SC1007
Data Structures and Algorithms

Backtracking

Dr Liu Siyuan (syliu@ntu.edu.sg)

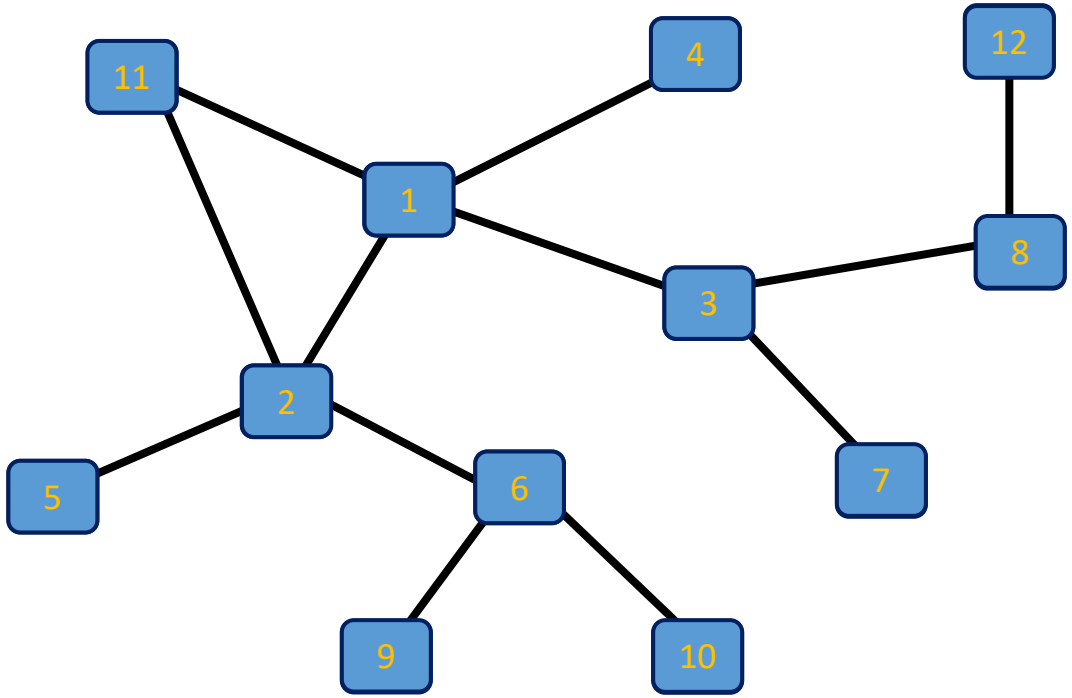
N4-02C-117a

Office Hour: Mon & Wed 4-5pm

Week	Date	1830-1930	1930-2030	2030-2130	Venue
7	29-Feb-24	T3 (BT & BST)	Lab 4 (BST)	Lab 4 (BST)	SW2
Recess	07-Mar-24		**Lab Test 1**		SW2
8	14-Mar-24	Analysis of Algorithm (AA)		Hash Table	TR+3
9	21-Mar-24	Graph Representation, BFS, DFS	T4 (AA)	Lab 5 – Hash Table	SW2
10	28-Mar-24	Backtracking, Permutation	Lab 6 - Graph	Lab 6 - Graph	SW2
11	04-Apr-24	Dynamic Programming	Lab 7 - Backtracking	Lab 7 - Backtracking	SW2
12	11-Apr-24	Matching Problem	Lab 8 – Dynamic Programming	Lab 8 – Dynamic Programming	SW2
13	18-Apr-24	T5 (Hash Table)	T6 (Graph)		TR+3
	25-Apr-24	**Lab Test 2 + Final Quiz, Makeup Lab Test 2 + Quiz (26/04/2024 Morning)**			SW2

Depth First Search (DFS)

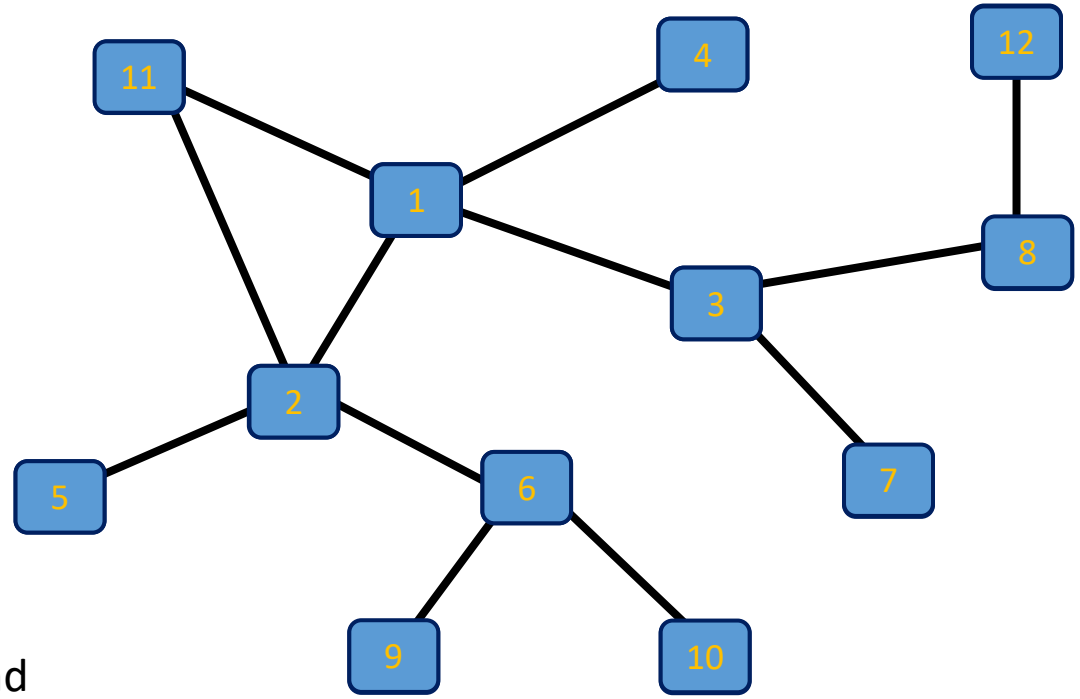
- Work similar to preorder traversal of the trees
- DFS systematically explores along a path from vertex v as deeply into the graph as possible before backing up.

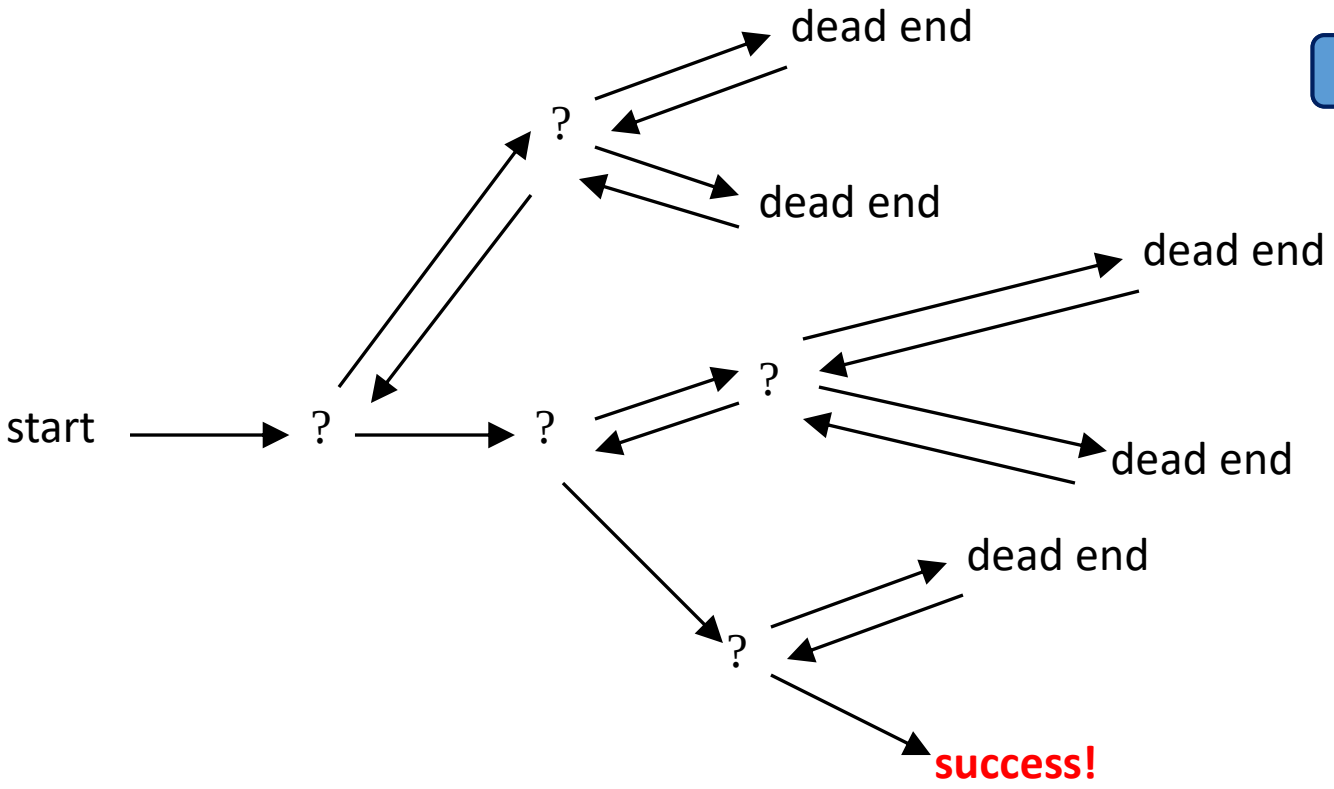


Backtracking

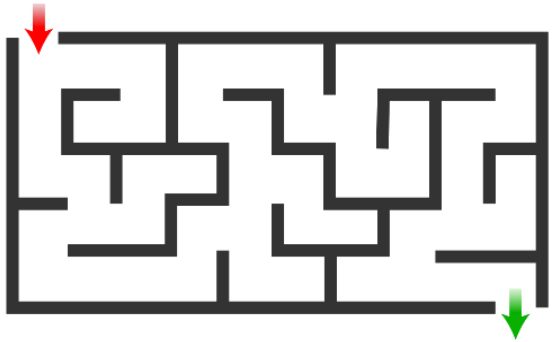
- Suppose you make a series of *decisions,* among various *choices,* where:
  - You don’t have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”

Backtracking





Solving a maze

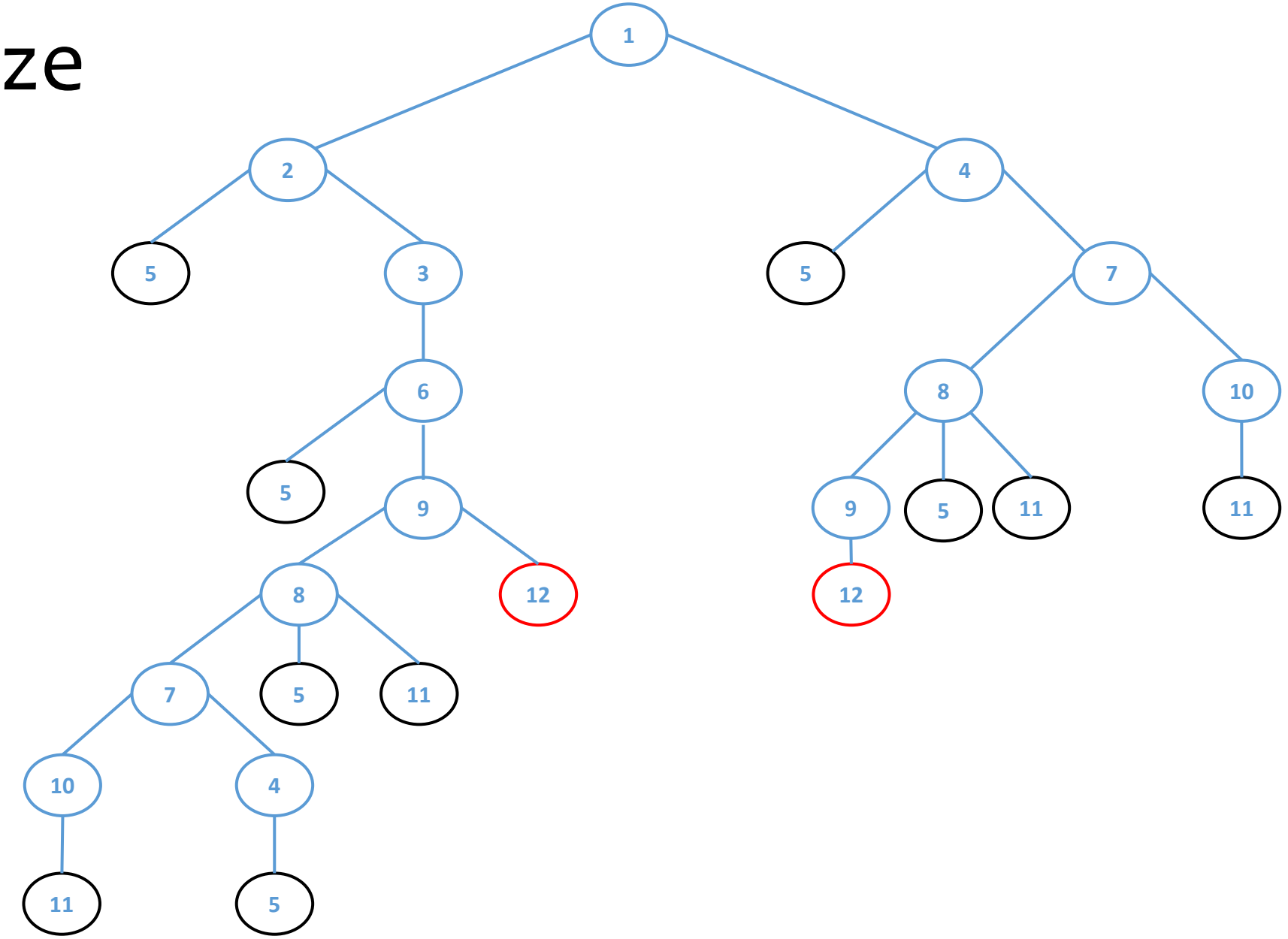


- Given a maze, find a path from start to finish
- At each intersection, you have to decide:
  - Go straight
  - Go left
  - Go right
- You don’t have enough information to choose correctly
  - Each choice leads to another set of choices
  - One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

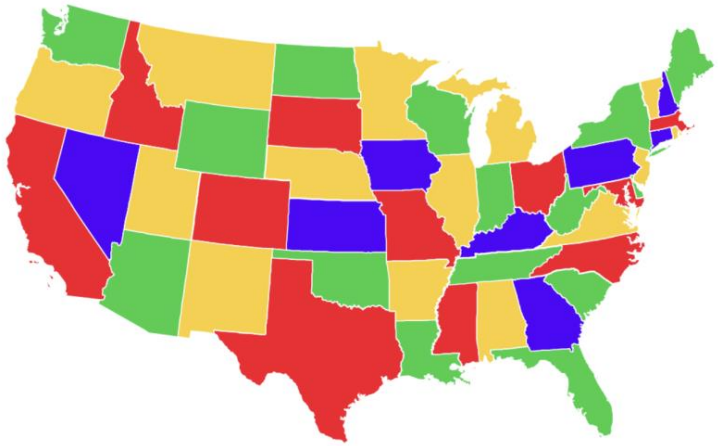
Solving a maze

10	11	12 (F)
7	8	9
4	5	6
1 (S)	2	3

We can only know it is the finish when we arrive at the cell.



Coloring a map



- You wish to color a map with not more than m colors
- Adjacent regions must be in different colors
- You don’t have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking

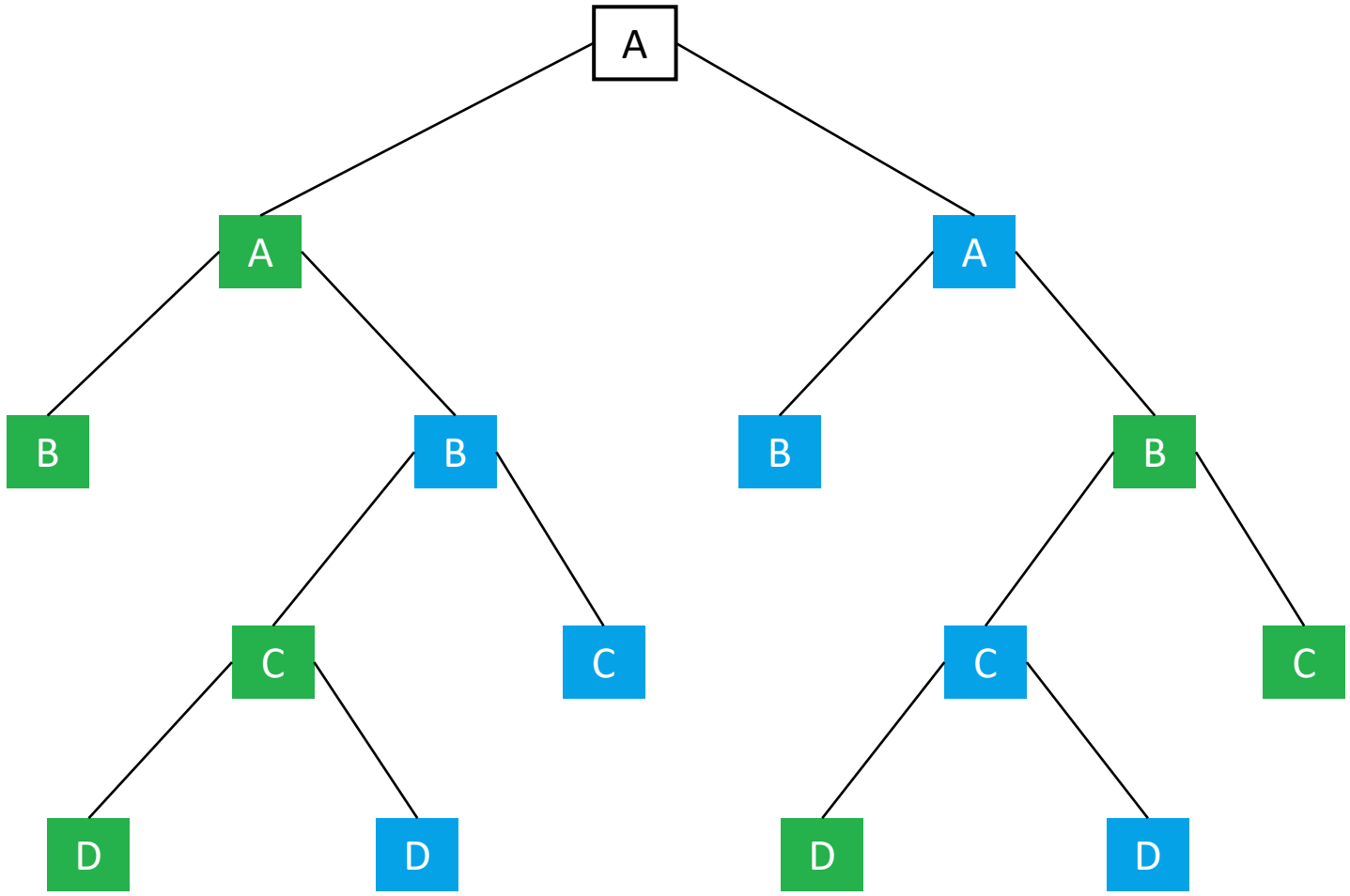
Coloring a map

A	B
D	C

Colors used:

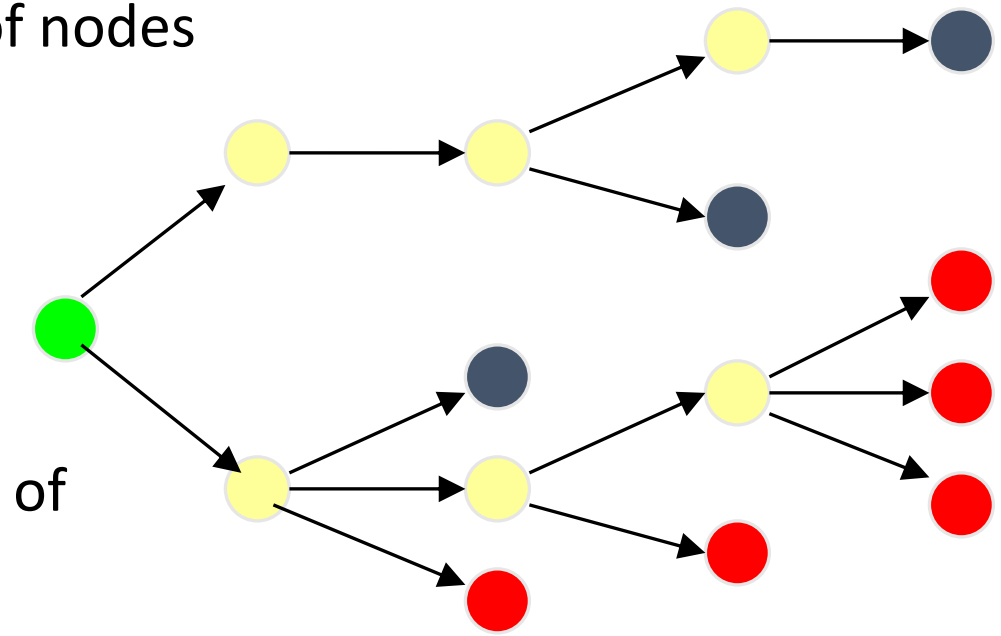
A	B
D	C

A	B
D	C



The backtracking algorithm

A tree is composed of nodes



There are three kinds of nodes:

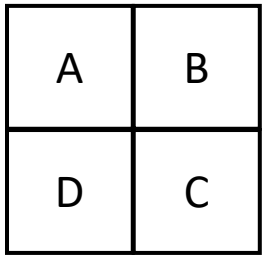
- The (one) root node
- Internal nodes
- Goal nodes
- Dead end

Goal nodes and Dead end: Leaf nodes

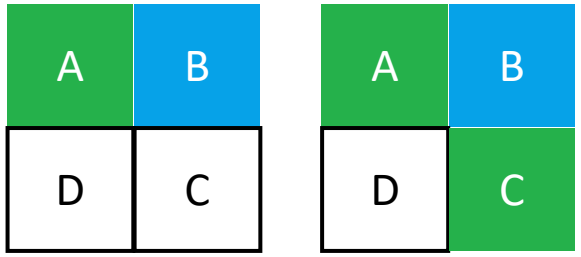
Backtracking can be thought of as searching a tree for a particular “goal” leaf node, or all “goal” leaf nodes

Example: Coloring a map

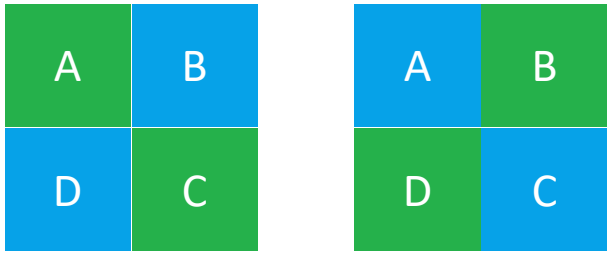
- Root node:



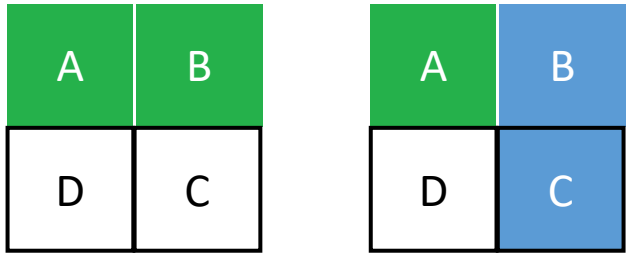
- Internal node, e.g.,



- Goal node



- Dead end, e.g.,

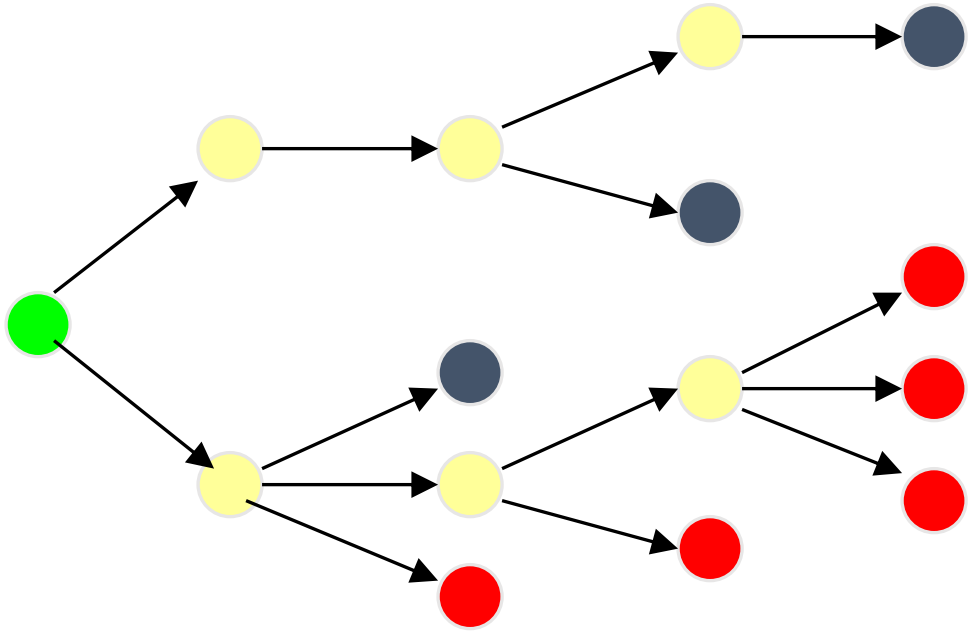


The backtracking algorithm

- Backtracking is simple – we “explore” each node. To “explore” node N:

```
if N is a goal node, return “success”
else if N is a leaf node, return “failure”
for each child C of N:
        explore C
        if C exploration is successful, return “success”
return “failure”
```

```
Backtracking(N)
            if N is a goal node, return “success”
            else if N is a leaf node, return “failure”
            for each child C of N,
                        if Backtracking(C) == “success”
                                    return “success”
            return “failure”
```



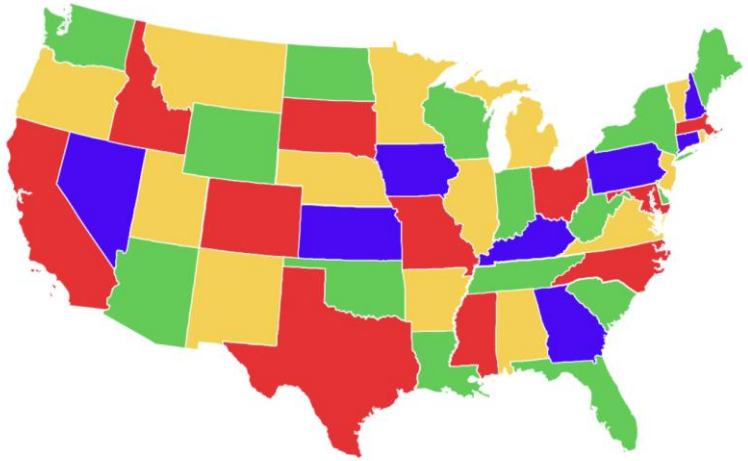
```
Backtracking(N)
        if N is a goal node, return true
        else if N is a leaf node, return "failure"
        for action a = 1 to m //the actions that can be taken at current node N
                If taking a leads to a child C of N //taking action will arrive at a child
                        move forward to child C
                        if (Backtracking(C)==true) return true
                        reverse whatever you have done earlier //go back to N
        return false;
```

10	11	12 (G)
7	8	9
4	5	6
1 (S)	2	3

```
Backtracking(N)
        if N is a goal node, return true
        for action a = 1 to m //the actions that can be taken at current node N
                if (a is safe) //taking action will arrive at a child and will not lead to a dead end
                        move forward to child C
                        if (Backtracking(C)==true) return true
                        reverse whatever you have done earlier //go back to N
        return false;
```

Example: Coloring a Map

- Input format:
  - Number of regions: |V|
  - 2D adjacency matrix representation graph[V][V]
  - Number of colors: M
- Output format:
  - array color[V], element is a number in [1, M]
  - Each element is initialized as 0 (uncolored)
- Naïve solution: check all possible combinations
  - $M^{|V|}$ combinations
  - If M= 4 and |V| = 50 → billion years to run 😵



```
Backtracking(N)
        if N is a goal node, return true
        for action a = 1 to m //the actions that can be taken at current node N
                if (a is safe) //taking action will arrive at a child and will not lead to a dead end
                        move forward to child C
                        if (Backtracking(C)==true) return true
                        else reverse whatever you have done earlier //go back to N
        return false;
```

```
ColorMap(M colors, color[V], current region v)
        if all regions are colored, return true //goal node
        for c= 1 to M           //the actions that can be taken at current node
                if (color current region v as c is safe) //taking action is not a dead end
                        color[v] = c //move forward
                        if (ColorMap(colors, color, v+1) == true) //explore child node
                                return true
                        color[v]=0 //reverse what have been done
        return false;
```

Example: Coloring a Map

1. Start with an initial map
2. Select an uncolored region and try to assign it a color.
3. Check if the color assignment is valid, i.e., check if no adjacent regions have the same color.
4. If the color assignment is valid, repeat steps 2-3.
5. If the color assignment is not valid, remove the color assignment and try a different color assignment for that region.
6. If all possible color assignments for a region have been tried and none of them are valid, remove the color assignment for previous region and try a different color assignment for that previous region.
7. Repeat steps 2-6 until all regions have been colored or until it is determined that a valid coloring cannot be found.

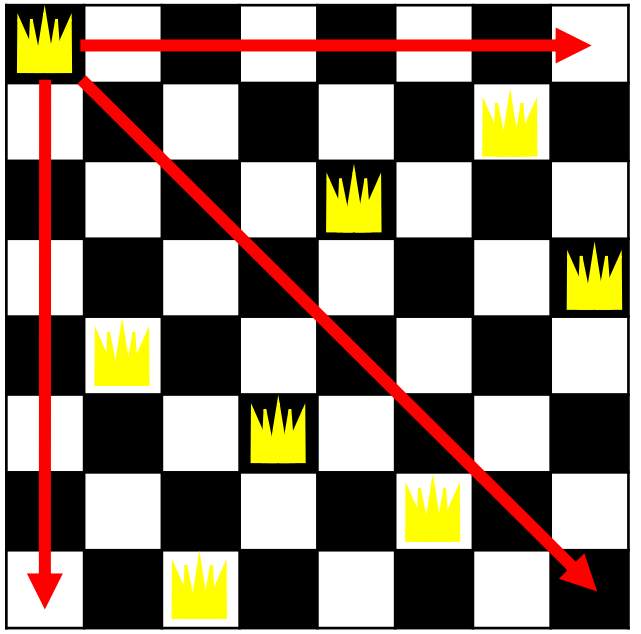
```
bool mColoring(int colors, int color[], int vertex){
   if (vertex == V) //when all vertices are considered
      return true;
   for (int col = 1; col <= colors; col++) {
      if (isValid(vertex,color, col)) { //check whether color col is valid or not
         color[vertex] = col;
         if (mColoring (colors, color, vertex+1) == true) //go for additional vertices
            return true;
         color[vertex] = 0;
      }
   }
   return false; //when no colors can be assigned
}

bool isValid(int v,int color[], int c){   //check whether putting a color valid for v
   for (int i = 0; i < V; i++)
      if (graph[v][i] && c == color[i])
         return false;
   return true;
}

int main(){
   int colors = 3; // Number of colors
   int color[V]; //make color matrix for each vertex
   for (int i = 0; i < V; i++)
      color[i] = 0; //initially set to 0
   if (mColoring(colors, color, 0) == false) { //for vertex 0 check graph coloring
      printf("Solution does not exist.");
   }
   printf("Assigned Colors are: \n");
   for (int i = 0; i < V; i++)
      printf("%d ", color[i]);
   return 0;
}
```

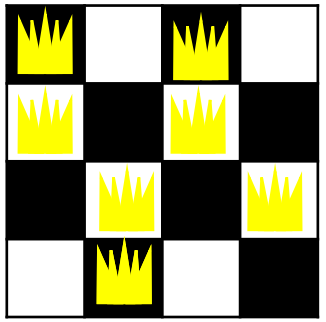
```
int V=4;
bool graph[V][V] = {
   {0, 1, 1, 0},
   {1, 0, 1, 1},
   {1, 1, 0, 1},
   {0, 1, 1, 0},
};
```

Example: The Eight Queens Problem



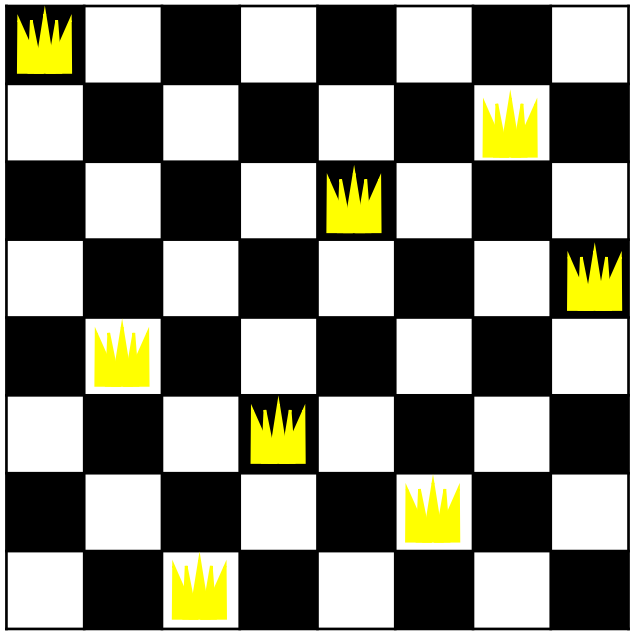
- A chessboard has 8 rows and 8 columns
- A queen can move within its row or its column or along its diagonal
- Place 8 queens on the board
  - No queen can attack any other queen in a move
- Exhausting search will take $\binom{64}{8} = 4.43$ billion ways
- Each row can contain exactly one queen: $8! = 40{,}320$
- Can be generalized to N queens
- Better algorithm?

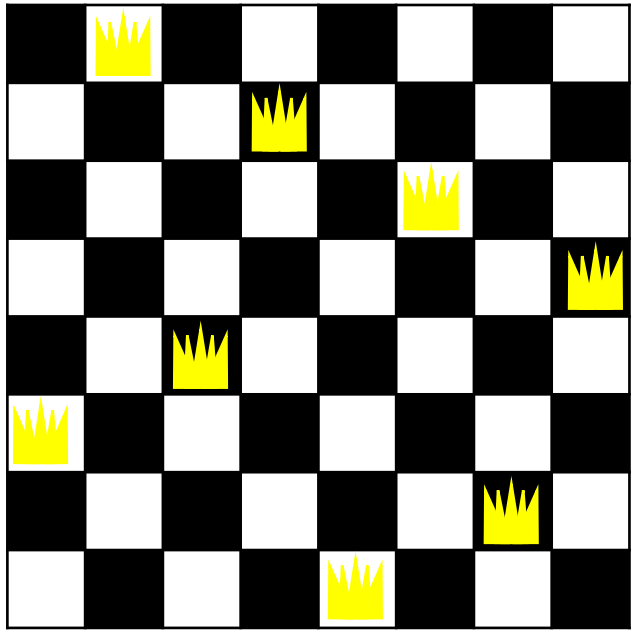
Backtracking for N Queens

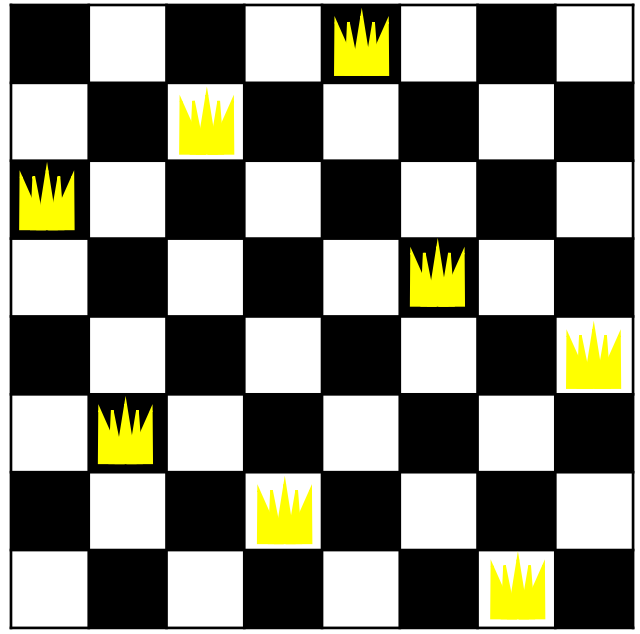


- Starts by placing a queen on the top left corner of the chess board.
- Places a queen on the second column and moves her until a place where she cannot be hit by the queen on the first column.
- Places a queen on the third column and moves her until she cannot be hit by either of the first two queens and so on.
- If there is no place for the i^{th} queen, the program backtracks to move the $(i - 1)^{\text{th}}$ queen.
- If the $(i - 1)^{\text{th}}$ queen is at the end of the column, the program removes the queen and backtracks to the $(i - 2)$ column and so on.
- If the current column is the last column and a safe place has been found for the last queen, then a solution to the puzzle has been found.

Backtracking for N-Queens



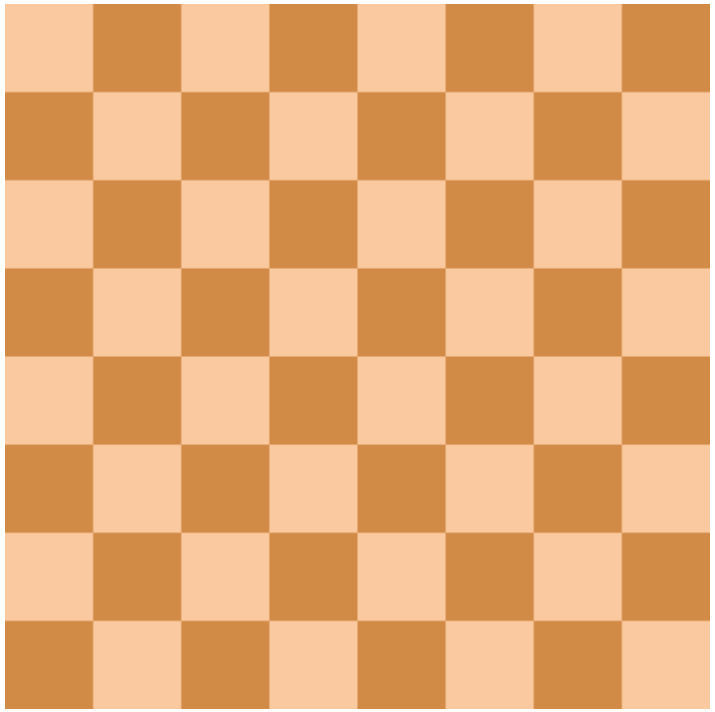




- Once found a solution, store it.
- If the current column is the first column and its queen is being moved off the board then all possible configurations have been examined, all solutions have been found, and the algorithm terminates.
- This puzzle has **92** solutions.

N-Queens Problem

n	Possible Solutions
4	2
5	10
6	4
7	40
8	92
10	724
12	14,200
15	2,279,184
20	39,029,188,884



```
Backtracking(N)
        if N is a goal node, return true
        for action a = 1 to m //the actions that can be taken at current node N
                if (a is safe) //taking action will arrive at a child and will not lead to a dead end
                        move forward to child C
                        if (Backtracking(C)==true) return true
                        else reverse whatever you have done earlier //go back to N
        return false;
```

```
NQueen(board[N][N], current_column)
        if current_column >=N return true //goal node
        for row i = 0 to N-1 //the actions that can be taken at current node
                if (putting queen on row i is safe) //taking action is not a dead end
                        board[i][current_column] = 1 //move forward, put queen on row i
                        if (NQueen(board, current_column+1) == true) //explore child node
                                return true
                         board[row][current_column] = 0 //reverse what have been done
        return false;
```

```
bool solveNQ(int board[N][N], int col)
{
    // Base case: If all queens are placed
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on board[i][col]
        if (isSafe(board, i, col)) {

            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place rest of the queens
            if (solveNQ(board, col + 1))
                return true;

            board[i][col] = 0; // BACKTRACK
        }
    }
    return false;
}
```

```
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}


int main()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQ(board, 0) == false) {
        printf("Solution does not exist");
    }
    else printSolution(board);

}
```

https://algorithm-visualizer.org/

Backtracking Algorithm Problems

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

- Maze problem
- Coloring Problem
- N Queen Problem
- Sudoku
- Hamiltonian Cycle
  - Hamiltonian Path is a path that visits each vertex exactly once in an undirected graph.
  - If the last vertex has an edge back to the first vertex, then a Hamiltonian Cycle exists.

Summary

- If trying all possible permutations, cost is very high
- Usually not all permutations are worth to investigate
  - You may terminate it in the halfway of your searching
- It is a “not-very-smart” approach

```
Backtracking(N)
        if N is a goal node, return true
        for action a = 1 to m //the actions that can be taken at current node N
                if (a is safe) //taking action will arrive at a child and will not lead to a dead end
                        move forward to child C
                        if (Backtracking(C)==true) return true
                        else reverse whatever you have done earlier //go back to N
        return false;
```

Problem for you to think about

- 3 missionaries and 3 cannibals must cross a river using a boat which can carry at most 2 people, subject to the constraints that, if there are missionaries present on either of the banks, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board. Find all the solutions to cross the river.

- Advanced version: m missionaries and n cannibals cross a river using a boat which can carry at most k people.

- Another version: a farmer must transport a fox, goose and bag of beans from one side of a river to another using a boat which can only carry one item in addition to the farmer, subject to the constraints that the fox cannot be left alone with the goose, and the goose cannot be left alone with the beans. Find the solutions to cross the river.