .$$

We assume triangle inequality, i.e. $$c(u, w) \le c(u, v) + c(v, w) .$$

TRAVELLING SALESMEN PROBLEM WITH TRIANGLE INEQUALITY



```
APPROX-TSP-TOUR$(G, c)$
1  select a vertex $r \in V[G]$ to be a "root" vertex
2  compute a minimum spanning tree $T$ for $G$ from root $r$
           using MST-PRIM$(G, c, r)$
3  let $L$ be the list of vertices visited in a preorder tree walk of $T$
4  **return** the hamiltonian cycle $H$ that visits the vertices in the order $L$
```

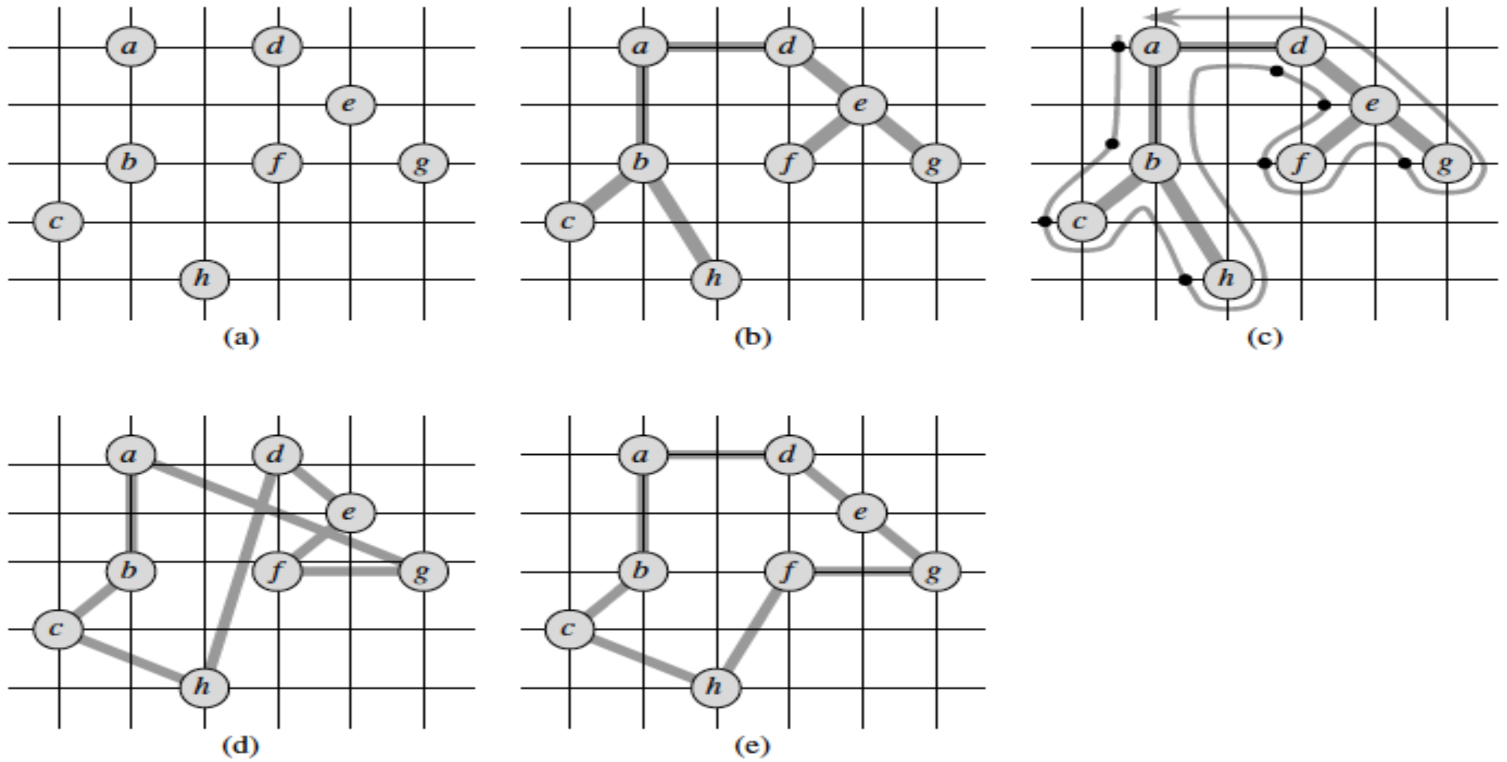


Figure 35.2 The operation of APPROX-TSP-TOUR. **(a)** The given set of points, which lie on vertices of an integer grid. For example, f is one unit to the right and two units up from h. The ordinary euclidean distance is used as the cost function between two points. **(b)** A minimum spanning tree T of these points, as computed by MST-PRIM. Vertex a is the root vertex. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. **(c)** A walk of T, starting at a. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g. **(d)** A tour of the vertices obtained by visiting the vertices in the order given by the preorder walk. This is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. **(e)** An optimal tour H^* for the given set of vertices. Its total cost is approximately 14.715.

APPROXIMATION FACTOR

The given algorithm is a ploynomial time 2-approximation algorithm for TSP with triangle inequality.

Let H* be the optimal cycle

A minimum spanning tree is an acyclic graph with total least edge cost.
Let the cost of MST be T c(T) <=c(H*)

A full walk, W, of T visits every edge in T twice, c(W)=2c(T)

Together c(W) <= 2c(H*)

BACKTRACKING

Backtracking is a systematic method for generating *all (or subsets of) combinatorial objects*.

- *Each object is a solution*
- *Constructing & traversing a search tree*

If the number of solutions expand combinatorially, then we can recursively branch to alternative solutions to find the ones that fulfill the constraints

BACKTRACKING ALGORITHM

The algorithm will generate all valid arrays X[1:N] whose elements come from domain S= {a_1,a_2,...,a_m} of successive integers, such that the constraints C are satisfied.

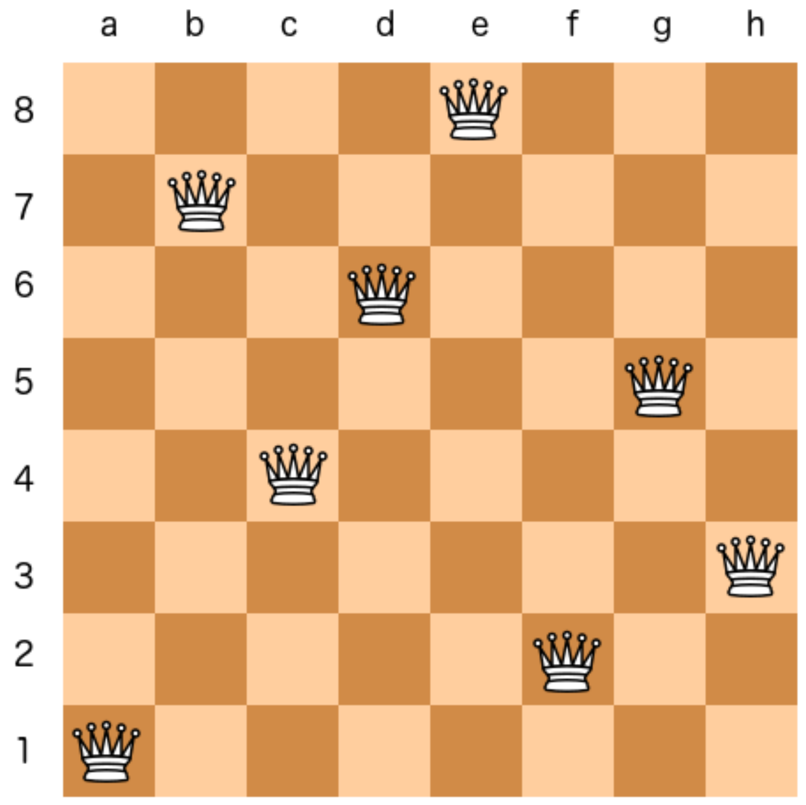
The algorithm is generates a *tree* that represents the entire *solution space*.

In the tree, the *root* designates the starting point, and every *path* from the root to a leaf is of length N, where the i-th node specifies a value for element X[i]. The whole path represents a single solution, that is, a single object.

- During tree generation, when we are to create a new node corresponding to X[i], we try to assign X[i] a new *next domain value* (given the current value of X[i] as reference).
- If that value does not violate the *constraints* C, it is assigned.
- If that value violates C, the next value after that will be tried, and so on, until either a C-compliant value is found or all remaining values are exhausted.
- If a C-compliant value is found and assigned to X[i], we move to the next level for *X[i+1]*.
- If no C-compliant value is found for X[i], we *backtrack* to the previous level for *X[i-1].*
- When we backtrack to the root, the whole tree has been searched, and the algorithm stops.

N QUEENS PROBLEM

Given an NxN chessboard and N queens. Can you place the Queens on the board such that they do not attack each other.

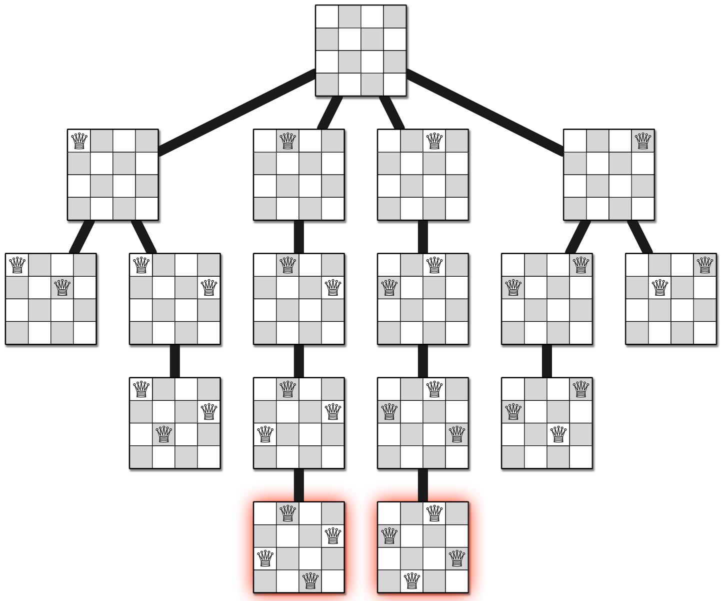


N QUEENS PROBLEM

Add Queen to first free spot.

If spot is not free move to the next spot

Continue until spot is selected



```
PLACEQUEENS($Q[1..n], r$):
  if $r = n + 1$
      print $Q[1..n]$
  else
      for $j \leftarrow 1$ to $n$
          $legal \leftarrow$ TRUE
          for $i \leftarrow 1$ to $r - 1$
              if $(Q[i] = j)$ or $(Q[i] = j + r - i)$ or $(Q[i] = j - r + i)$
                  $legal \leftarrow$ FALSE
          if $legal$
              $Q[r] \leftarrow j$
              PLACEQUEENS($Q[1..n], r + 1$)        ⟨⟨Recursion!⟩⟩
```

LONGEST INCREASING SUBSEQUENCE

Given a set of numbers, find the longest subsequence such that the numbers are in increasing order.

Example ={3, 10,2,1,20,15}

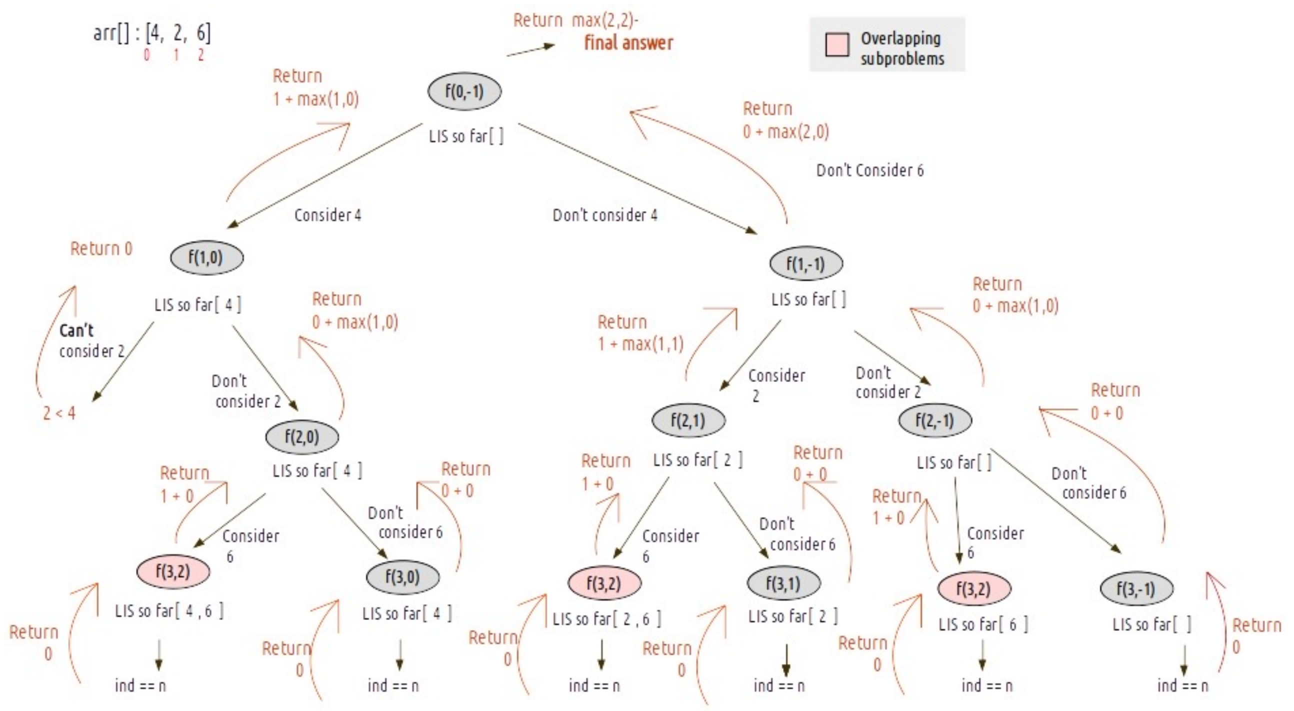
Longest increasing subsequence is: {3,10,20} or {2,20,15} or {1,20,15}

LONGEST INCREASING SUBSEQUENCE

Branching solution

From root create branch with each element.

Build the tree for each possible candidate



Does this remind you of something ?

GRAPH COLORING

Assign a color to each vertex of a graph, such that two adjacent vertices do not have the same color (vertex coloring)

The minimum number of colors with which the vertices of a graph can be colored is called the chromatic number of the graph, χ(G).

Graph coloring is used for resources allocation

- Least number registers in computing
- Fewest rooms to be allocated
- Fewest channels in mobile transmission

GRAPH COLORING IS NP-COMPLETE

Reduce from 3-SAT coloring to prove for 3 colors.

Create a triangle of 3 colors T,F,B

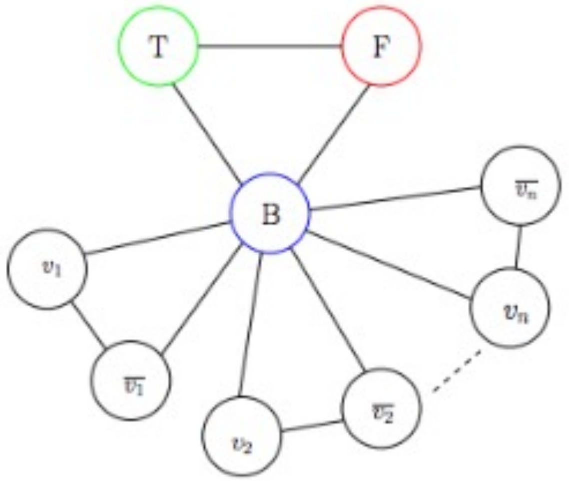
Connect every variable and its inverse with B

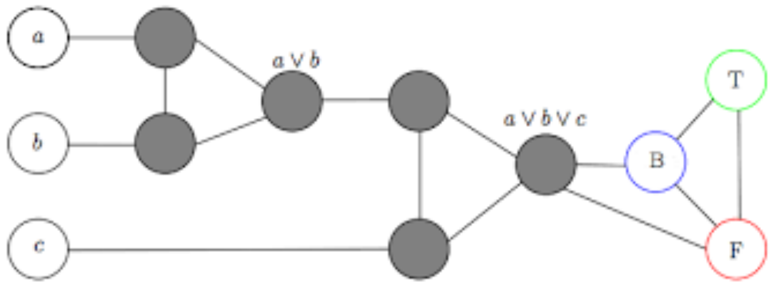
- If v1 is T, then v1' is F. Thus this is the truth assignment of the variables

For every clause create the gadget graph and connect final node to B and F.

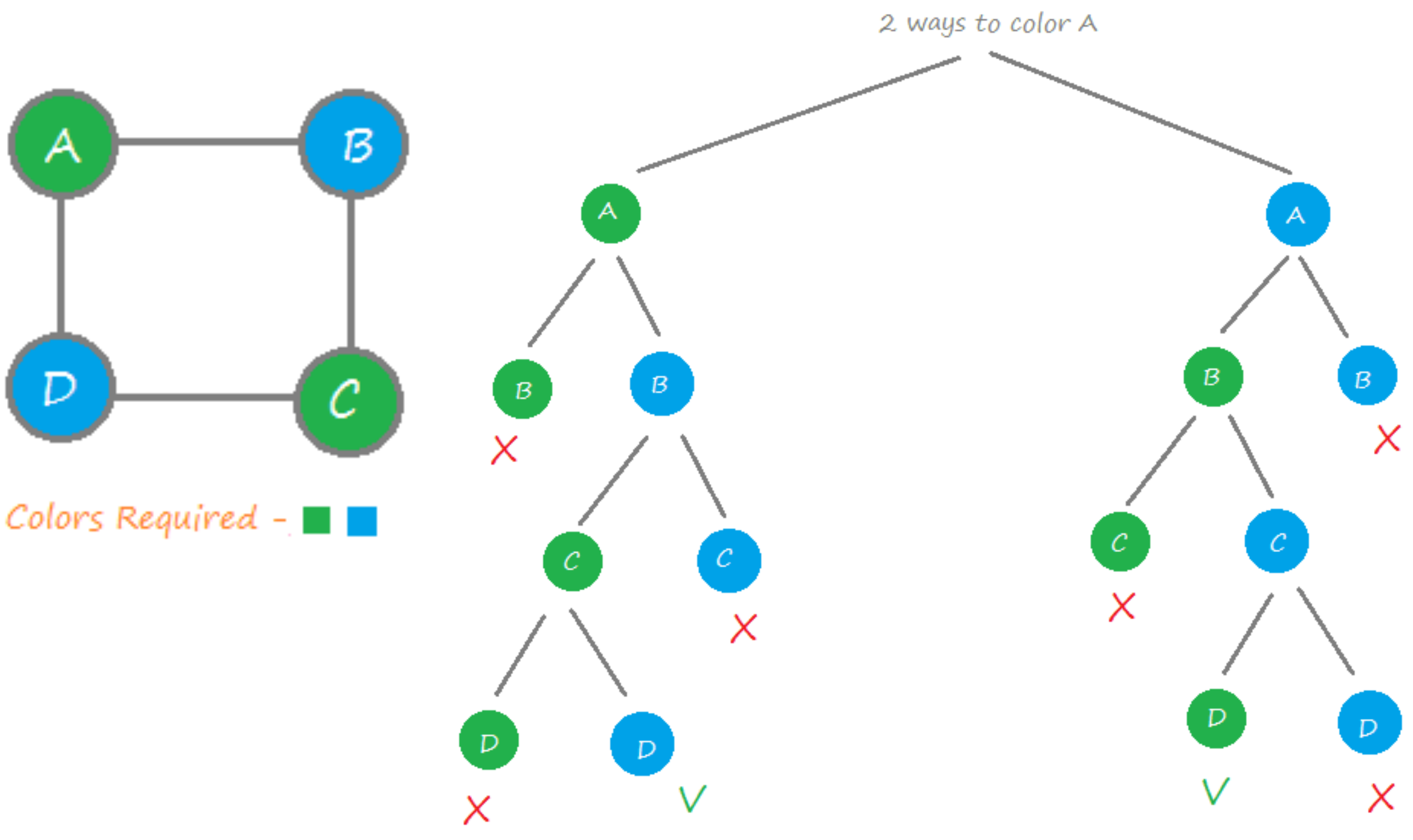
- This means that the final output has to be true

If all the clauses are true, then this graph is 3-colorable





GRAPH COLORING USING BACKTRACKING


A
B
D
C
Colors Required - 🟩 🟦
2 ways to color A
A
A
B
B
B
B
C
C
C
C
D
D
D
D
X
V