Backtracking

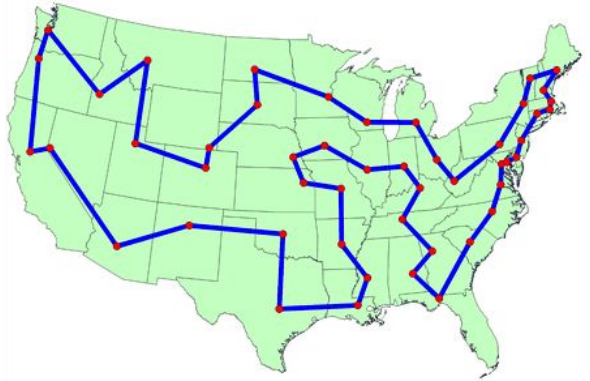
CSCI 104 Lab 9

Introduction: Graph Algorithms

- What are graphs useful for? Why do we represent data with graphs as compared to “unordered collections”?
  - They help describe **relationships** between elements within different systems

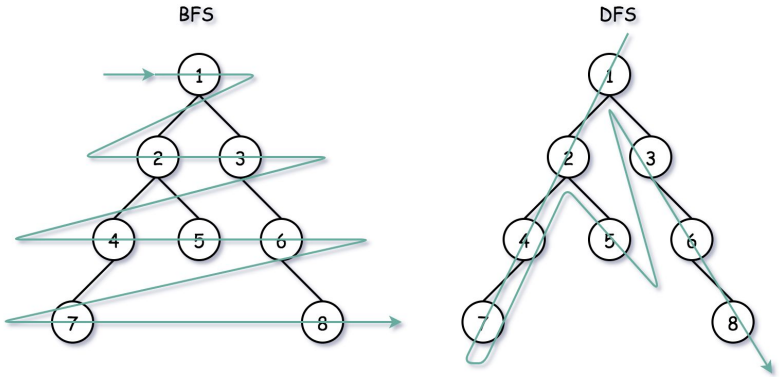
Graph Algorithms

- The information conveyed in graphs helps us solve many different types of problems, ex:
  - Arrangement: ex. Alphabetically sort a database
  - Networking: ex. Shortest path to take between two cities (roads = edges)
  - Optimization: ex. Shortest path to visit *every* city (traveling salesman)
  - Matching: ex. Match every patient to a doctor
  - Any many more!!!



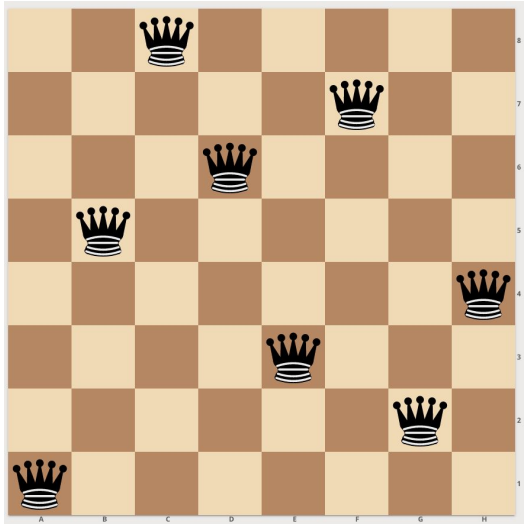
Introduction: Graph Algorithms

- The baseline graph algorithms to getting to solve all sorts of complicated problems are:
  - Breadth first search
    - Visiting neighbors in “concentric rings”, a.k.a. With a FIFO queue
  - Depth first search
    - Visiting neighbors by “digging deep”, a.k.a. With a LIFO stack



…Backtracking?

- Backtracking is a type of graph algorithm that is used to solve **constraint satisfaction** problems, i.e. the final state of the graph must meet a certain list of requirements
- Examples of this include:
  - Sudoku (what we are doing today)
  - N-Queens
  - Map coloring (3-color, 4-color, etc.)
  - And many more!

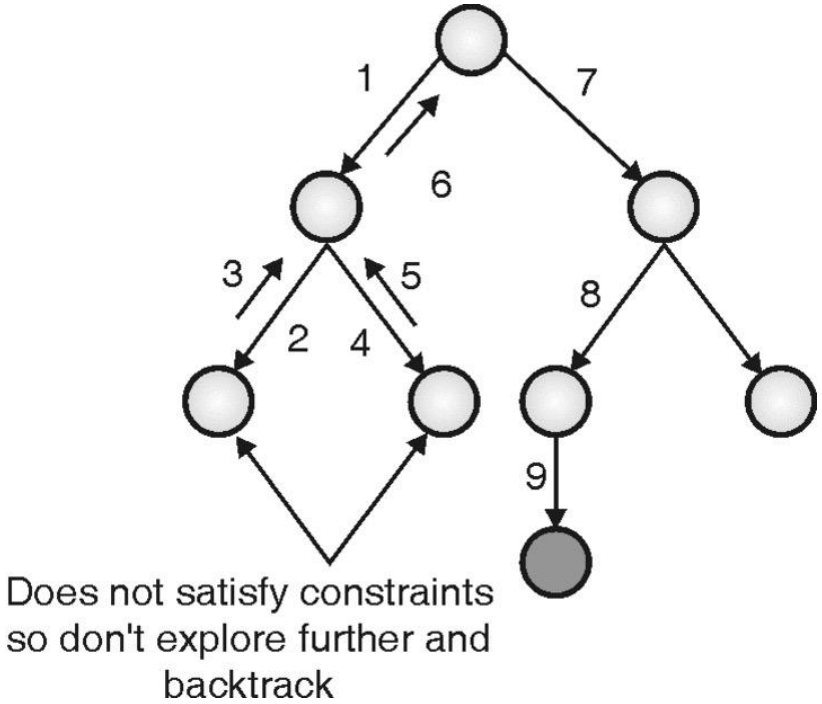


…Backtracking?

- The general backtracking algorithm is a modification of **depth first search**
- Apply a change to our current state
- if it's valid, then we continue down that path, adding new changes (depth first search!!!)
    - If we find a solution, great! Return and all done
    - If there are no more possible valid options and we haven't hit an end state yet, then we "undo" our last change, return false, and are brought back up to the previous state where we can try a different option

Example

- Numbers by arrows are the order the backtracking algo visits nodes in



Pseudocode

- This is a very general idea! (i.e. lab code is not this simple)

- Not **ALL** backtracking algos will return a bool
- Sometimes you check state/validity at multiple points in the function
- Etc.

```
void solve():
  recursive_helper(params)

bool is_valid():
  # returns whether state is valid

bool recursive_helper(params):
  if finished state and valid:
    save solution
    return true

  for each next possible state choice:
    apply choice to state
    if choice is valid:
      recursive call with current state
    remove choice (Backtrack)

  return false (no viable solution found)
```

Time Complexity?

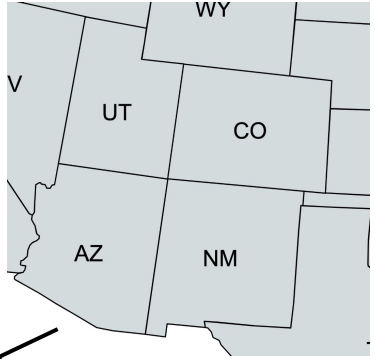
- Backtracking is not that efficient :(
- Different problems that backtracking solves take more or less time, so there is no set time complexity
  - N-Queens problem has an upper bound of O(n!) time, where n is the number of queens
  - Map coloring has an upper bound of O(n * m^n), where n is the number of nodes (countries) and m is the number of colors used

Time Complexity?

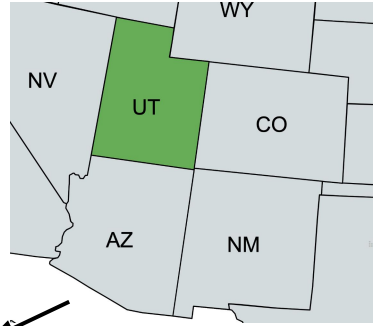
- More efficient than DFS though!
  - This is because as soon as backtracking determines a potential solution is **invalid**, it backtracks and tries other things; DFS would keep going
- Example on next slide

Time Complexity?

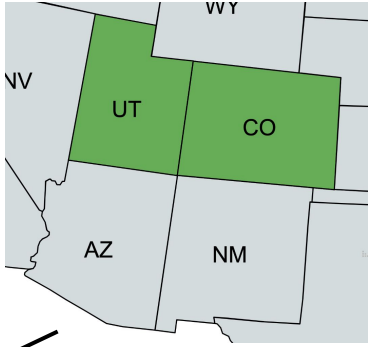
- DFS map coloring



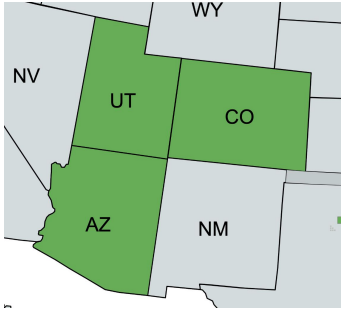
Initial state



adds color; map not done



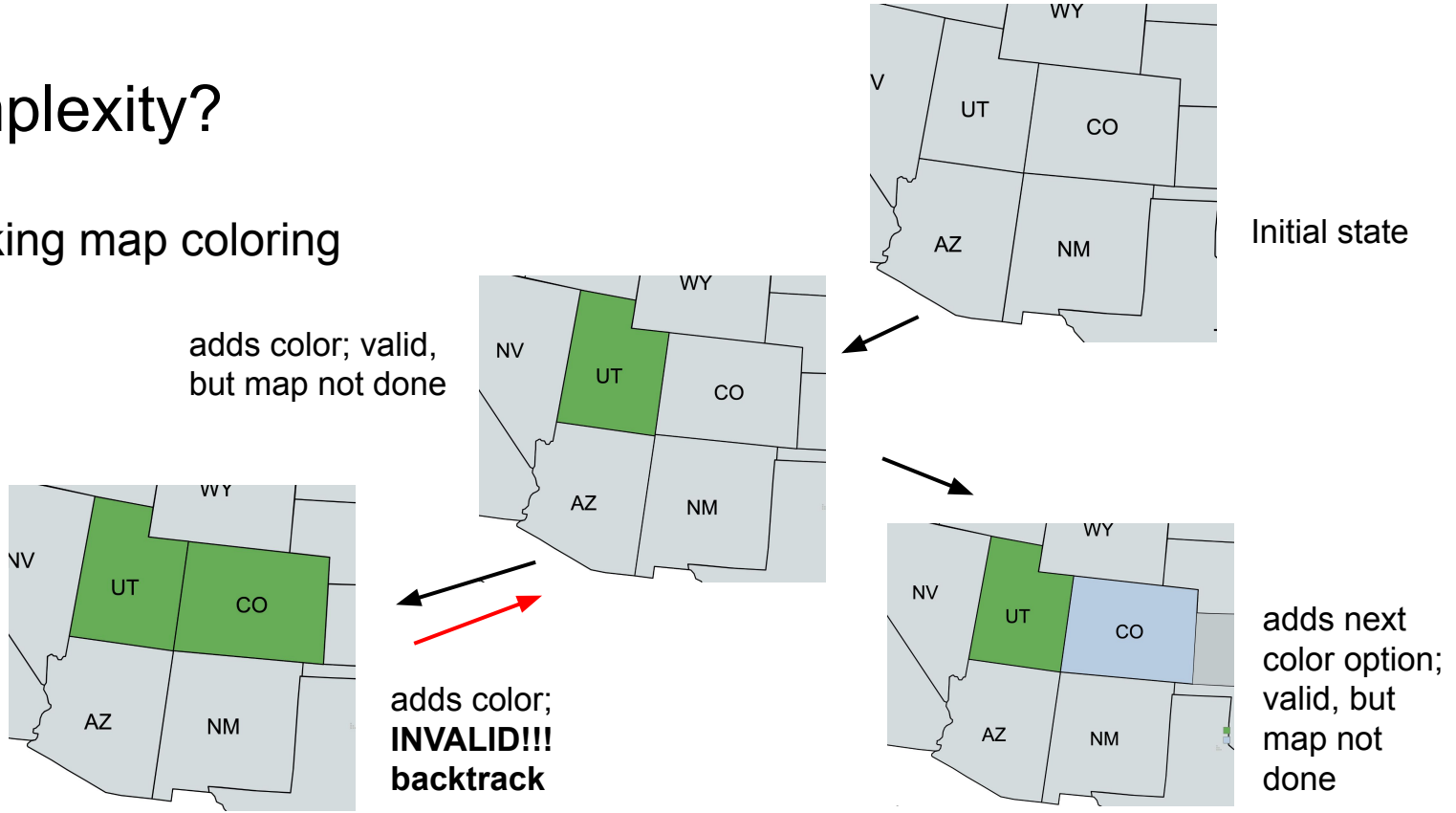
adds color; map not done



DFS keeps coloring green until all states colored; only when **done** it realizes it's invalid!

Time Complexity?

- Backtracking map coloring



Backtracking will **immediately stop** when it realizes something is invalid

The Lab

- Fill in the solveHelper function in Sudoku.cpp
- While there is no backtracking for current PA, PAs 5 and 6 will both have :)
- Show passing tests to TA/CP