Algorithms and Lab (CSE130)

State Space Search Techniques : Backtracking

Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: These notes are prepared from the following resources.

- (main text)Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour
- Python Algorithm (파이썬 알고리즘) by Y.K. Choi (2021) (Korean)
- Introduction to the Design and Analysis of Algorithms by Anany Levitin
- Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- https://www.geeksforgeeks.org

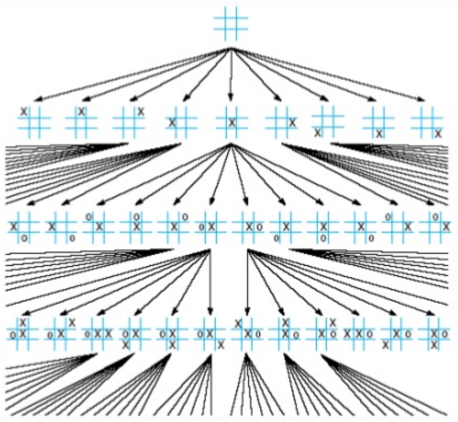
Contents

State Space Search Technique

- **State space search** is a process used in the field of computer science, in which successive configurations or states of an instance are considered, with the intention of finding a **goal state** (solution).
- The concept of **state space** is important in solving real world problems.
- These states/configurations may represent all possible arrangements of objects (permutations) or all possible ways of building a collection of them (subsets).
- Thus, these states/configurations may represent partial solutions.
- Different search strategies or algorithms are applied to explore the search space in order to find a solution
- Within an AI context, a **search algorithm** takes a problem as input and returns a solution in the form of an action sequence.
- The **state space** concept often provides the framework to solve the real world problems .

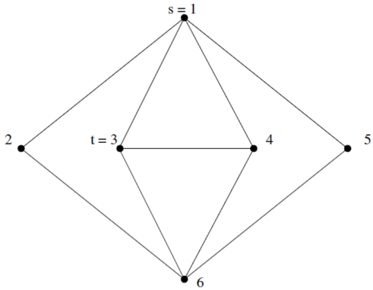
State Space Tree

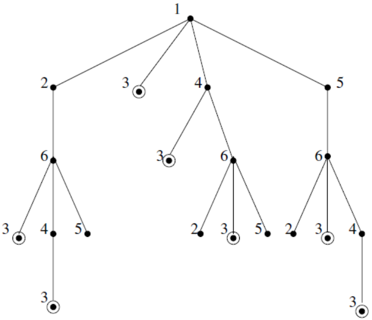
- A problem can be represented by a tree structure(named as **State Space Tree**)
- Few terminologies are provided related to **state space** formulation of a problem
- **State Space:** Set of All states reachable from the initial state and it forms a graph (Tree) in which the nodes are states and the arcs are actions
- **A Start State:** The state from where the search begins.
- **Path:** A path in the state space is a sequence of states connected by a sequence of actions
- **Solution:**
  - A path from the initial state to another state (the goal state)
  - A state/node
- **Optimal Solution:** Has lowest path cost amongst all solutions



State Space Tree (cont...)

Components of State Space Tree : State Space Tree is defined formally by the five components:



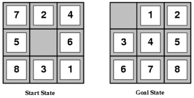


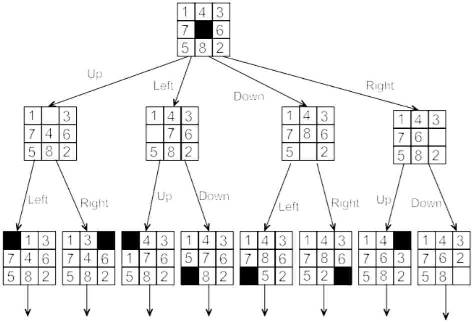
1. **Initial state:** Usually, root node of the tree
2. **Actions:** A description of the possible actions available at a state. Given a particular state *s*, *ACTIONS*(*s*) returns the set of actions that can be executed in *s*.
3. **Transition Model:** A description of what each action does. The resultant state after performing an action. It is specified by a function *RESULT*(*s*, *a*) that returns the state that results from doing action *a* in state *s*.
4. **Goal Test:** determines whether a given state is a goal state.
5. **Path Cost:** A function that assigns a numeric cost to each path.

State Space Tree (cont...)

Example: 8-puzzle Problem A typical instance of the 8-puzzle problem is shown in Figure

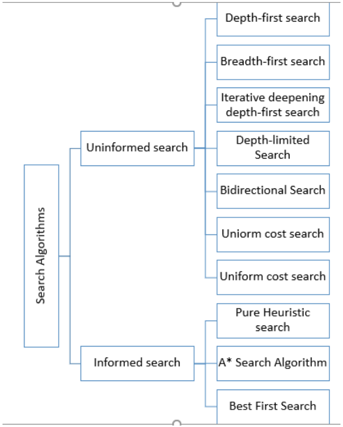
1. **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
2. **Initial state:** Any state can be designated as the initial state.
3. **Actions:** The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.
4. **Transition model:** Given a state and action, this returns the resulting state.
5. **Goal test:** This checks whether the state matches the goal configuration
6. **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.





Searching State Space Tree

- Once a **state space tree** is generated for a problem then the main issue is to find the goal state(solution).
- There are many searching techniques and can be divided into two major categories
- **Uninformed search** algorithms do not have additional information about state or search space other than how to traverse the tree.
- **Informed search algorithm** contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node.

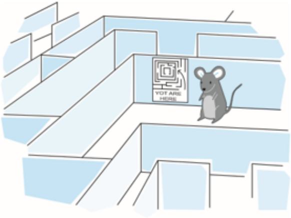


State Space Tree (cont...)

Maze Problem

- A Maze is given as $N \times N$ binary matrix of blocks where source block is $maze[0][0]$ and destination block is maze[N-1][N-1].
- A rat starts from source and has to reach the destination. The rat can move only in four directions: upward, down, forward, backword .
- In the maze matrix, 1 means the block is a dead end and 0 means the block can be used in the path from source to destination.

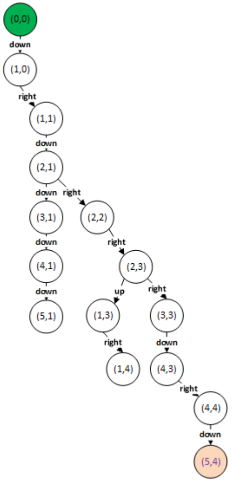
```
char maze[N][N] = {
{ ‘e', '1', '1', '1', '1', '1' },
{ ‘0', '0', '1', '0', '0', '1' },
{ '1', '0', '0', '0', '1', '1' },
{ '1', '0', '1', '0', '1', '1' },
{ '1', '0', '1', '0', '0’, ‘1' },
{ '1', '1', '1', '1', ‘0’, ‘1' },
};
```



- The task is to check if there exists any path so that the rat can reach at the destination or not.

State Space Tree for Maze Problem

1. **States:** location of the rate in the maze at (row, col)
2. **Initial state:** The first cell maze[0][0]
3. **Actions:** up, down, left, right
4. **Transition model:** upon an action new location in maze[row][col]
5. **Goal test:** reaching at exit in maze[5][4]
6. **Path cost:** Each step costs 1, so the path cost is the number of steps in the path (number of edges).



Depth First Search

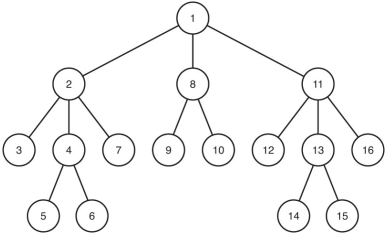
Depth First Search

- A preorder tree traversal is a depth-first search of the tree.
- The root is visited first, and a visit to a node is followed immediately by visits to all descendants of the node
- Depth First Search using Stack data structure

```
procedure DFS(Tree T)
    Stack S, Node u, v
    initialize(S)
    v=root of T
    visit v
    push(S, v)
    while (!empty(S)) do
        pop(S)
        for (each child u of v) do
            visit u
            push(S, u)
        end for
    end while
end procedure
```

- A simple recursive algorithm for depth-first search is given below.

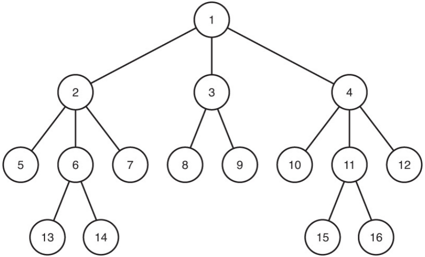
```
procedure DFS(v)
    Node u
    visit v
    for (each child u of v) do
        DFS(u)
    end for
end procedure
```



Breadth First Search

Breadth First Search

- A breadth-first search consists of visiting the root first, followed by all nodes at level 1, followed by all nodes at level 2, and so on.



- Unlike depth-first search, there is no simple recursive algorithm for breadth-first search. However, it can be implemented using a queue data structure.

```
procedure BFS(Tree T)
    Queue Q
    Node u, v
    initialize(Q)
    v=root of T
    visit v
    enqueue(Q, v)
    while (!empty(Q)) do
        v=dequeue(Q)
        for ((each child u of v)) do
            visit u
            enqueue(Q, u)
        end for
    end while
end procedure
```

Backtracking

- **Backtracking** is a modified ***depth-first search(DFS)*** of a tree.
- **Backtracking** is the procedure whereby, after determining that a node can lead to nothing but dead ends, we go back ("backtrack") to the node's parent and proceed with the search on the next child.
- We call a **nonpromising** node if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise, we call it **promising node**.
- In other words, backtracking consists of doing a depth-first search of a state space tree, checking whether each node is promising, and, if it is nonpromising, backtracking to the node's parent.
- This is called **pruning the state space tree**, and the subtree consisting of the visited nodes is called the **pruned state space tree**.

- A general algorithm for the backtracking approach is as follows:

```
procedure BACKTRACK(v)
    Node u
    if (promising(v)) then
        if (there is a solution at v) then
            write the solution
        else
            for (each child u of v) do
                backtrack(u)
            end for
        end if
    end if
end procedure
```

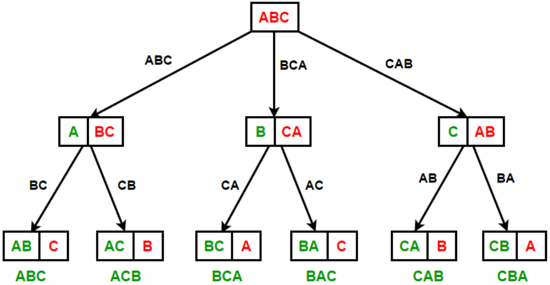
- The **function promising** is different in each application of backtracking.

- An improved version of the general algorithm for the backtracking approach is as follows:

```
procedure BACKTRACK2(v)
    Node u
    for (each child u of v) do
        if (promising(v)) then
            if (there is a solution at v) then
                write the solution
            else
                backtrack2(u)
            end if
        end if
    end for
end procedure
```

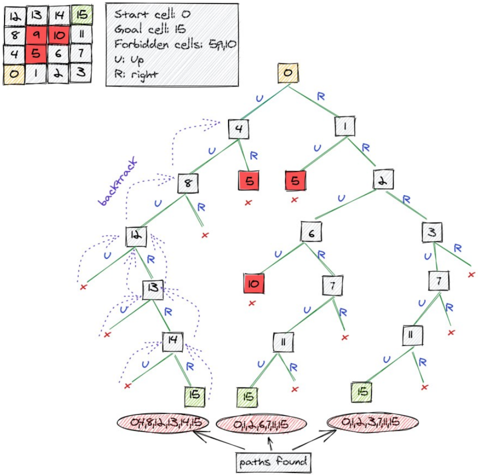
Permutations

- For generating permutations $\{1, ..., n\}$, there are n distinct choices for the value of the first element of a permutation.
- Once have fixed a_1, there are $n-1$ candidates remaining for the second position.
- Repeating this argument yields a total of $n! = \prod_{i=1}^{n}$, i distinct permutations
- The set of candidates for the ith position will be the set of elements that have not appeared in the $(i-1)$ elements of the partial solution, corresponding to the first $i-1$ elements of the permutation.
- In the scheme of the general backtrack algorithm, $S_k = \{1, ..., n\} - a$ and if $k == n$ then it will be a solution



Path Finding

- Generally, we use backtracking when all possible solutions of a problem need to be explored.
- It is also often employed to identify solutions that satisfy a given criterion also called a constraint.
- For constraint satisfaction problems, the search tree is "pruned" by abandoning branches of the tree that would not lead to a potential solution.
- Suppose we have a rectangular grid with a robot placed at some starting cell. It then has to find all possible paths that lead to the target cell. Few cells are forbidden.
- Once a solution is found, the algorithm backtracks (goes back a step, and explores another path from the previous point) to explore other tree branches to find more solutions.



Backtracking for n-Queens Problem

- The goal of the 4-queens(n=4) problem is to place four queens on a chessboard such that no queen attacks any other. A queen attacks any piece in the same row, column or diagonal

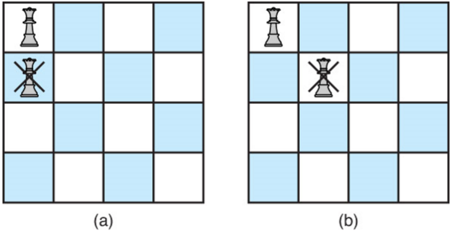


Figure 1: Diagram showing that if the first queen is placed in column 1, the second queen cannot be placed in column 1 (a) or column 2 (b).

1. **States:** Any arrangement of 0 to 4 queens on the board is a state.
2. **Initial state:** No queens on the board.
3. **Actions:** Add a queen to any empty square.
4. **Transition model:** Returns the board with a queen added to the specified square.
5. **Goal test:** 4 queens are on the board, none attacked.
6. **Path Cost:** No of steps to search place on the board for each queen.

n-Queens Problem (cont...)

- State space tree for 4-Queens problem: A possible solution is visiting all nodes between (level-1) node to (max-level) a leaf node

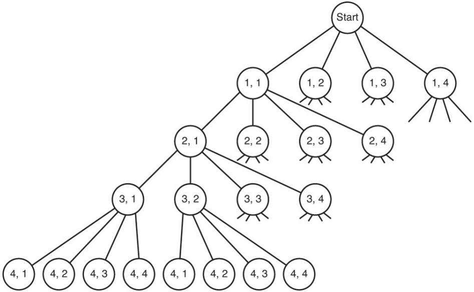


Figure 2: A portion of the state space tree for the instance of the n-Queens problem in which $n = 4$. The ordered pair $<$i, j$>$, at each node means that the queen in the i th row is placed in the jth column.

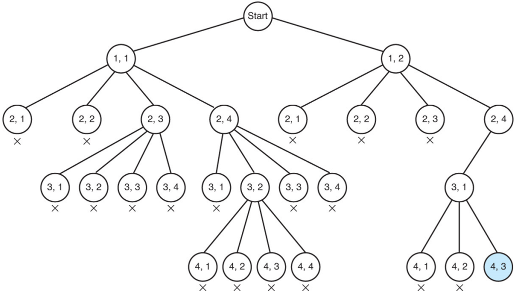


Figure 3: A portion of the pruned state space tree produced when backtracking is used to solve the instance of the n-Queens problems in which n=4

Backtracking Algorithm for n-Queens Problem

- A backtracking algorithm does need actually to create a tree. The state space tree exists **implicitly** in the algorithm because it is not actually constructed.
- **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, were $col[i]$ is the column where the queen in the i_{th} row is placed.
- A candidate solution in 8-queens problem

$$\begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ [0 & 4 & 7 & 5 & 2 & 6 & 1 & 3] \end{matrix} \left\{ \begin{matrix} i \\ Col\,[i] \end{matrix} \right.$$

$$\frac{\langle 0,0\rangle}{Q1}, \frac{\langle 1,4\rangle}{Q2}, \frac{\langle 2,7\rangle}{Q3}, \frac{\langle 3,5\rangle}{Q4}, \frac{\langle 4,2\rangle}{Q5}, \frac{\langle 5,6\rangle}{Q6}, \frac{\langle 6,1\rangle}{Q7}, \frac{\langle 7,3\rangle}{Q8} \quad \rightarrow \frac{\langle i, Col\,[i]\rangle}{Q_{i+1}}$$

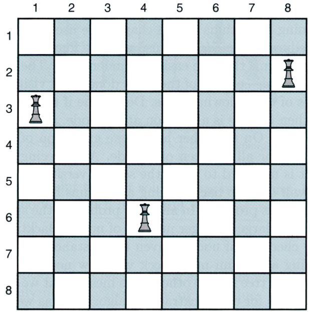
- **Promising function :** For the n-Queens problem, function promising must return false if a node and any of the node's ancestors place queens in the same column or diagonal.

- Condition-1 (check two queens are in the same column). let col(i) be the column where the queen in the i_{th} row is located, then to check whether the queen in the k_{th} row is in the same column

$$col[i] = col[k]$$

- Condition-2 (check two queens are in the same diagonal)

$$|col(i) - col(k)| = |i - k|$$



- Examples:

$$|col[6] - col[3]| = |6 - 3|$$
$$|4 - 1| = |3|$$

$$|col[6] - col[2]| = |6 - 2|$$
$$|4 - 8| = |4|$$

n-Queens Problem (cont...)

Algorithm

```
procedure QUEENS(int i)
    int j
    if (promising(i)) then
        if if(i == n) then
            print col[1] through col[n]
        else
            for (j = 1; j <= n; j++) do
                col[i + 1] = j
                queens(i + 1)
            end for
        end if
    end if
end procedure
```

Promising Function : Pseudo-code

```
procedure PROMISING(int i)
    int k
    bool switch
    k = 1
    switch = true
    while (k < i && switch == true) do

        if
((col[i] == col[k]) || (|col[i] − col[k]| == |i − k|)
then
            switch = false
        end if
        k=k+1
    end while
    return switch
end procedure
```

Analysis of Backtracking Algorithm for n-Queens Problem

- Backtracking is used to avoid unnecessary checking of nodes.
- The number of nodes as a function of n, the number of queens
  - at top root node =1
  - at level 1 =n
  - at level 2= n^2
  - ....
  - at level $n = n^n$

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

- for n=8, 8-queens problem

$$\frac{8^{8+1} - 1}{8 - 1} = 19,173,961 \ \ nodes$$

- A straightforward way to determine the efficiency of the algorithm is to actually run the algorithm on a computer and count how many nodes are checked. Table 1 shows the results for several values of n

n-Queens Problem (cont...)

Table 1: An illustration of how much checking is saved by backtracking in the n-Queens problem

n	Algorithm-1(DFS)	Backtracking	Promising nodes
4	341	61	17
8	19,173,961	15,721	2057
12	9.73×10^{12}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	3.78×10^{8}	2.74×10^{7}

- Given two instances with the same value of n, a backtracking algorithm may check very few nodes for one of them but the entire state space tree for the other.
- This means that we cannot compute time complexities for backtracking algorithms as we did for the algorithms in the previous chapters.
- Backtracking algorithms are analyzed by using the **Monte Carlo technique.**
- This technique enables us to determine whether we can expect a given backtracking algorithm to be efficient for a particular instance.

Problem definition

- In the Sum-of-Subsets problem, there are n positive integers (weights) w_i and a positive integer W.
- The goal is to find all subsets of the integers that sum to W as shown in the Example

Suppose that $n = 5$, $W = 21$, and

$$w_1 = 5 \qquad w_2 = 6 \qquad w_3 = 10 \qquad w_4 = 11 \qquad w_5 = 16.$$

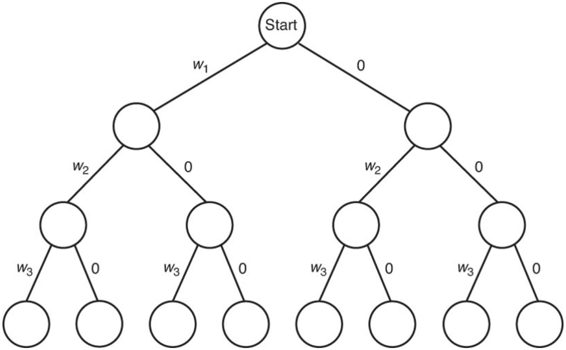
Because

$$\begin{aligned} w_1 + w_2 + w_3 &= 5 + 6 + 10 = 21, \\ w_1 + w_5 &= 5 + 16 = 21, \text{and} \\ w_3 + w_4 &= 10 + 11 = 21, \end{aligned}$$

the solutions are $\{w_1, w_2, w_3\}$, $\{w_1, w_5\}$, and $\{w_3, w_4\}$.

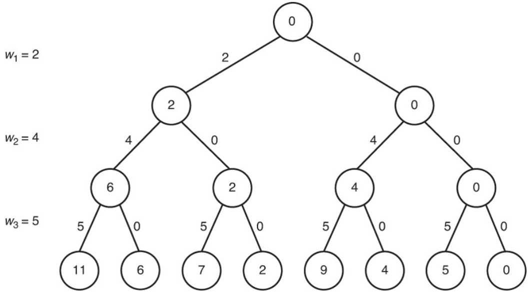
- The Sum-of-Subsets problem can be represented as search space tree as shown in Figure (the tree in this figure is for only three weights)

- We go to the left from the root to include w_1, and we go to the right to exclude w_1



The Sum-of-Subsets Problem (cont...)

- Example: Consider the problem instance: $n = 3$, $W = 6$, and $\{w_1 = 3,\ w_2 = 4,\ w_3 = 5\}$
- A state space tree for this problem is shown in the Figure
- At each node, we have written the sum of the weights that have been included up to that point.
- Therefore, each leaf contains the sum of the weights in the subset leading to that leaf.
- The second leaf from the left is the only one containing a 6. Because the path to this leaf represents the subset w1, w2, this subset is the only solution.



Promising function

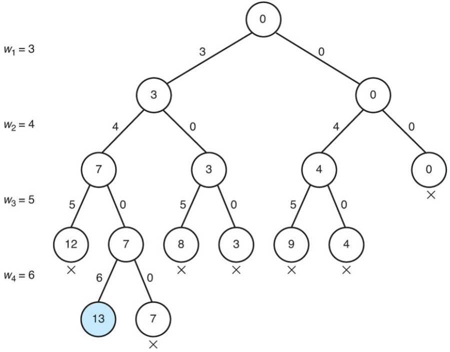
- Condition-1: Let *weight* be the sum of the weights that have been included up to a node at level i. if we include the next element then w_{i+1} will be included in the *weights*. Then the following condition should be satisfied

$$weight + w_{i+1} > W$$

- *total* is defined as the remaining weight. Its initial value will be the total weight of all elements i.e. $total = \sum_{j=1}^{n} w_j$. It will decrease while elements are including in *include* array.
- Condition-2: if the *weight* could never become equal to W then we will have the following condition

$$weight + total < W$$

- The pruned state space tree produced using backtracking for problem instance: $n = 4$, $W = 13$, and $\{w_1 = 3,\ w_2 = 4,\ w_3 = 5,\ w_4 = 6\}$



1: **procedure** Promising(int i)
2: **return** $(weight + total) \geq W$ *and* $(weight == W || (weight + w[i+1]) \leq W)$
3: **end procedure**

The Sum-of-Subsets Problem (cont...)

The Backtracking Algorithm for the Sum-of-Subsets Problem

- **Problem:** Given n positive integers (weights) and a positive integer W, determine all combinations of the integers that sum to W.
- **Inputs:** positive integer n, sorted (nondecreasing order) array of positive integers w indexed from 1 to n, and a positive integer W.
- **Outputs:** all combinations of the integers that sum to W.
- **Pseudo-code**

```
procedure SUMOFSUBSETS(int i, int weight, int total)
    if (promising(i)) then
        if (weight == W) then
            print include[1] through include[n]
        else
            include[i + 1] = "yes"                                    ▷ includew[i + 1].
            sumofsubsets(i + 1, weight + w[i + 1], total − w[i + 1])
            include[i + 1] = "no"                                     ▷ Do not includew[i + 1].
            sumofsubsets(i + 1, weight, total − w[i + 1]);
        end if
    end if
end procedure
```

Graph Coloring (mcoloring) Problem

Problem definition

- 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.
- We usually call the $m - Coloring$ problem a unique problem for each value of m.

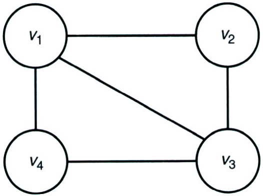


Figure 4: Graph for which there is no solution to the 2-Coloring problem. A solution to the 3-Coloring problem for this graph is shown in Table

Table 2: solution to the 3-Coloring

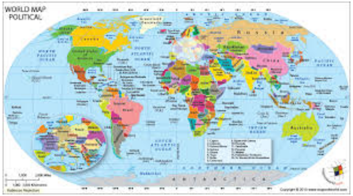
Vertex	Color
v1	color 1
v2	color 2
v3	color 3
v4	color 2

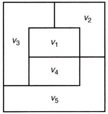
- There are a total of six solutions to the $3 - Coloring$ problem for this graph. However, the six solutions are only different in the way the colors are permuted.
- For example, another solution is to color $v1$ color 2, $v2$ and $v4$ color 1, and $v3$ color 3.

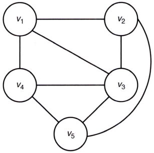
Graph Coloring (mcoloring) Problem (cont...)

Planar Graph and Maps

- An important application of graph coloring is the coloring of maps
- A graph is called **planar** if it can be drawn in a plane in such a way that no two edges cross each other
- To every map there corresponds a planar graph
- Each region in the map is represented by a vertex.
- If one region is adjacent to another region, we join their corresponding vertices by an edge







- The m-Coloring problem for planar graphs is to determine how many ways the map can be colored, using at most m colors, so that no two adjacent regions are the same color.

- How many colors, vertices, edges, countries and their adjacent countries?

State Space Tree for $m - Coloring$

- A straightforward state space tree for the $m - Coloring$ problem is one in which each possible color is tried for vertex $v1$ at level 1, each possible color is tried for vertex $v2$ at level 2, and so on until each possible color has been tried for vertex vn 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 vertices are the same color.
- 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 first solution is found at the shaded node.

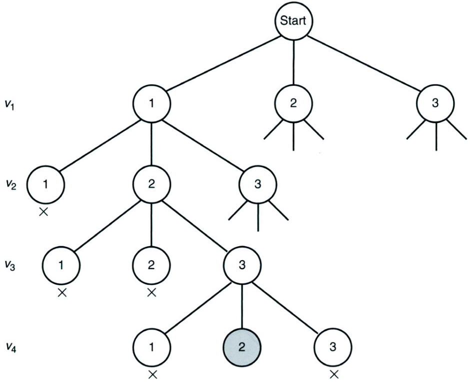


FIGURE 5: A portion of the pruned state space tree produced using backtracking to do a 3-coloring of the graph in Figure 4. The first solution is found at the shaded node. Each nonpromising node is marked with a cross.

Graph Coloring (mcoloring) Problem (cont...)

Algorithm for $m - Coloring$ problem

- **Problem:** Determine all ways in which the vertices in an undirected graph can be colored, using only m colors, so that adjacent vertices are not the same color.
- **Inputs:** positive integers n and m, and an undirected graph containing n vertices. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where $W[i][j]$ is true if there is an edge between i_{th} vertex and the j_{th} vertex and false otherwise
- **Outputs:** all possible colorings of the graph, using at most m colors, so that no two adjacent vertices are the same color. 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 i_{th} vertex.

Pseudo-code

```
procedure MCOLORING(int i)
    int color
    if (promising(i)) then
        if if(i == n) then
            print vcolor[1] through vcolor[n]
        else
                                        ▷ Children of the vortex i
            for (color = 1; color <= m; color ++) do
                vcolor[i + 1] = color
                mcoloring(i+1)
            end for
        end if
    end if
end procedure
```

Graph Coloring (mcoloring) Problem (cont...)

- 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.

- **Pseudo-code: for Promising function**

```
procedure Promising(int i)                          ▷ Procedure for Function Promising
    int j                                           ▷ variable k of type integer
    bool switch                                     ▷ variable switch of type boolean
    j = 1
    switch = true
    while (j ≤ i − 1 && switch) do
                                                    ▷ Check if an adjacent vertex is already this color
        if (W[i][j] && vcolor[i] == vcolor[j]) then
            switch = false
        end if
        j = j + 1
    end while
    return switch
end procedure
```

Sudoku: Puzzle

Problem definition

- Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.

							1	2
				3	5			
			6				7	
7						3		
			4			8		
1								
			1	2				
	8						4	
	5					6		

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

Figure 6: Challenging Sudoku puzzle (left) with solution (right)

- Brute-Force Approach: (Expensive) The naive approach is to generate all possible configurations of numbers from 1 to 9 to fill the empty cells. Try every configuration one by one until the correct configuration is found, i.e. for every unassigned position fill the position with a number from 1 to 9. After filling all the unassigned position check if the matrix is safe or not. If safe print else recurs for other cases.

Sudoku: Puzzle (cont...)

- Backtracking lends itself nicely to the problem of solving Sudoku puzzles.

- Algorithm

1. Find row, col of an unassigned cell
2. If there is none, return true
3. For digits from 1 to 9
    - if there is no conflict for digit at row,col assign digit to row,col and recursively try fill in rest of grid
    - if recursion successful, return true
    - if not successful, remove digit and try another
4. if all digits have been tried and nothing worked, return false to trigger backtracking

Summary

1. State Space Search Technique
   - State Space Tree
   - Depth First Search
   - Breadth First Search
2. Backtracking
3. n-Queens Problem
4. The Sum-of-Subsets Problem
5. Graph Coloring (mcoloring) Problem
6. Sudoku: Puzzle