BACKTRACKING

Need to have unique colors for each state that borders another, so no repeating colors touch eachother.

48 continental states, color it.

Start with which state is touching which state, adjacency lists

Backtracking:

1. progressively solve the problem

2. as long as partial solution is valid, keep going.

3. if we get stuck, we backtrack to a previous position, where we haven’t used all the choices yet.

4. keep going until: 1. Valid complete solution is found

2. exhaust all possibilities

\*\*code in lecture 12-2-13\*\*

How do we:

1. know where to start?

2. know where to go next?

3. choose what color to assign?

4. know things have gone wrong?

5. know where to back up to?

6. know what to try next?

7. know if we solved the problem?

8. know if we have run out of possibilities?

9. what our search space is?

10. know how many possible complete colorings there are? (does not imply valid)

11. visualize the search process?

12. implement the search?

Answers:

**9**. colorings of the 48 states, partial or complete

**10**. 48 states, 4 colors to choose from? 4^48

Tree, searching it:

\* Root node, uncolored initial map

(Tennessee) 1 2 3 4 <- choices for coloring first state, a coloring in which one state has been assigned

4 more branches off each node before it, exponential expansion, continues until depth of 48, make last assignment of last color

We have 4^48 nodes in our tree at depth 48

Need to test every branch to see if it corresponds to a solution

Observations:

1. root corresponds to initial condition, no color assigned

2. leaf node corresponds to a conmpelte coloring (a possible solution)

3. internal nodes corresponds to partial solution (maybe valid/ maybe not)

4. define an order by which we make progresss, move from one node to next.

make an example tree, 48 \* 4 total branches, 192

second level47 \* 4 branches emanating from each branch at depth 1, new branches’

massive duplication, identical branches with same colors

want to design tree so that are no repeated color combinations

don’t want tenessee red and new York blue in one scenario, and new York red and tenessee blue in another scenario.

**3**. use a for loop so you can know you assigned a color to each state.

**4**. Check for adjacency information to match color info, don’t want repeats, skip next if failed

**Pruning** speeds up sorting, helps to identify failed options

**5**. back up to nodes where colors are chosen for a state and then you choose the next color

**6**. next color is tested

**7**. still valid, if we check validiry after assigning all colors, if no conflicts, then a complete coloring I found

Implementing the algorithm:

Pass in partially colored map, adjacency info

Start with root node, with 48 uncolored states

Basecase: when map is full

Def find.coloring( [1]current\_coloring, adj\_map):

If current\_coloring[3] is completely filled

Return as solution

Else:

[4][5]Generate all successors of current coloring

For each coloring C in our list of successors:

[7]Check if C is valicd

If valid:

Recursively search -> call find\_colorings(C, adj\_map)

If this returns solution, return that solution

[6]Return None

[1] current\_coloring is current (partial) solution. Some states colored, some not

[1b] assumes that current\_coloring is valid when we enter the function

Defined so that we only enter function with current\_coloring that is valid

[3] checking for solution only requires complete coloring check

[4] USES DEFINED ORDERING

[5] we must make copies of the current\_coloring. Use deepcopy – completely new set of allocated memory, complete copy of information, recirsove called function

[6] if we fail to find a solution, return Nno

[7] where we prune

If we prune those staes with most neighbors first, the nyou get rid of problems first and the rest goes smoothly

Def solve(config):

#config is information given to you

If ([2]isGoal([1]config):

Return config

Else:

For child in [3]successors(config):

If i[4]svalid(child):

Solution = solve(config) #recursive call with updated config file

If solution != None

Return solution

Return None

[1] configuration, tells you all information you need about what part of the solution has been fileld in and what is left. A list of the states, indicates color or adjacent items

[2] isGoal checks if complete solution (usually short function)

[3] use deepCopy to generate new configs, make copy of config, add to it, add new information

[4] looks to prune, if found, return false, child was invalid anyway

For Sudoku, a 9x9 list of lists, config is storing 2d grid, 9x9 list of values, some filled in some not

Validity check for all spaces filled, then it’s a solution, goal check for filled cells too

Successors checks for cells with 1-9, order is left to right, top to bottom

Successors progress solution

Empty cell is filled, check for validity in new spots

Make tree so there is only one possible tree, not 2 at same thing

8 queens puzzle:

Try to find a way to put 8 queens on board so none of them can attack eachother

Successors checks for queens, adds one in a spot, either row or column at a time. Generates children, to ensure paths that don’t merge together, stop duplicating, 8 children, then 8 new configurations for each one of the next queens, then test for valid place, then prevent duplicaitons, no branches pointing to other branches and rpeating