Chapter 6

Coping with the Hardness

Coping with the Hardness

Approximation algorithms

ILP Formulation

Backtrack

Branch and Bound

Coping with the Hardness

Approximation algorithms

ILP Formulation

Backtrack

Branch and Bound

In this chapter!

ILP Formulation

LP/ILP Formulation

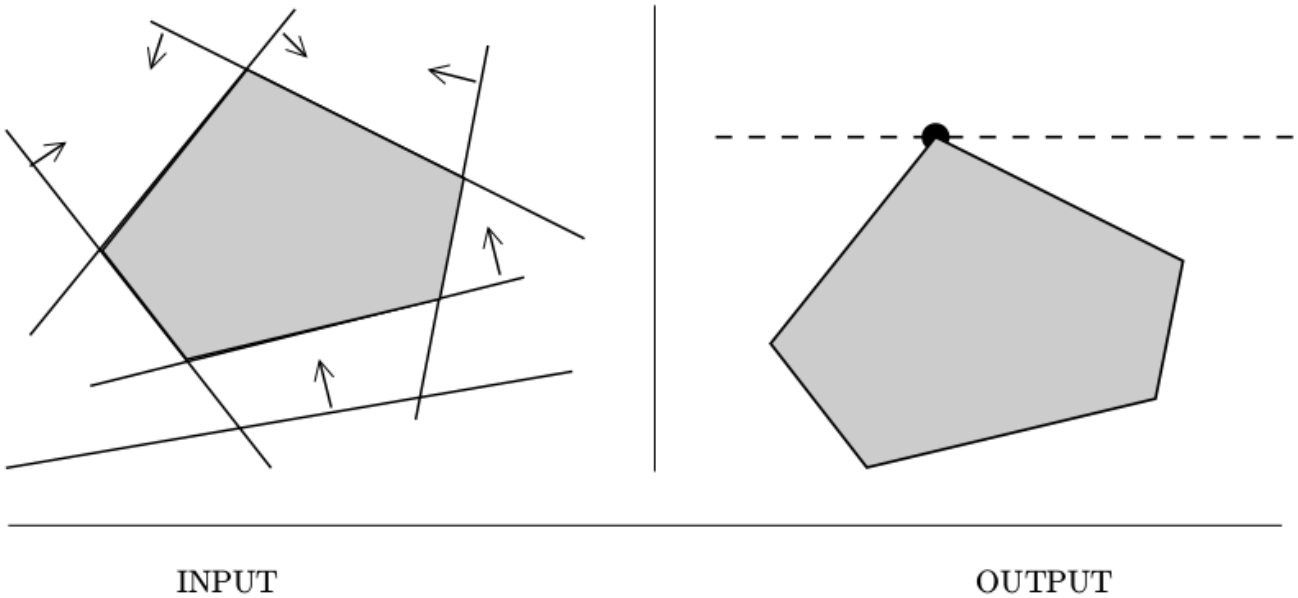
Linear programming (LP)– Optimization problems with linear objective and constraints

Integer linear programming (ILP)– An LP with integer variables

One method to deal with many (hard) problems is to convert them to an LP or ILP.

- There are many efficient LP and ILP solvers
- **ILP is NPC**. So we can convert all NP problems to ILP and enjoy ILP solvers.
  - Converting to ILP in many casses is easier from converting to SAT and using SAT-solvers

Linear Programming (LP)



Input description: A set S of n linear inequalities on m variables

$$S_i \ := \ \sum_{j=1}^{m} c_{ij}.x_j \geq b_i, \quad 1 \leq i \leq n$$

and a linear optimization function $f(X) = \sum_{j=1}^{m} c_j.x_j$

Problem description: Which variable assignment X' maximizes the objective function f while satisfying all inequalities S?

Example:

$$\begin{aligned} \text{maximize} \quad & 3x + 5y \\ \text{subject to} \quad & x + \ \ y \leq 4 \\ & x + 3y \leq 6 \\ & x \geq 0, \quad y \geq 0. \end{aligned}$$

The optimal solution is $x = 3, y = 1$

Motivation: Linear programming is **the most important problem in mathematical optimization and operations research**. Applications include:

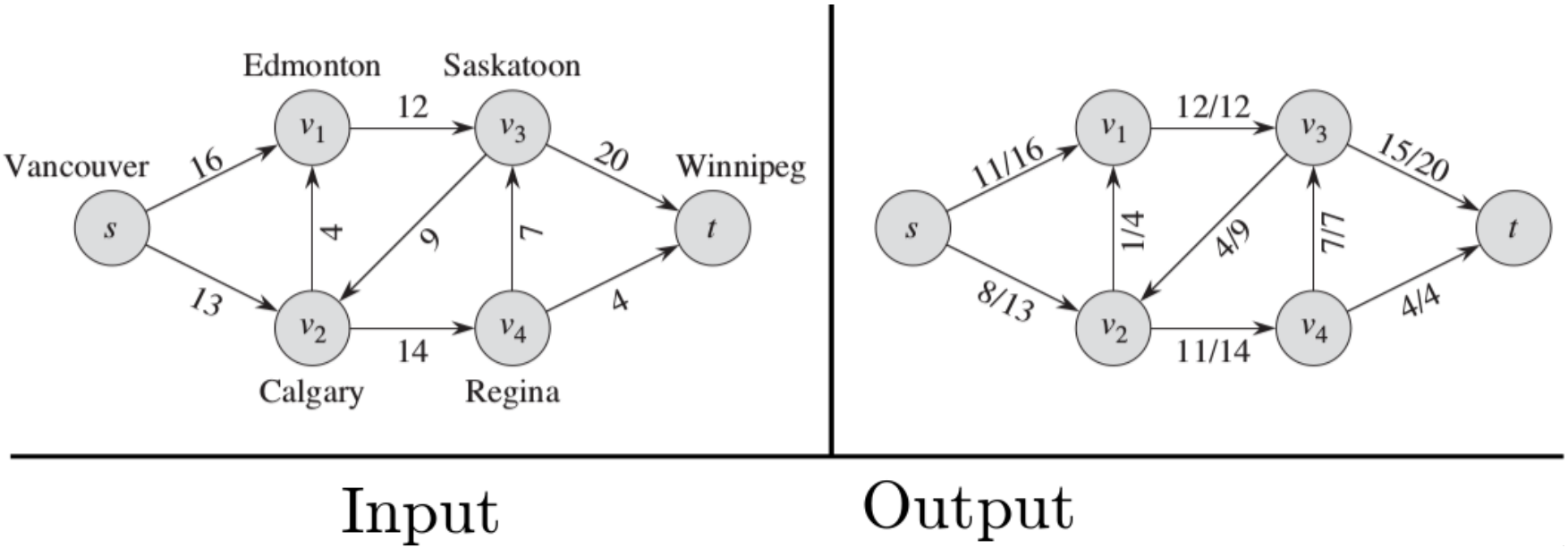
- *Resource allocation* – We seek to invest a given amount of money to maximize our return. Often our possible options, payoffs, and expenses can be expressed as a system of linear inequalities such that we seek to maximize our possible profit given the constraints. Very large linear programming problems are routinely solved by airlines and other corporations.
- *Graph algorithms* – Many of the graph problems, including shortest path, bipartite matching, and network flow, can be solved as special cases of linear programming. Most of the rest, including traveling salesman, set cover, and knapsack, can be solved using *integer linear programming*.

LP Solvers:

- Algorithms
  - *Simplex* method: Exponential worst case complexity but usually efficient
  - *Interior point* method: Polynomial complexity
- Both free and commercial codes (visit lpsolve.sourceforge.net)
  - free: lp_solve, GPLk, ...
  - commercial: dozens!, for example:

Solver Product	Features	Platform	Phone (+1)	E-mail address
C-WHIZ	SI	PC U	703-412-3201	info@ketronms.com
CPLEX	SBINQ	PC U	800-367-4564 775-831-7744	info@ilog.com
FortMP	SBIQ	PC U	+44 18-9525-6484	optirisk@optirisk-systems.com
HI-PLEX	S	PC U	+44 20-7594-8334	i.maros@ic.ac.uk
HS/LP	SI	PC	201-627-1424	info@haverly.com
ILOG Opt Suite	SBINQ	PC U	800-367-4564 775-831-7744	info@ilog.com

Maximum Network Flow



Input description: A directed graph $G = (V, E)$, where $|V| = n$ and each edge $e = (i, j) \in E$ has a capacity $c_e = u_{i,j}$. A source node s and sink node t.

Problem description: What is the maximum flow you can route from s to t while respecting the capacity constraint of each edge?

A linear programming model

Let $x_{i,j}$ be a variable accounting for the flow from vertex i through directed edge (i, j). Also let the index for source be 1 and the index for the sink be n. The flow through this edge is constrained by its capacity $u_{i,j}$, so

$$0 \leq x_{i,j} \leq u_{i,j}, \quad 1 \leq i, j \leq n$$

Furthermore, an equal flow comes in as goes out at each nonsource or sink vertex, so (*flow conservation*)

$$\sum_{j=1}^{n} x_{i,j} - \sum_{j=1}^{n} x_{j,i} = 0$$

We seek the assignment that maximizes the flow into sink, namely $\sum_{i=1}^{n} x_{i,n}$ (or $\sum_{i=1}^{n} x_{1,i}$)

$$\max \sum_{i=1}^{n} x_{1,i}$$

$$\sum_{j=1}^{n} x_{i,j} - \sum_{j=1}^{n} x_{j,i} = 0,\ i = 1, \ldots, n$$

$$0 \leq x_{i,j} \leq u_{i,j},\ \forall e = (i,j) \in E$$

$$0 \leq x_{i,j} \geq u_{i,j},\ \forall e = (i,j) \in E$$

Fractional Knapsack

Consider the problem parameters as W: the knapsack capacity, v_i: the value of the item i, the w_i: the weight of the item i.

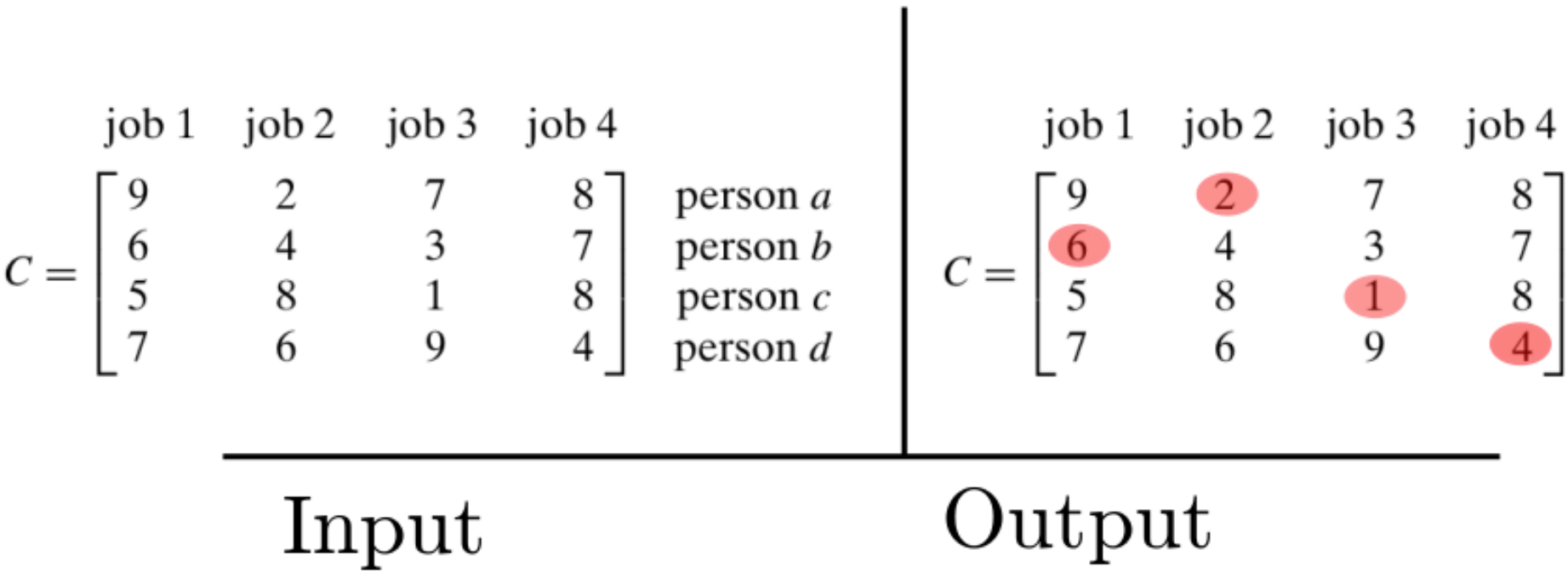
- Let x_i be the fraction of item i decided to be in knapsack.

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \leq W$$

$$0 \leq x_i \leq 1,\ 1 \leq i \leq n$$

Job Assignment



Input description: an $n \times n$ cost matrix C

Problem description: select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible.

A linear programming model

Let $x_{ij} = 1$, if person i performs task j, and 0 otherwise $(1 \leq i, j \leq n)$.

$$\text{Minimize} \qquad Z = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij},$$

subject to

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \text{for } i = 1, 2, \ldots, n,$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \text{for } j = 1, 2, \ldots, n,$$

and

$$x_{ij} \geq 0, \qquad \text{for all } i \text{ and } j$$
$$(x_{ij} \text{ binary}, \qquad \text{for all } i \text{ and } j).$$

If we delete the parenthetical restriction that the x_{ij} be binary, the model clearly is a special type of linear programming problem and so can be readily solved. **Fortunately, in this problem, we can delete this restriction.**

We are not always so lucky!

Integer Linear Programming (ILP)

An *integer linear programming problem* (ILP) is a linear programming problem in which **at least one of the variables is restricted to integer values**.

> Unfortunately, it is NP-complete to solve integer or mixed programs to optimality.

► Other approaches such as the **branch-and-bound technique** are typically used for solving integer linear programming problems.

0/1 Knapsack

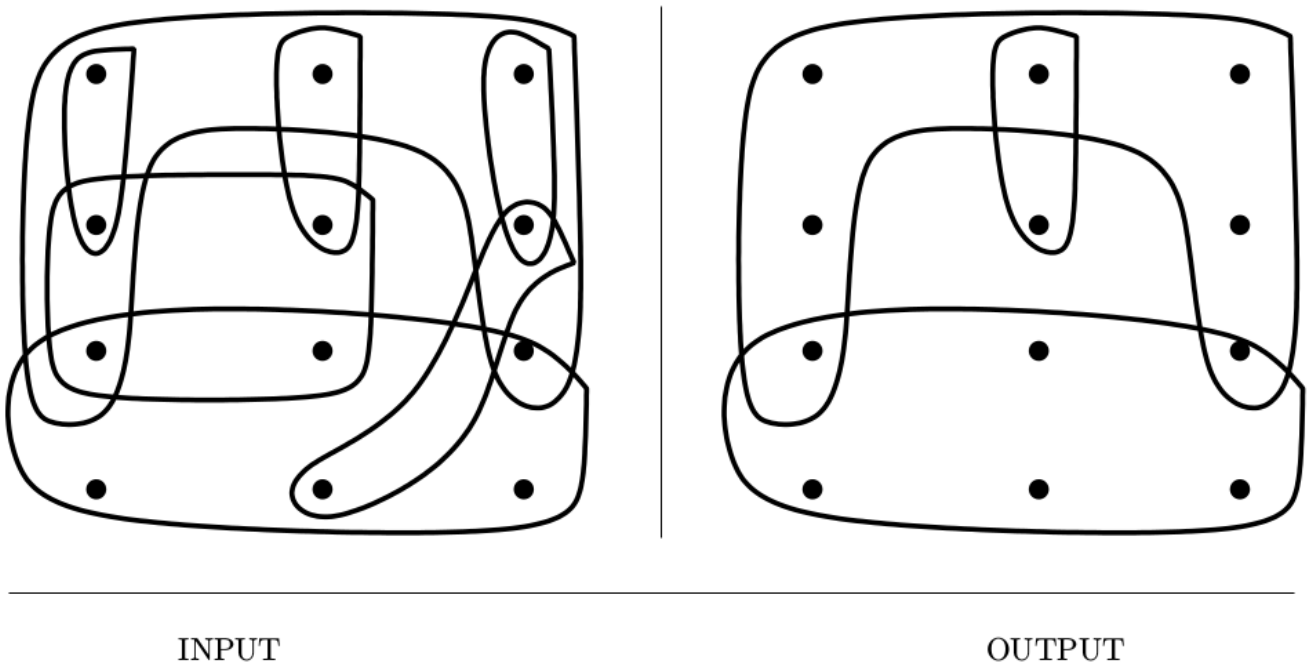
Consider the problem parameters as W: the knapsack capacity, v_i: the value of the item i, the w_i: the weight of the item i.
Let $x_i = 1$ if item i is decided to be in knapsack and 0, otherwise.

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \leq W$$

$$x_i \in \{0, 1\},\ 1 \leq i \leq n$$

Set Cover



Input description: A collection of subsets $S = \{S_1, \ldots, S_m\}$ of the universal set $U = \{1, \ldots, n\}$.

Problem description: What is the smallest subset T of S whose union equals the universal set – i.e., $\cup_{i=1}^{|T|} T_i = U$?

Motivation:
The set-covering problem abstracts many commonly arising combinatorial problems. As a simple example, suppose that U represents a set of skills that are needed to solve a problem and that we have a given set of people available to work on the problem. We wish to form a committee, containing as few people as possible, such that for every requisite skill in U, at least one member of the committee has that skill.

► Vertex-cover problem is a special case of the set-cover problem: To turn vertex cover into a set cover problem, let universal set U represent the set E of edges from G, and define S_i to be the set of edges incident on vertex i. A set of vertices defines a vertex cover in graph G iff the corresponding subsets define a set cover in this particular instance. However, since each edge can be in only two different subsets, vertex cover instances are simpler than general set cover.

Set cover is NP-hard.

Set Covering

Cover all the elements $\{1, 2, 3, 4, 5\}$ using the minimum number of subsets $\{1, 2, 3, 5\}$, $\{1, 2, 4, 5\}$, $\{1, 3, 4\}$, $\{2, 3, 4, 5\}$, $\{3, 4, 5\}$.

- Let x_i denote the subset i is in the optimal solution or not.

$$\min \sum_{i=1}^{5} x_i$$

$$x_1 + x_2 + x_3 \geq 1$$

$$x_1 + x_2 + x_4 \geq 1$$

$$x_1 + x_3 + x_4 + x_5 \geq 1$$

$$x_2 + x_3 + x_4 + x_5 \geq 1$$

$$x_1 + x_2 + x_4 + x_5 \geq 1$$

$$x_i \in \{0, 1\},\ 1 \leq i \leq 5$$

Backtracking

Backtracking

Backtracking is a more intelligent variation of exhaustive search approach: it is often possible to reject a solution by looking at just a small portion of it.

The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.

- If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component.
- If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm **backtracks** to replace the last component of the partially constructed solution with its next option.

We will model our combinatorial search solution as a vector$a = (a_1, a_2, ..., a_n)$, where each element a_i is selected from a finite ordered set S_i.

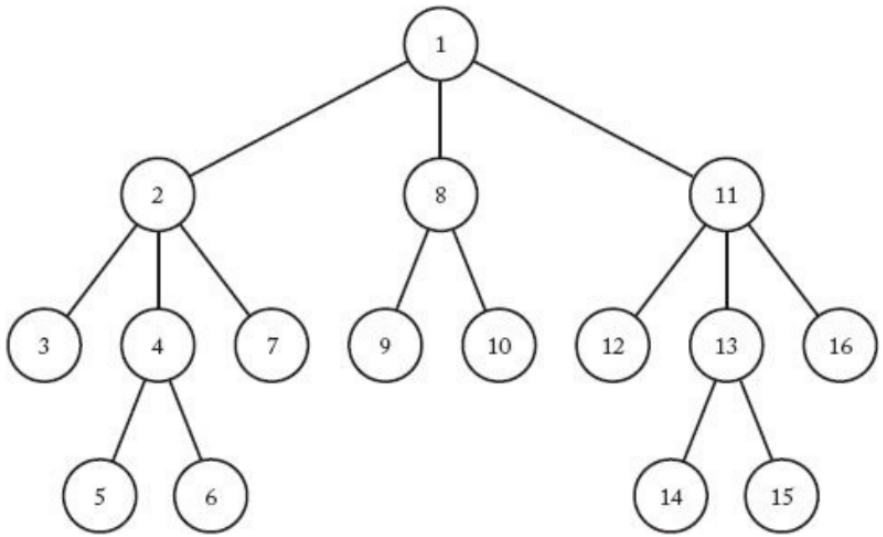
- Such a vector might represent an arrangement where a_i contains the ith element of the permutation.
- Or, the vector might represent a given subset S, where a_i is true if and only if the ith element of the universe is in S.
- The vector can even represent a sequence of moves in a game or a path in a graph, where a_i contains the ith event in the sequence.

At each step in the backtracking algorithm, we try to extend a given partial solution $a^{(k)} = (a_1, a_2, ..., a_k)$ by adding another element at the end. After extending it, we must test whether what we now have is a solution: if so, we should print it or count it. If not, we must check whether the partial solution is still potentially extendible to some complete solution.

It is convenient to implement this kind of processing by constructing a tree of choices being made, called the **state-space tree**.

- Its root represents an initial state before the search for a solution begins.
- The nodes of the first level in the tree represent the choices made for the first component of a solution, the nodes of the second level represent the choices for the second component, and so on.
- A node in a state-space tree is said to be **promising** if it corresponds to a partially constructed solution that may still lead to a complete solution, otherwise, it is called **nonpromising**.
- Leaves represent either nonpromising dead ends or complete solutions found by the algorithm.

Backtracking is a modified depth-first search of a tree. So let's review the depth-first search of a tree before proceeding.




```
void depth_first_tree_search (node v)
{
   node u;
   visit v;
   for (each child u of v)
      depth_first_tree_search(u);
}
```

Discussion Question

What should be added to the previous DFS algorithm to effectively visit all nodes of a general graph?

Backtracking: DFS the state-space tree with pruning

- If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child.
- If the current node turns out to be nonpromising, the algorithm backtracks to the node's parent to consider the next possible option for its last component
- if there is no such option, it backtracks one more level up the tree, and so on.
- Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

A general algorithm for the backtracking approach is as follows:

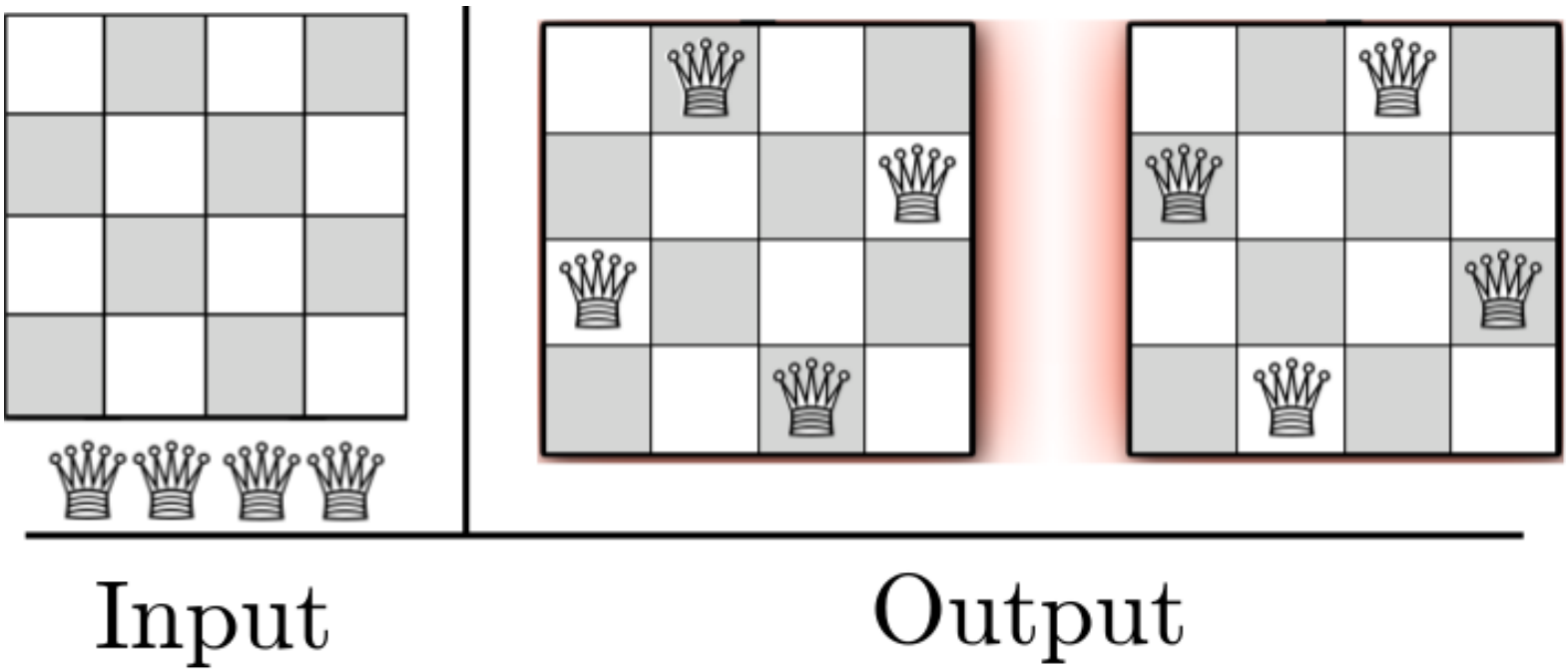
```
void checknode (node v)
{
  node u;

  if (promising(v))
     if (there is a solution at v)
        write the solution;
     else
        for (each child u of v)
           checknode(u);
}
```

The root of the state space tree is passed to checknode at the top level.

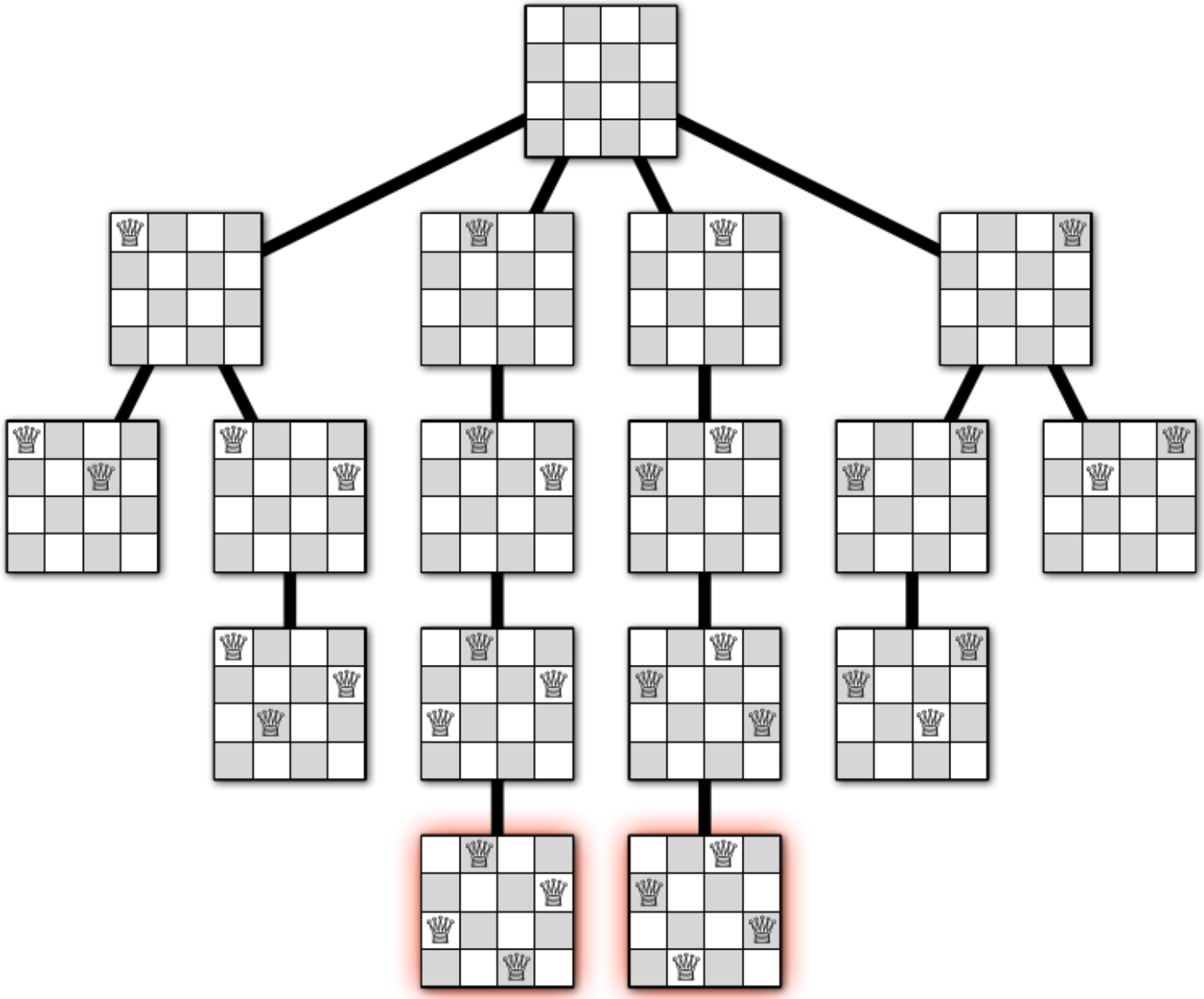
A backtracking algorithm is identical to a DFS, except that the children of a node are visited only when the node is promising and there is not a solution at the node.

N Queen



Input description: n queens and an $n \times n$ chessboard
Problem description: position n queens on an $n \times n$ chessboard so that no two queens threaten each other. That is, no two queens may be in the same row, column, or diagonal.

- Note that no two queens can be in the same row. Thus, the instance can be solved by assigning each queen a different row. Because each queen can be placed in one of four columns, there are $4 \times 4 \times 4 \times 4 = 256$ candidate solutions.
- We can create the candidate solutions by constructing a tree in which the column choices for the first queen (the queen in row 1) are stored in level-1 nodes in the tree, the column choices for the second queen (the queen in row 2) are stored in level-2 nodes, and so on.
- A path from the root to a leaf is a candidate solution (recall that a leaf in a tree is a node with no children). This tree is called a **state space tree**.



Notice that a backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithms. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

The Backtracking Algorithm for the n-Queens Problem

Problem: Position n queens on a chessboard so that no two are in the same row, column, or diagonal.

Inputs: positive integer n.

Outputs: all possible ways n queens can be placed on an $n \times n$ chessboard so that no two queens threaten each other. Each output consists of an array of integers col indexed from 1 to n, where $col[i]$ is the column where the queen in the ith row is placed.

```
void queens (index i)
{
  index j;

  if (promising(i))
     if (i == n)
        cout << col[1] through col [n];
     else
        for (j = 1; j <= n; j++){      // See if queen in
           col[i + 1] = j;             // (i + 1)st row can be
           queens(i + 1);              // positioned in each of
                                       // the n columns.
        }
}
bool promising (index i)
{
   index k;
   bool switch;

   k = 1;
   switch = true;                  // Check if any queen threatens
   while (k < i &&  switch){       // queen in the ith row.
      if (col[i] == col[k] || abs(col[i] - col[k]) == i - k)
         switch = false;
      k++;
   }
   return switch;
}
```

Diagonal Threats:
the difference in the columns is the same as the difference in the rows

Subset Sum

Input	Output
$n = 5$ $W = 21$ $w_1 = 5, w_2 = 6, w_3 = 10, w_4 = 11, w_5 = 16$	$\{w_1, w_2, w_3\}, \{w_1, w_5\}, \{w_3, w_4\}$

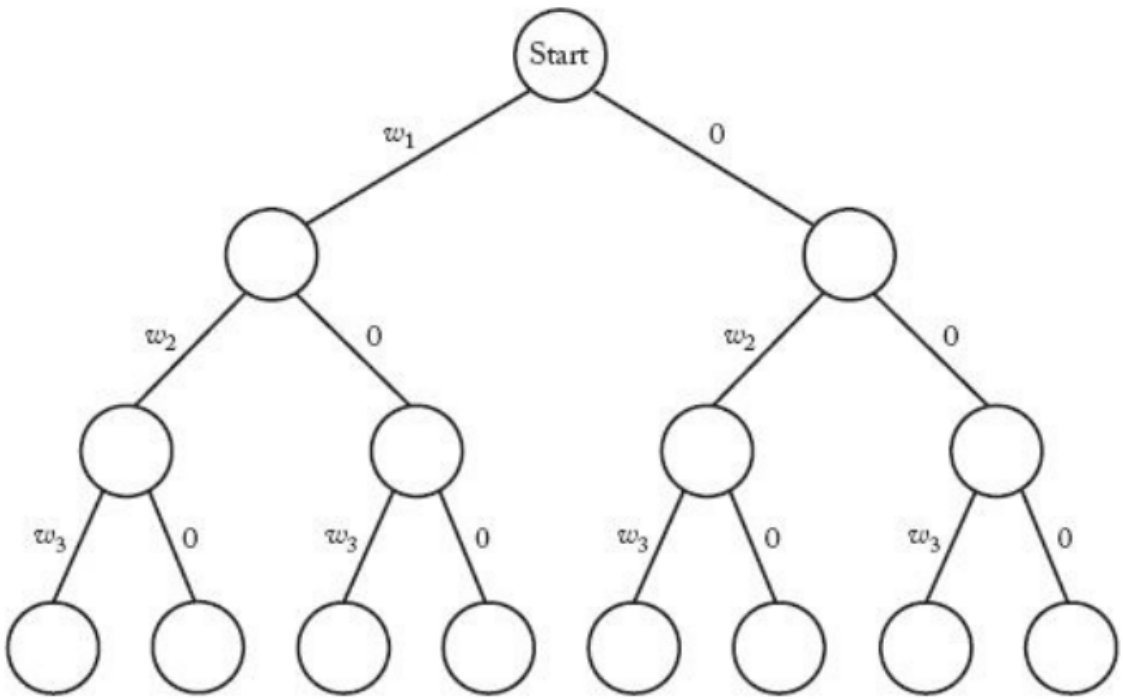
Input description: n positive integers (weights) w_i and a positive integer W

Problem description: is there a subset of elements that add up to W?

The subset sum problem:

- A special case of 0/1 knapsack where each item have the same price per pound. In this case, our problem is equivalent to ignoring the price and just trying to minimize the amount of empty space left in the knapsack.
- Unfortunately, even this restricted version is NP-complete, so we cannot expect an efficient algorithm that always solves the problem.

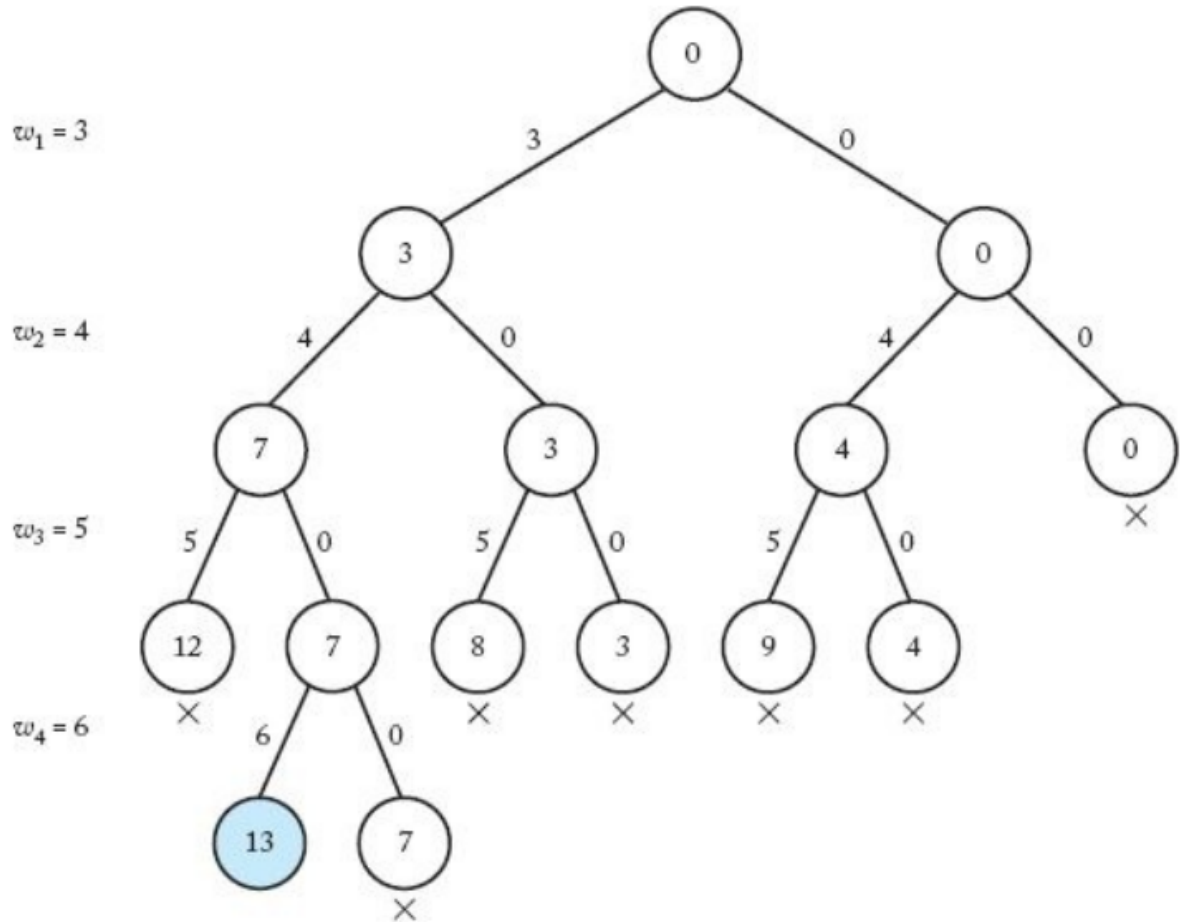
- A small instance can be solved by inspection. For larger values of n , a systematic approach is necessary.
- One approach is to create a **state space tree**: When we include w_i , we write w_i on the edge where we include it. When we do not include w_i , we write 0.



Non-promising Signs:
Let's sort the weights in nondecreasing order before doing the search. Also assume that $weight$ be the sum of the weights that have been included up to a node at level i.

- A node at the ith level is nonpromising if
$$weight + w_{i+1} > W$$
- if $total$ is the total weight of the remaining weights, a node at the ith level is nonpromising if
$$weight + total < W$$

Example: $n = 4, W = 13, \{w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6\}$



```
void sum_of_subsets (index i,
                     int weight, int total)
{
  if (promising(i))
     if (weight == W)
        cout << include[1] through include[i];
     else{
        include[i + 1] = "yes";                  // Include w[i + 1].
        sum_of_subsets(i + 1, weight + w[i +1], total - w[i + 1]);
        include[i + 1] = ''no";                  // Do not include w[i + 1].
        sum_of_subsets(i + 1, weight, total - w[i + 1]);
     }
}

bool promising (index i);
{
  return (weight + total >= W) && (weight == W || weight + w[i + 1] <= W);
}
```

- This algorithm uses an array $include$. It sets $include[i]$ to "yes" if $w[i]$ is to be included and to "no" if it is not.
- n , w , W , and $include$ were defined globally, and the top-level call to $sum\_of\_subsets$ would be as follows:

$$sum\_of\_subsets(0, 0, \sum_{i=1}^{n} w[i])$$

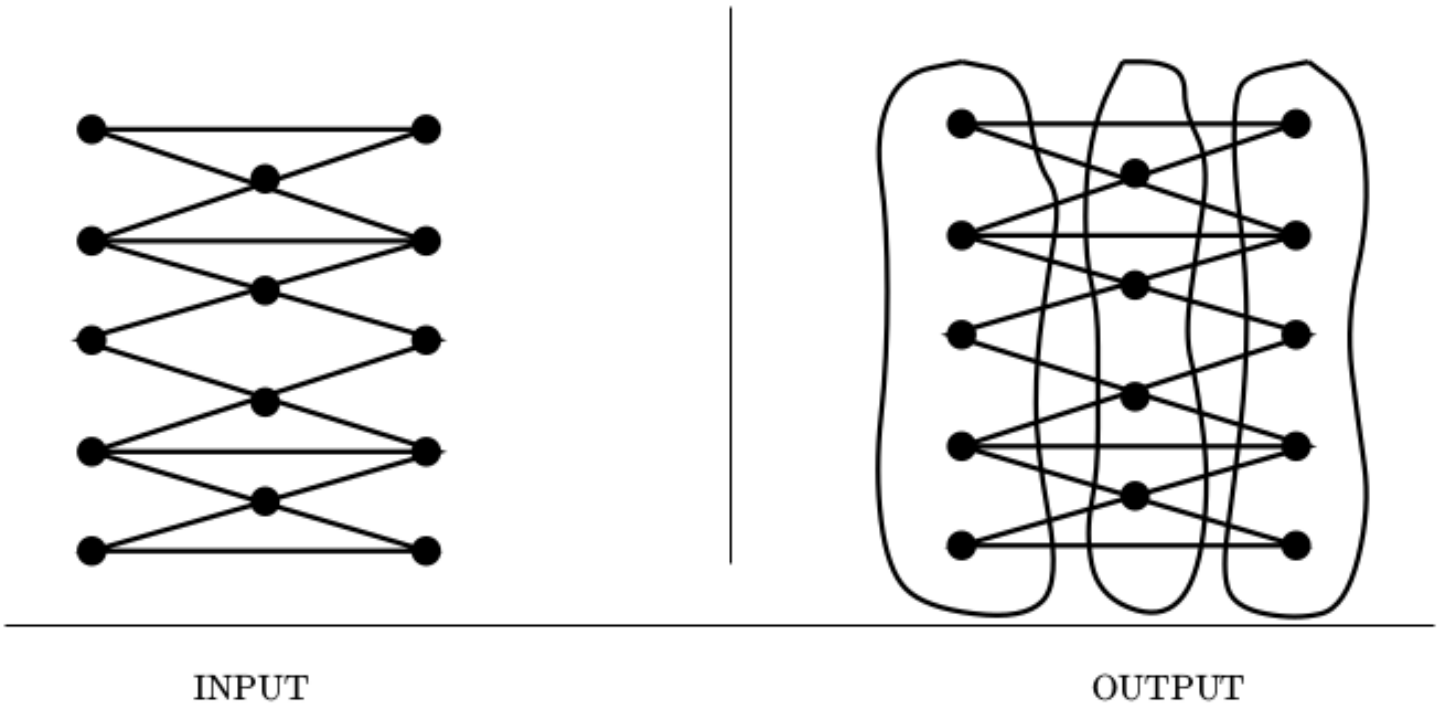
- We do not need to check for the terminal condition $i = n$. Also, there is never a reference to the nonexistent array item $w[n+1]$ in function $promising$
  - `Discussion Question: why?`
- The number of nodes in the state space tree:

$$1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$$

Does the $sum\_of\_subsets$ algorithm, for every instance, visit only a small portion of the state space tree? No
► if we take $\sum_{i=1}^{n-1} w_i < W, w_n = W$, there is only one solution $\{w_n\}$, and it will not be found until an exponentially large number of nodes are visited. (Exercise: what is the exact number?)

Even though the worst case is exponential, the algorithm can be efficient for many large instances.

Graph/Vertex Coloring



Input description: A graph $G = (V, E)$
Problem description: Color the vertices of V using the minimum number of colors such that i and j have different colors for all $(i, j) \in E$. (The smallest number of colors sufficient to vertex-color a graph is its chromatic number.)

Motivation: Vertex coloring arises in many **scheduling** and **clustering** applications.

- Register allocation in compiler optimization: Each variable in a given program fragment has a range of times during which its value must be kept intact, in particular after it is initialized and before its final use. Any two variables whose life spans intersect cannot be placed in the same register.
  - Construct a graph where each vertex corresponds to a variable, with an edge between any two vertices whose variable life spans intersect.
  - Since none of the variables assigned the same color clash, they all can be assigned to the same register.
  - No conflicts will occur if each vertex is colored using a distinct color.

  But computers have a limited number of registers, so we seek a coloring using the fewest colors.

This ia an instance of the **Design Graphs, Not Algorithms** strategy

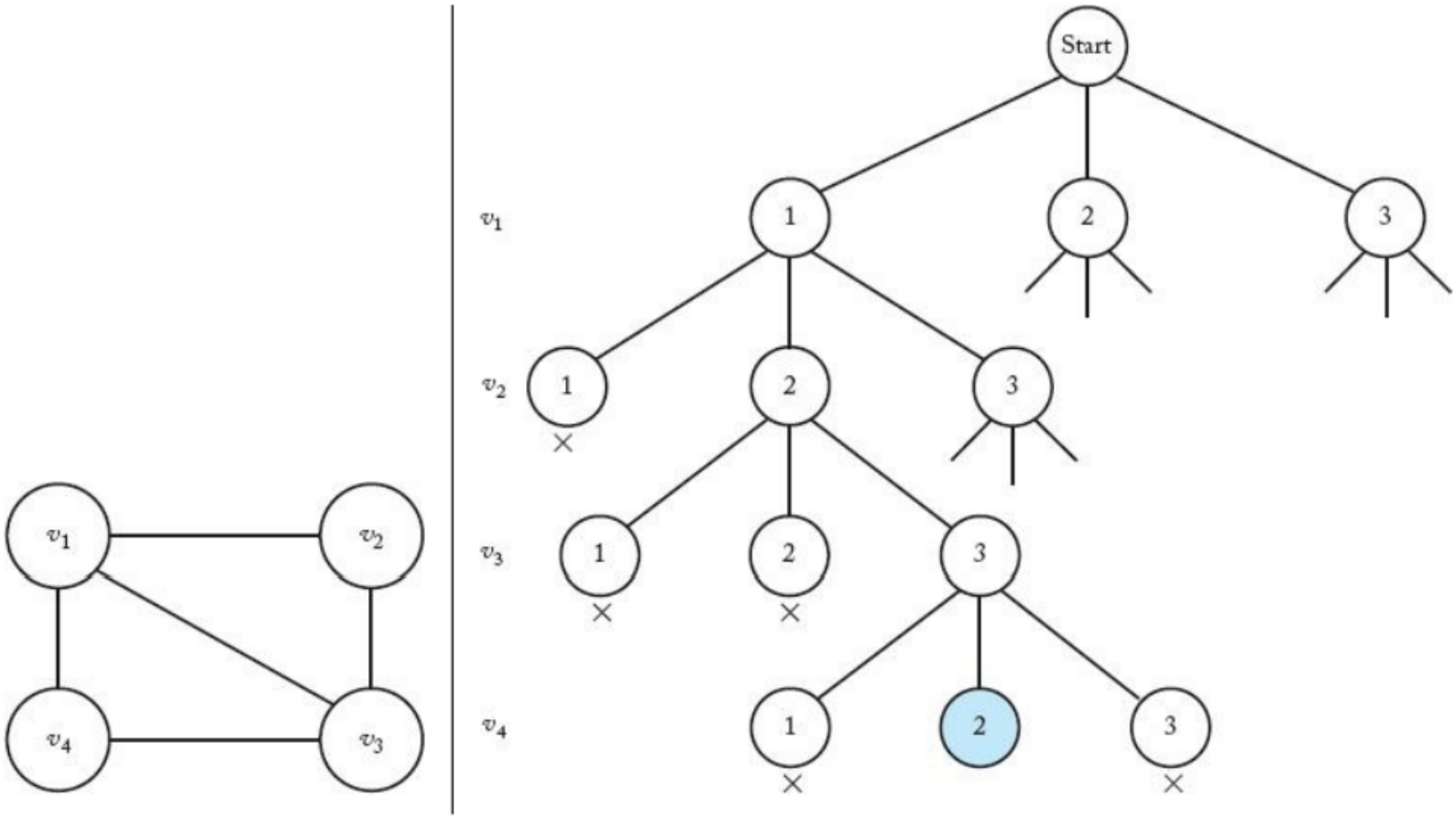
The m**-Coloring problem** concerns finding all ways to color an undirected graph using **at most** m **different colors**, so that no two adjacent vertices are the same color.

- This is the decision version of the original graph coloring problem.
- Here, we consider backtracking to solve instances of the m-Coloring problem.

A straightforward state space tree for the m-Coloring problem:

- Each possible color is tried for vertex v_1 at level 1, each possible color is tried for vertex v_2 at level 2, and so on until each possible color has been tried for vertex v_n at level n.
- Each path from the root to a leaf is a candidate solution.
- We check whether a candidate solution is a solution by determining whether any two adjacent verices are the same color.

To avoid confusion, remember in the following discussion that "node" refers to a node in the state space tree and "vertex" refers to a vertex in the graph being colored.



We can backtrack in this problem because a node is nonpromising if a vertex that is adjacent to the vertex being colored at the node has already been colored the color that is being used at the node. (The number in a node: the number of the color)

```
void m_coloring (index i)
{
  int color;

  if (promising(i))
     if (i == n)
        cout << vcolor[1] through vcolor[n];
     else
        for (color = 1; color <= m; color++){    // Try every
           vcolor[i + 1] = color;                // color for
           m_coloring(i + 1);                    // next vertex.
        }
}

bool promising (index i)
{
  index j;
  bool switch;

  switch = true;
  j = 1;
  while (j < i && switch){                       // Check if an
     if (W[i][j] && vcolor[i] == vcolor[j])      // adjacent vertex
        switch = false;                          // is already
     j++;                                        // this color.
  }
  return switch;
}
```

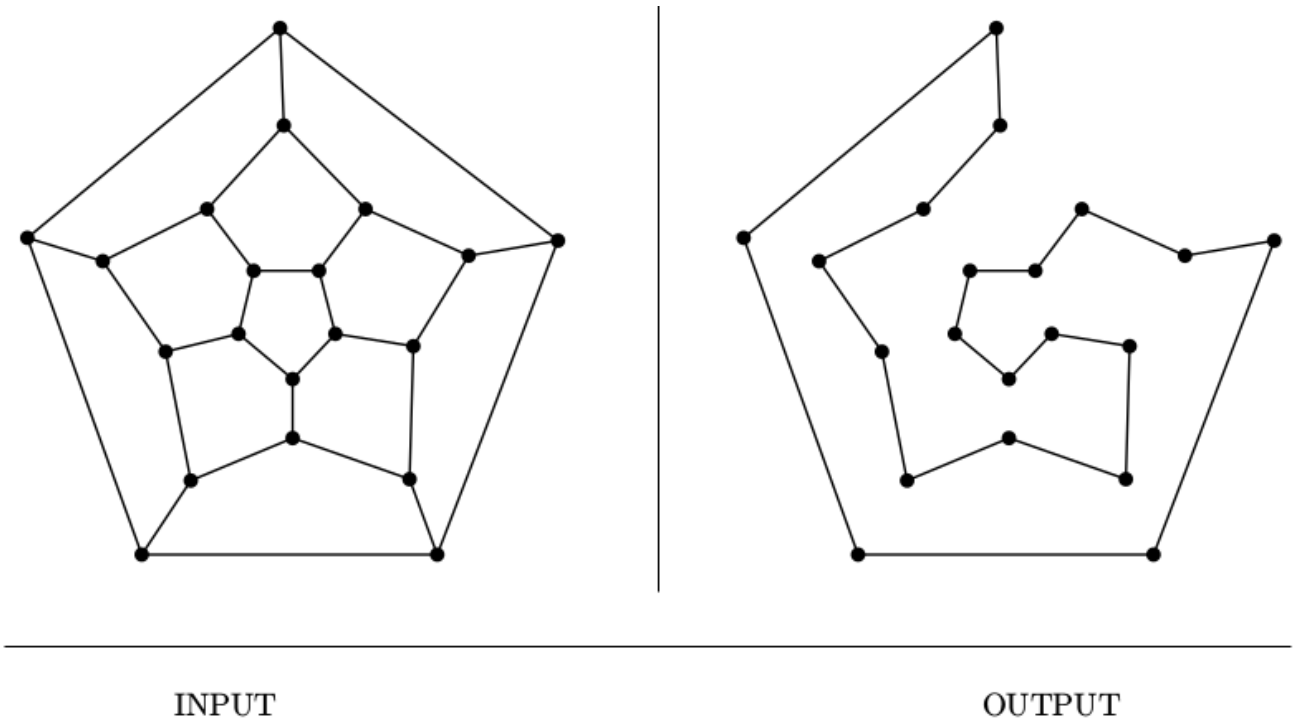
$W[i][j]$ is true iff there is an edge between i th vertex and the j th vertex

The output for each coloring is an array $vcolor$ indexed from 1 to n, where $vcolor[i]$ is the color (an integer between 1 and m) assigned to the ith vertex.

The top level call is $m\_coloring(0)$.

- The number of nodes in the state space tree:
$$1 + m + m^2 + \cdots m^n = \frac{m^{n+1}-1}{m-1}$$
- For a given m and n, it is possible to create an instance that checks at least an exponentially large number of nodes (in terms of n).
  Example: almost every node in the state space tree will be visited to determine that no solution exists fot the 3-coloring problem of the following instance.

Hamiltonian Circuit



Input description: A graph $G = (V, E)$
Problem description: Find a tour of the vertices using only edges from G, such that each vertex is visited exactly once.

A state space tree for this problem is as follows.

- Put the starting vertex at level 0 in the tree; call it the zeroth vertex on the path.
- At level 1, consider each vertex other than the starting vertex as the first vertex after the starting one.
- At level 2, consider each of these same vertices as the second vertex, and so on.
- Finally, at level $n-1$, consider each of these same vertices as the $(n-1)$ st vertex.

The following considerations enable us to backtrack in this state space tree:

1. The ith vertex on the path must be adjacent to the $(i-1)$ st vertex on the path.
2. The $(n-1)$st vertex must be adjacent to the 0th vertex (the starting one).
3. The i th vertex cannot be one of the first $i-1$ vertices.

```
void hamiltonian (index i)
{
  index j;

  if (promising(i)
     if (i == n - 1)
        cout << vindex[0] through vindex[n - 1];
     else
        for (j = 2; j <=n; j++){          // Try all vertices as
           vindex[i + 1] = j;             // next one.
           hamiltonian(i + 1);
        }
}
bool promising (index i)
{
  index j;
  bool switch;

  if (i == n - 1 &&   !W[vindex[n - 1]] [vindex[0]])
     switch = false;                      // First vertex must be adjacent
  else if (i > 0 &&   !W[vindex[i - 1]]
     switch = false; [vindex[i]])         // to last. ith vertex must
  else{                                   // be adjacent to (i - 1)st.
     switch = true;
     j = 1;
     while (j < i &&  switch){            // Check if vertex is
         if (vindex[i] == vindex[j])      // already selected.
            switch = false;
         j++;
     }
   }
   return switch;
}
```

$W[i][j]$ is true iff there is an edge between i th vertex and the j th vertex

$vindex$ indexed from 0 to $n-1$, where $vindex[i]$ is the index of the ith vertex on the path. The index of the starting vertex is $vindex[0]$.

the top-level called to hamiltonian:
$vindex[0] = 1;$
$hamiltonian(0);$

The number of nodes in the state space tree for this algorithm is

$$1 + (n-1) + (n-1)^2 + (n-1)^3 + \cdots + (n-1)^{n-1} = \frac{(n-1)^{n-1}}{n-2},$$

which is much worse than exponential.

- Let the only edge to v_1 be one from v_2, and let all the vertices other than v_1 have edges to each other. There is no Hamiltonian Circuit for the graph, and the algorithm will check a worse-than-exponential number of nodes to learn this (although the algorithm does not check the entire state space tree).