

Backtracking

Backtracking

- Suppose you have to **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 systematic way of trying out various sequences of decisions, until you find one that “works”

2



3



4

- It's called 'Bhool Bhulaiya' because there are more than a 1000 pathways stretching to the length of the hall, but only 24 of these are the right ones i.e. only 24 of these pathways will take you from one end of the hall to the other; and it's quite tough to distinguish them as one can get lost amidst these paths.

5



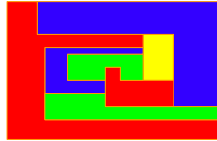
Solving a maze

- Given a maze, find a path from start to finish
- At each intersection, you have to decide between three or fewer choices:
 - 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

8

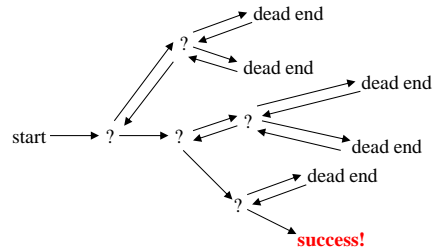
Coloring a map

- You wish to color a map with not more than four colors
 - red, yellow, green, blue
- Adjacent countries 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



9

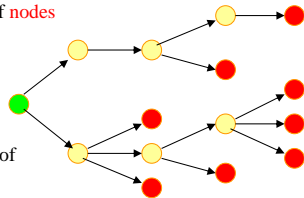
Backtracking



10

Terminology

A tree is composed of **nodes**



There are three kinds of nodes:

- The (one) **root** node
- Internal** nodes
- Leaf** nodes

Backtracking can be thought of as searching a tree for a particular "goal" leaf node

11

Backtracking

- For some problems, the only way to solve is to check all possibilities.
- Backtracking** is a systematic way to go through all the possible configurations of a search space.

12

Trees

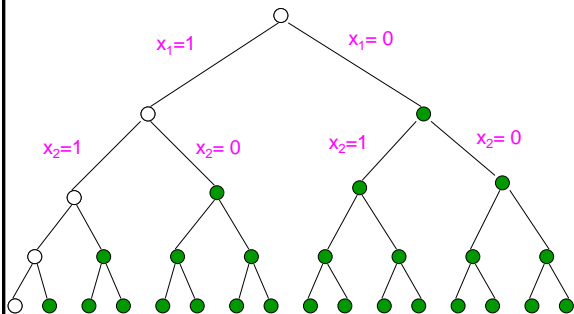
- If we diagram the sequence of choices we make, the diagram looks like a **tree**
- The solution space tree exists only in your mind, **not in the computer**.

13

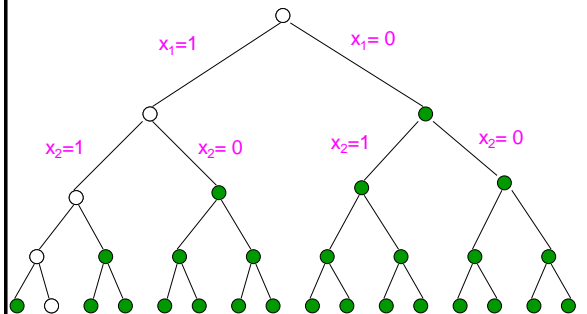
Backtracking

- Search the solution space tree in a **depth-first manner**.
- May be done recursively or use a stack to retain the path from the root to the current node in the tree.

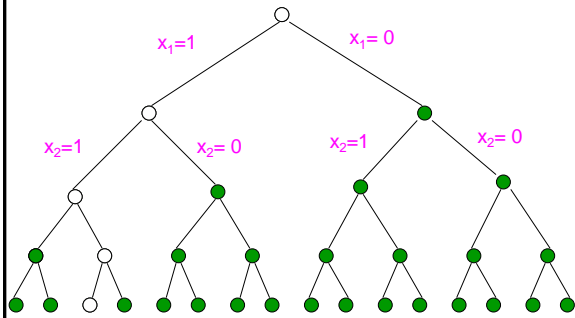
Backtracking Depth-First Search



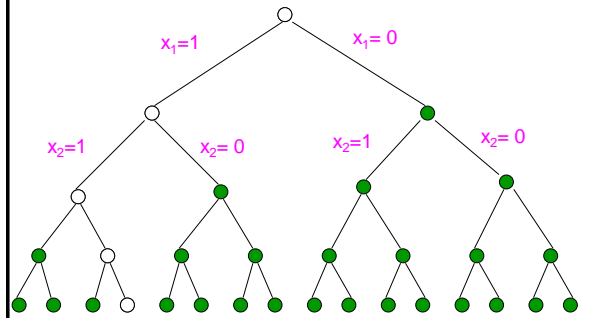
Backtracking Depth-First Search



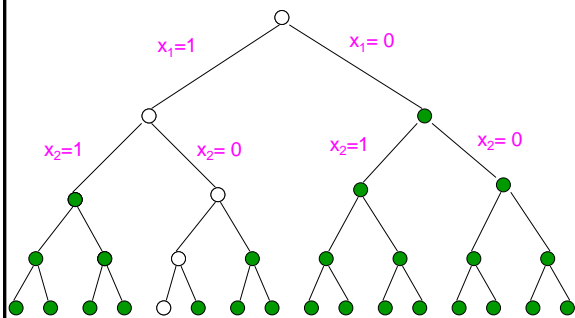
Backtracking Depth-First Search



Backtracking Depth-First Search



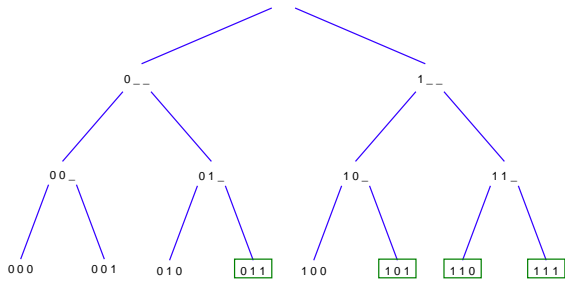
Backtracking Depth-First Search



Backtracking

- Problem:
Find out all 3-bit binary numbers for which the sum of the 1's is greater than or equal to 2.
- The only way to solve this problem is to check all the possibilities: (000, 001, 010,, 111)
- The 8 possibilities are called the search space of the problem. They can be organized into a tree.

Backtracking: Illustration



21

The backtracking algorithm

- To “explore” node N:
 1. If N is a goal node, return “success”
 2. If N is a leaf node, return “failure”
 3. For each child C of N,
 - 3.1. Explore C
 - 3.1.1. If C was successful, return “success”
 4. Return “failure”

22

Graph Coloring Problem

- Assign colors to the nodes of a graph so that no adjacent nodes share the same color
 - nodes are adjacent if there is an edge for node i to node j .

23

Graph Coloring

- **m-Coloring Problem**

Find all ways to color an undirected graph using at most **m** color, so that **no** two adjacent vertices are the same color.

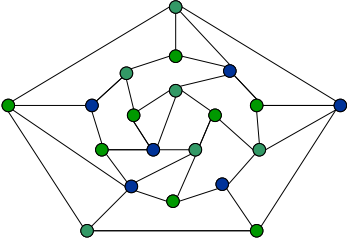
Small Example (4-node graph)

```
graph TD; A(( )) --- B(( )); A --- C(( )); A --- D(( )); B --- E(( )); B --- F(( )); C --- G(( )); C --- H(( )); D --- I(( )); D --- J(( ))
```

Small Example (all possible 3-coloring)

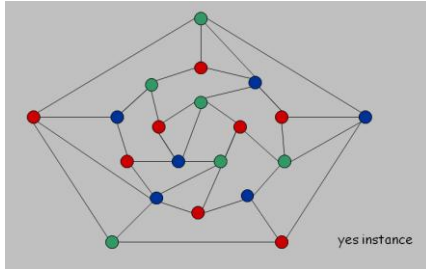
3-Colorability

- **3-COLOR:** Given an undirected graph G does there exist a way to color the nodes **red, green, and blue** so that no adjacent nodes have the same color?



Yes

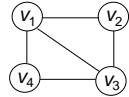
28



29

Example

Find all ways to color the 4 vertices graph.



Sol : If $m = 2$, no solution.

If $m = 3$, $(v_1: \text{color 1}), (v_2, v_4: \text{color 2}), (v_3: \text{color 3})$;

$(v_1: \text{color 1}), (v_2, v_4: \text{color 3}), (v_3: \text{color 2})$;

$(v_1: \text{color 2}), (v_2, v_4: \text{color 1}), (v_3: \text{color 3})$;

$(v_1: \text{color 2}), (v_2, v_4: \text{color 3}), (v_3: \text{color 1})$;

$(v_1: \text{color 3}), (v_2, v_4: \text{color 1}), (v_3: \text{color 2})$;

$(v_1: \text{color 3}), (v_2, v_4: \text{color 2}), (v_3: \text{color 1})$.

If $m = 4$, $(v_i: \text{color } i)$, for $i = 1$ to 4, total $4! = 24$.

$m < N$, the number of vertices.

Pseudo code

Problem : Find all ways for the m -Coloring Problem.

Inputs : positive integer n , m , and an adjacent matrix W .

Output : an array $vcolor$, where $vcolor[i]$ is the color of vertex i .

```

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++)
        {
            vcolor[i+1]=color;
            m_coloring(i+1);
        }
}

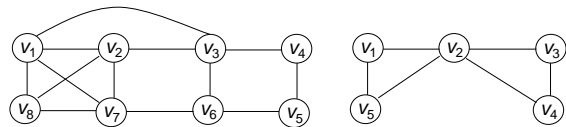
bool promising(index i)
{
    index j;
    bool switch;
    switch = true;
    j = 1;
    while (j < i && switch)
    {
        if (W[i][j] && vcolor[i]==vcolor[j])
            switch = false;
        j++;
    }
    return switch;
}

```

The Hamiltonian Problem

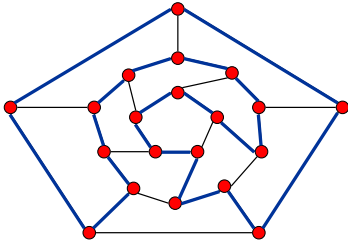
Hamiltonian

A path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex.



Hamiltonian Cycle

- Hamiltonian-cycle: given an undirected graph $G = (V, E)$, does there exist a simple cycle that contains every node in V .

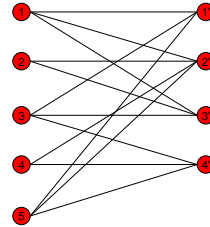


YES

34

Hamiltonian Cycle

- Hamiltonian-cycle: given an undirected graph $G = (V, E)$, does there exist a simple cycle that contains every node in V .



NO

35

Algorithm

Problem : Find all Hamiltonian cycles in the graph

Inputs : an undirected graph with n vertices, and an adjacency matrix W .

Output : an array `index`, where `index[i]` is the index of the i -th vertex on the path.

```
void hamiltonian (index i)
{
    index j;
    if (promising(i))
        if (i == n-1)
            cout << index[0] through index[n-1];
        else
            for (j = 2; j <= n; j++)
                {
                    index[j+1] = j;
                    hamiltonian(i+1);
                }
}

bool promising (index i)
{
    index j;
    bool switch;
    if (i == n-1 && ! W[index[n-1]][index[0]])
        switch = false;
    else if (i > 0 && ! W[index[i-1]][index[i]])
        switch = false;
    else {
        switch = true;
        j = 1;
        while (j < i && switch)
            if (index[j] == index[i])
                switch = false;
            j++;
    }
    return switch;
}
```

Input:

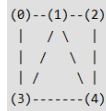
A 2D array `graph[V][V]` where V is the number of vertices in graph and `graph[V][V]` is adjacency matrix representation of the graph. A value `graph[i][j]` is 1 if there is a direct edge from i to j , otherwise `graph[i][j]` is 0.

Output:

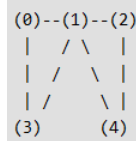
An array `path[V]` that should contain the Hamiltonian Path. `path[i]` should represent the i th vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

37

- A Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}. There are more Hamiltonian Cycles in the graph like {0, 3, 4, 2, 1, 0}



- Following graph doesn't contain any Hamiltonian Cycle.



38

State space tree

- Put the starting vertex at level 0 in the tree, call this the zero'th 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 vertices as the second vertex, and so on.
- You may now backtrack in this state space tree.

39

Backtracking Algorithm

- Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1.
- Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added.
- If we find such a vertex, we add the vertex as part of the solution.
- If we do not find a vertex then we return false.

40

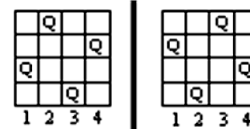
N-Queens Problem

The problem of placing N queens on an NxN chessboard in such a way that **no two of them are "attacking" each other**, is a classic problem used to demonstrate the backtracking method.

Queen can move forward & back, side to side and to its diagonal and anti-diagonals

A simple brute-force method would be to try placing the first queens on the first square, followed by the second queen on the first available square, scanning the chessboard in a row-column manner.

A more efficient backtracking approach is to note that each queen must be in its own column and row.

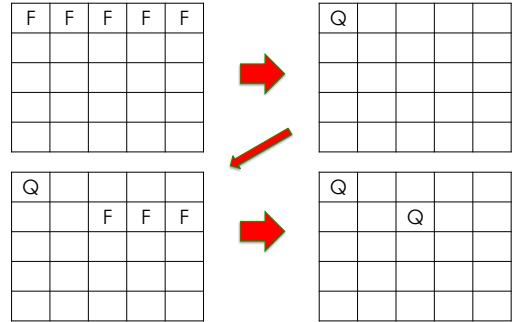


Algorithm

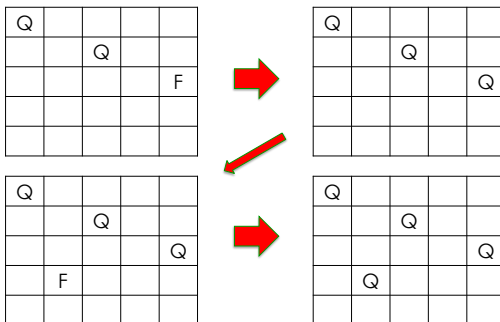
- Place a queen in the top row, then note the column and diagonals it occupies.
- Then place a queen in the next row down, taking care not to place it in the same column or diagonal. Keep track of the occupied columns and diagonals and move on to the next row.
- If no position is open in the next row, we **back track to the previous row and move the queen over to the next available spot in its row** and the process starts over again.

42

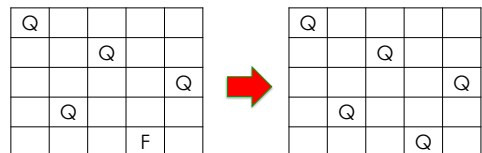
Basic Algorithm (example)



Basic Algorithm (example)



Basic Algorithm (example)



Rat in a Maze

- In walking through a maze, probably walk a path as far as you can go
 - Eventually, reach destination or dead end
 - If dead end, must retrace your steps
 - Loops: stop when reach place you've been before
- Backtracking systematically tries alternative paths and eliminates them if they don't work

Maze Solving Algorithm: findPath(x, y)

1. if (x,y) outside grid, return *false*
2. if (x,y) barrier or visited, return *false*
3. if (x,y) is maze exit, color PATH and return *true*
4. else:
 5. set (x,y) color to PATH ("optimistically")
 6. for each neighbor of (x,y)
 7. if findPath(neighbor), return *true*
 8. set (x,y) color to TEMPORARY ("visited")
 9. return *false*