



Backtracking





A short list of categories

- Algorithm types we will consider include:
 - Simple recursive algorithms
 - ➔ ■ Backtracking algorithms
 - Divide and conquer algorithms
 - Dynamic programming algorithms
 - Greedy algorithms
 - Branch and bound algorithms
 - Brute force algorithms
 - Randomized algorithms



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 methodical way of trying out various sequences of decisions, until you find one that “works”

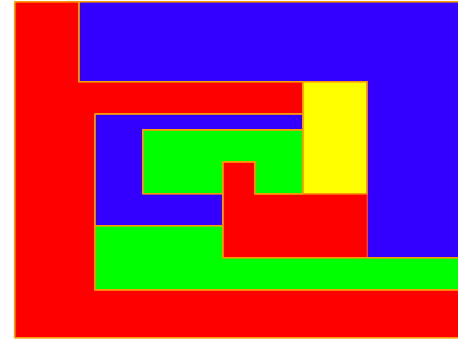


Solving a maze

- 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

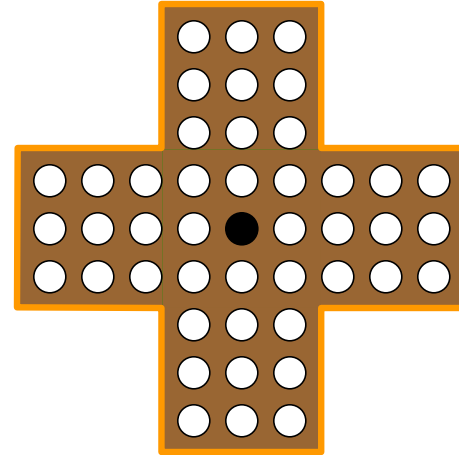
Coloring a map

- 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

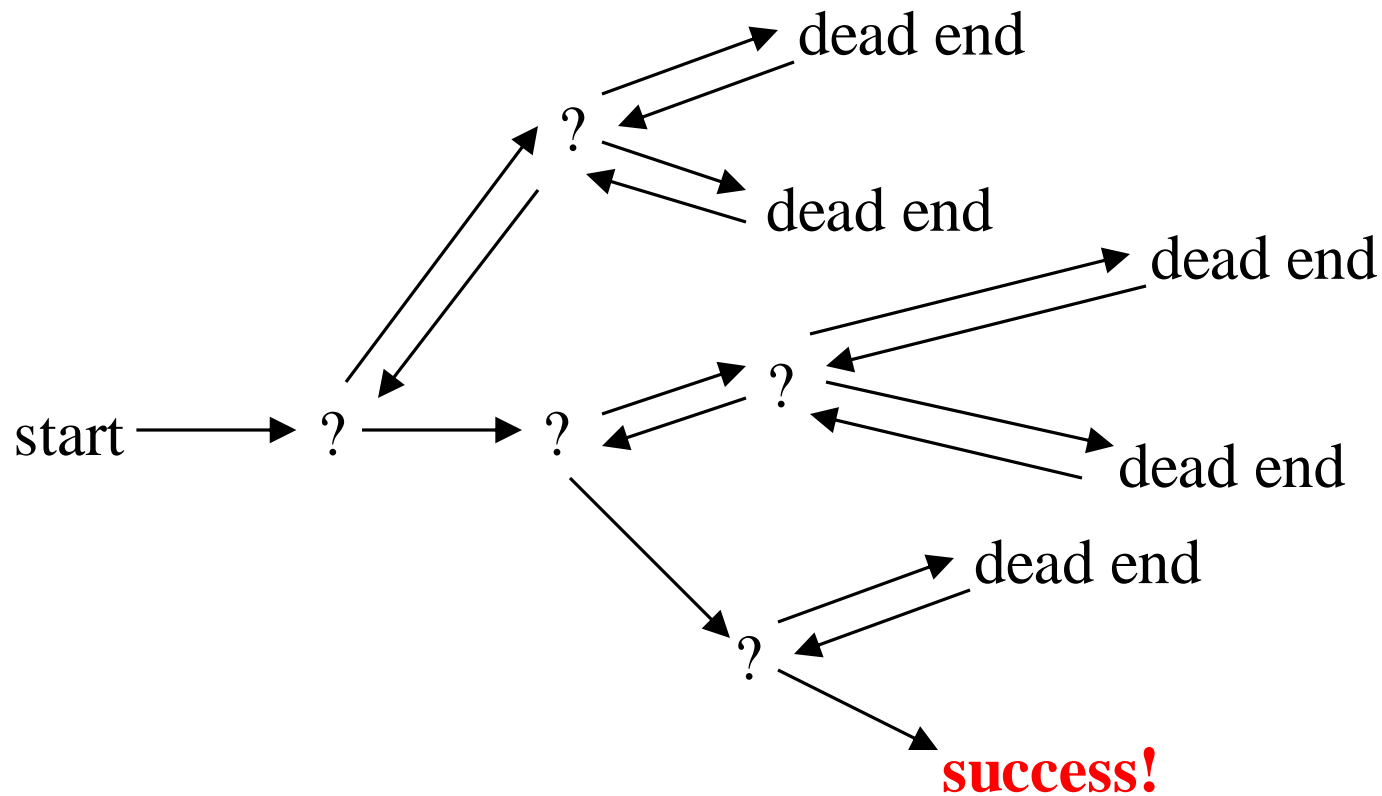


Solving a puzzle

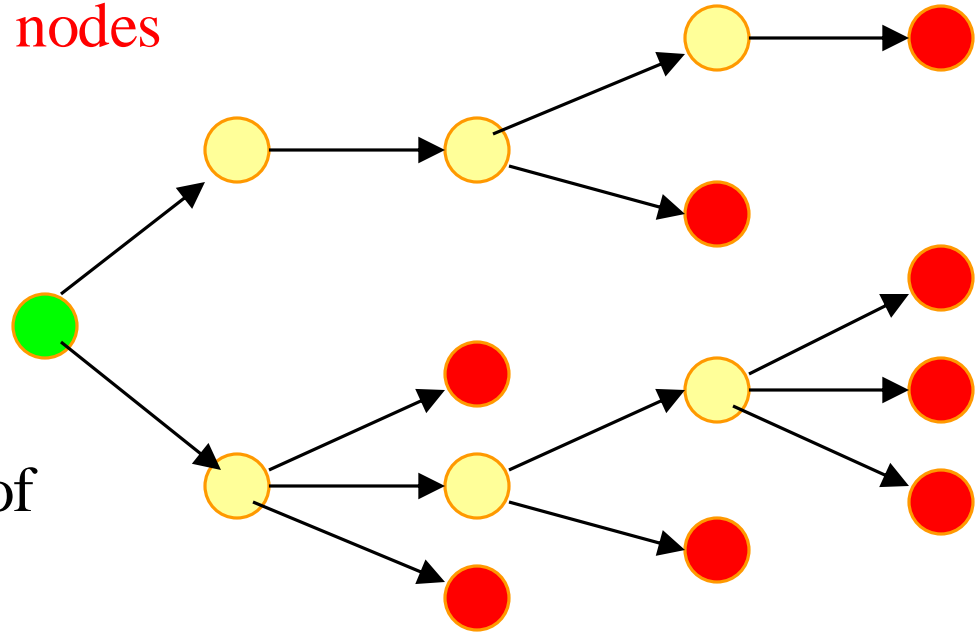
- In this puzzle, all holes but one are filled with white pegs
- You can jump over one peg with another
- Jumped pegs are removed
- The object is to remove all but the last peg
- You don't have enough information to jump correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many kinds of puzzle can be solved with backtracking



Backtracking (animation)



A tree is composed of **nodes**



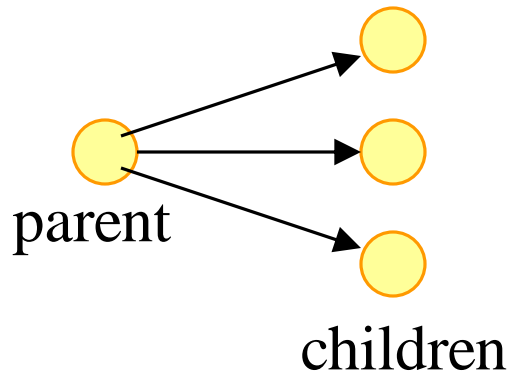
There are three kinds of nodes:

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

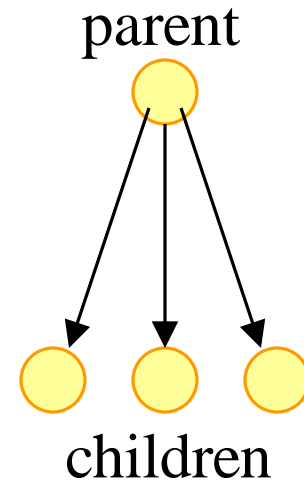
Backtracking can be thought of as searching a tree for a particular “goal” leaf node

Terminology II

- Each non-leaf node in a tree is a **parent** of one or more other nodes (its **children**)
- Each node in the tree, other than the root, has exactly one **parent**



Usually, however,
we draw our trees
downward, with
the root at the top





Real and virtual trees

- There is a type of data structure called a tree
 - But we are **not** using it here
- If we diagram the sequence of choices we make, the diagram looks like a tree
 - In fact, we did just this a couple of slides ago
 - Our backtracking algorithm “sweeps out a tree” in “problem space”



The backtracking algorithm

- Backtracking is really quite simple--we “explore” each node, as follows:
- 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”



Full example: Map coloring

- The **Four Color Theorem** states that any map on a plane can be colored with no more than four colors, so that no two countries with a common border are the same color
- For most maps, finding a legal coloring is easy
- For some maps, it can be fairly difficult to find a legal coloring
- We will develop a complete Java program to solve this problem



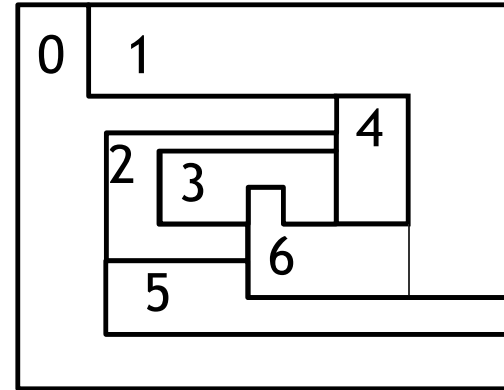
Data structures

- We need a data structure that is easy to work with, and supports:
 - Setting a color for each country
 - For each country, finding all adjacent countries
- We can do this with two arrays
 - An array of “colors”, where `countryColor[i]` is the color of the i^{th} country
 - A ragged array of adjacent countries, where `map[i][j]` is the j^{th} country adjacent to country i
 - Example: `map[5][3]==8` means the 3^{th} country adjacent to country 5 is country 8

Creating the map

```
int map[][];
```

```
void createMap() {  
    map = new int[7][];  
    map[0] = new int[] { 1, 4, 2, 5 };  
    map[1] = new int[] { 0, 4, 6, 5 };  
    map[2] = new int[] { 0, 4, 3, 6, 5 };  
    map[3] = new int[] { 2, 4, 6 };  
    map[4] = new int[] { 0, 1, 6, 3, 2 };  
    map[5] = new int[] { 2, 6, 1, 0 };  
    map[6] = new int[] { 2, 3, 4, 1, 5 };  
}
```





Setting the initial colors

```
static final int NONE = 0;  
static final int RED = 1;  
static final int YELLOW = 2;  
static final int GREEN = 3;  
static final int BLUE = 4;
```

```
int mapColors[] = { NONE, NONE, NONE, NONE,  
                   NONE, NONE, NONE };
```



The main program

(The name of the enclosing class is **ColoredMap**)

```
public static void main(String args[]) {  
    ColoredMap m = new ColoredMap();  
    m.createMap();  
    boolean result = m.explore(0, RED);  
    System.out.println(result);  
    m.printMap();  
}
```




The backtracking method

```
boolean explore(int country, int color) {  
    if (country >= map.length) return true;  
    if (okToColor(country, color)) {  
        mapColors[country] = color;  
        for (int i = RED; i <= BLUE; i++) {  
            if (explore(country + 1, i)) return true;  
        }  
    }  
    return false;  
}
```



Checking if a color can be used

```
boolean okToColor(int country, int color) {  
    for (int i = 0; i < map[country].length; i++) {  
        int ithAdjCountry = map[country][i];  
        if (mapColors[ithAdjCountry] == color) {  
            return false;  
        }  
    }  
    return true;  
}
```



Printing the results

```
void printMap() {  
    for (int i = 0; i < mapColors.length; i++) {  
        System.out.print("map[" + i + "] is ");  
        switch (mapColors[i]) {  
            case NONE:    System.out.println("none");    break;  
            case RED:     System.out.println("red");     break;  
            case YELLOW:  System.out.println("yellow");  break;  
            case GREEN:   System.out.println("green");   break;  
            case BLUE:    System.out.println("blue");    break;  
        }  
    }  
}
```



Recap

- We went through all the countries recursively, starting with country zero
- At each country we had to decide a color
 - It had to be different from all adjacent countries
 - If we could not find a legal color, we reported failure
 - If we could find a color, we used it and recurred with the next country
 - If we ran out of countries (colored them all), we reported success
- When we returned from the topmost call, we were done



The End
