# N-Queens

## N-Queens

- The object is to place queens on a chess board in such a way that no queen can capture another one in a single move.
  - Recall that a queen can move horizontally, vertically, or diagonally for an infinite distance.
    - This implies that no two queens can be on the same row, column, or diagonal.
  - We usually want to know how many different placements there are.

# 4-Queens

- Let’s take a look at the simple problem of placing four queens on a $4 \times 4$ board.
- The brute force solution is to place the first queen, then the second, third, and forth.
  - After all are placed, we determine if they are placed legally
- There are 16 spots for the first queen, 15 for the second, and so on.
  - Leading to 16 * 15 * 14 * 13 = 43,680 different combinations
- Obviously, this isn’t a good way to solve the problem.

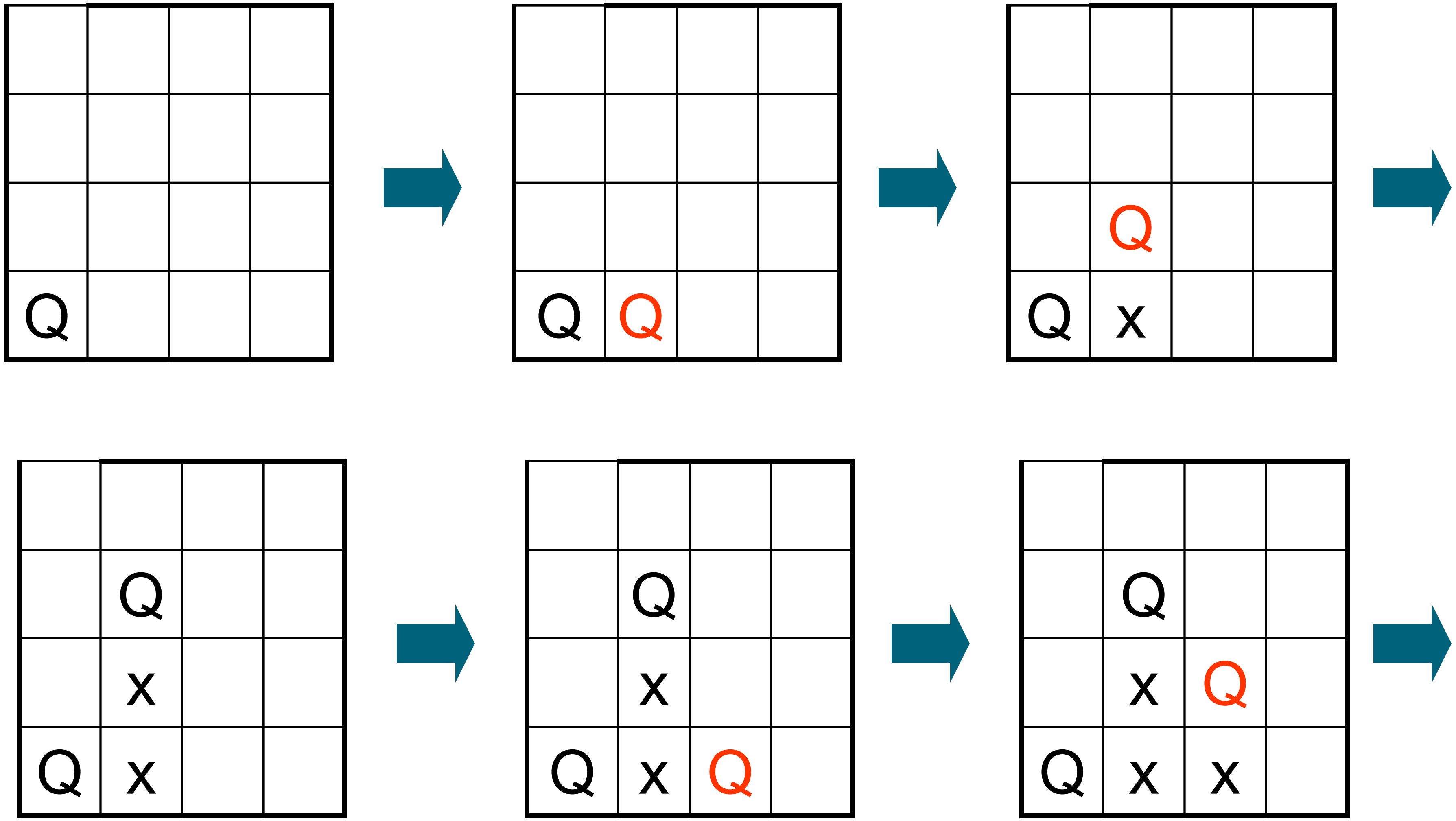
# 4-Queens

- First let's use the fact that no two queens can be in the same column to help us.
  - That means we get to place a queen in each column.
- So, we can place the first queen into the first column, the second into the second, and so on.
- This cuts down on the amount of work.
  - Now there are four spots for the first queen, four spots for the second, and so on.
    - 4 * 4 * 4 * 4 = 256 different combinations

## 4-Queens

- However, we can still do better because, as we place each queen, we can look at the previous queens we have placed to make sure that our new queen is not in the same row or diagonal as a previously place queen.
- Then we could use a Greedy-like strategy to select the next valid position for each column.

# 4-Queens



## 4-Queens

| | | | |
|---|---|---|---|
| | | | |
| | Q | Q | |
| | x | x | |
| Q | x | x | |

$\Rightarrow$

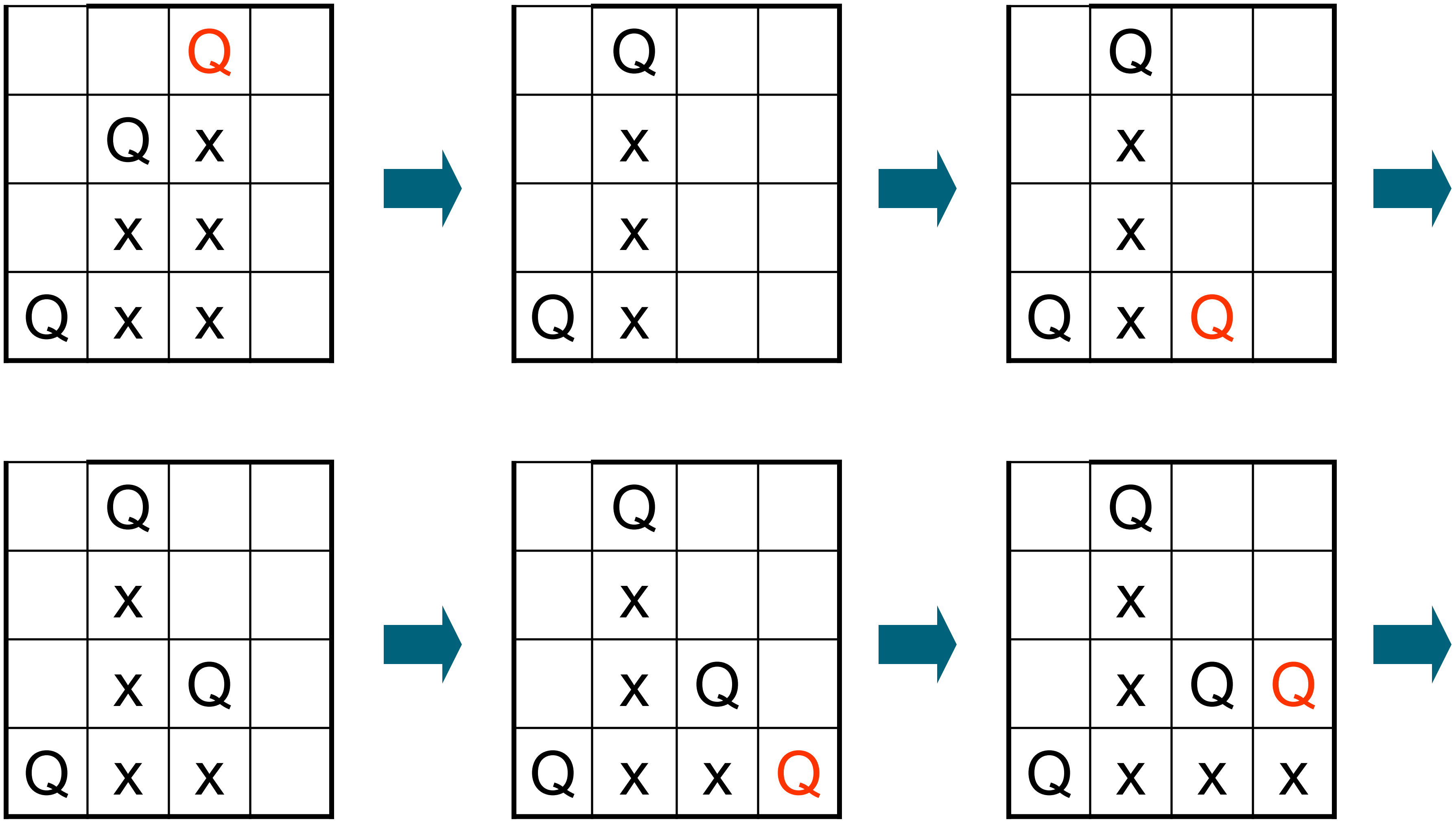
| | | | |
|---|---|---|---|
| | | Q | |
| | Q | x | |
| | x | x | |
| Q | x | x | |

$\Rightarrow$ **We are stuck!**

# 4-Queens

- So now what do we do?
- Well, this is very much like solving a maze.
  - As you walk through the maze, you have to make a series of choices.
  - If one of your choices leads to a dead end, you need to back up to the last choice you made and take a different route.
    - That is, you need to change one of your earlier selections.
  - Eventually, you will find your way out of the maze.

# 4-Queens



# 4-Queens

| | | | |
|---|---|---|---|
| | Q | | |
| | x | | Q |
| | x | Q | x |
| Q | x | x | x |

$\rightarrow$

| | | | |
|---|---|---|---|
| | Q | | Q |
| | x | | x |
| | x | Q | x |
| Q | x | x | x |

$\rightarrow$

| | | | |
|---|---|---|---|
| | Q | | |
| | x | Q | |
| | x | x | |
| Q | x | x | |

$\rightarrow$

| | | | |
|---|---|---|---|
| | Q | Q | |
| | x | x | |
| | x | x | |
| Q | x | x | |

$\rightarrow$

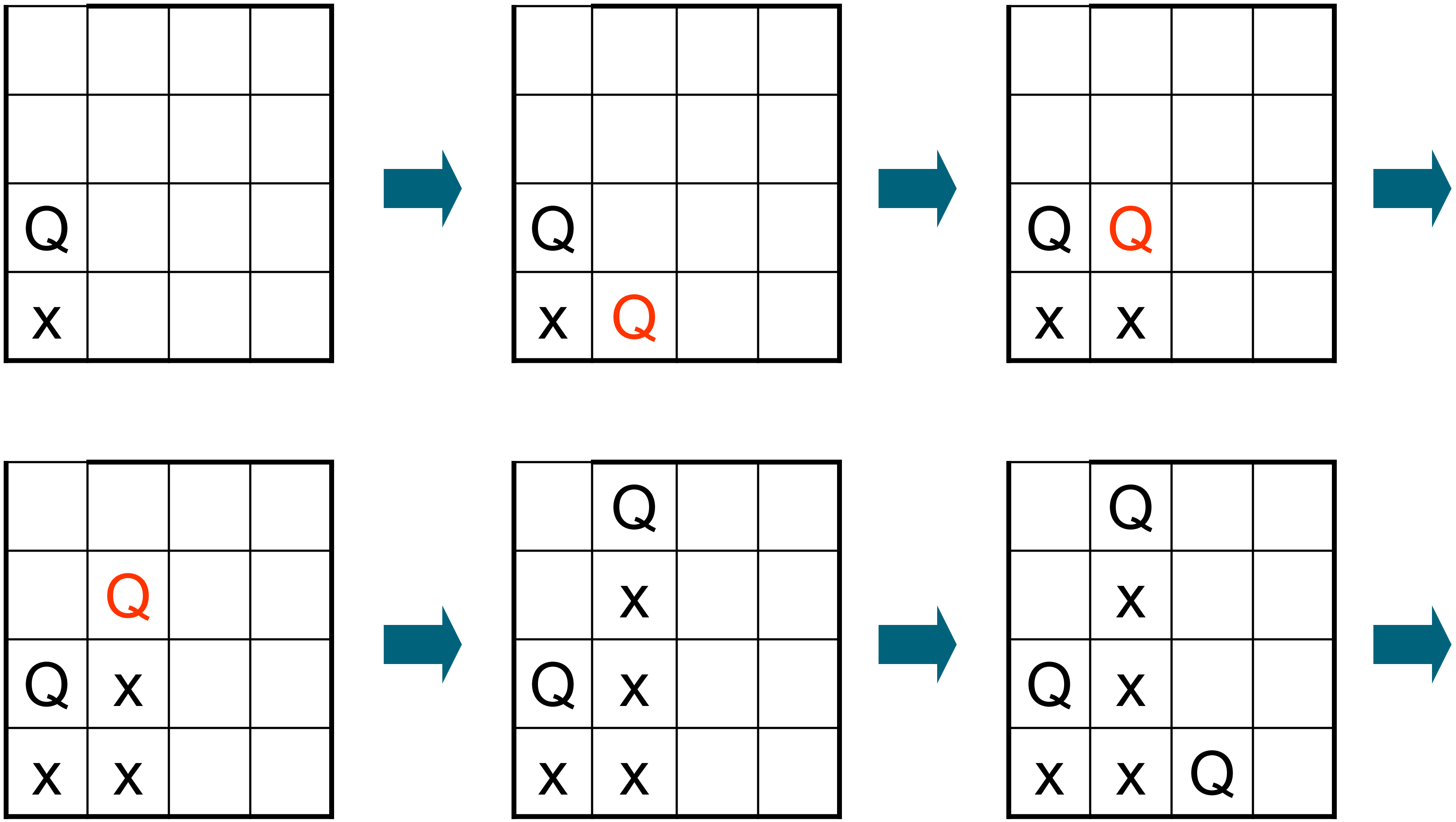
| | | | |
|---|---|---|---|
| | Q | | |
| | x | | |
| | x | | |
| Q | x | | |

$\rightarrow$

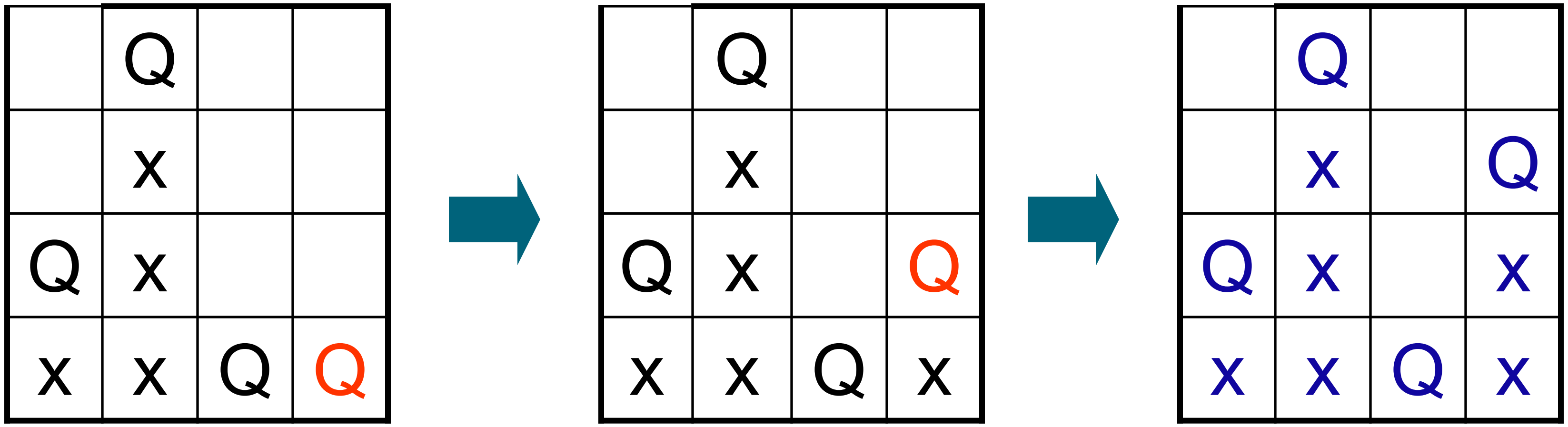
| | | | |
|---|---|---|---|
| | | | |
| | | | |
| Q | | | |
| x | | | |

$\rightarrow$

# 4-Queens



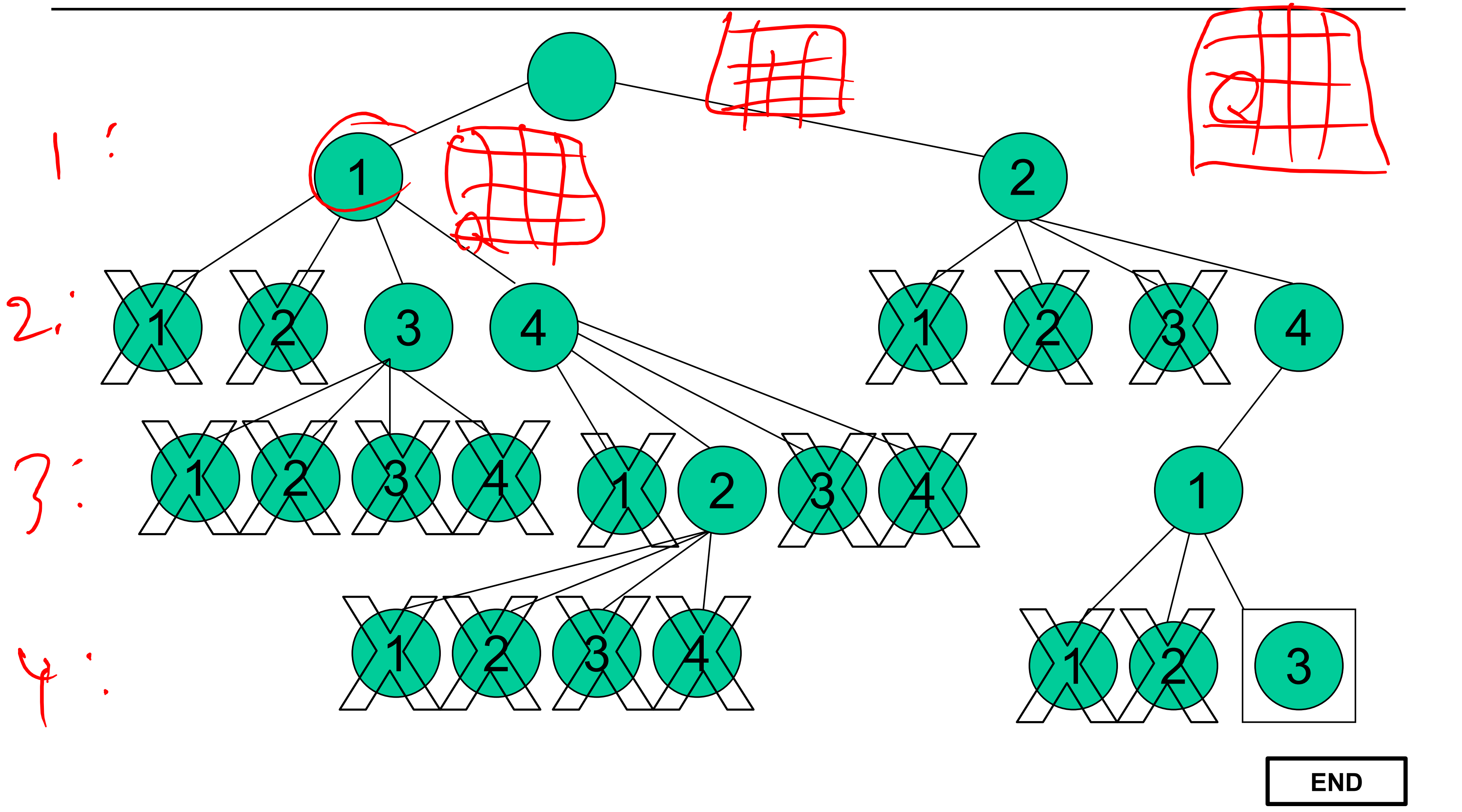
# 4-Queens



# 4-Queens

- This type of problem is often viewed as a state-space tree.
  - A tree of all the states in which the problem can be
- We start with an empty board state at the root and try to work our way down to a leaf node.
  - Leaf nodes are completed boards

# 4-Queens



## Graph Coloring

## Graph Coloring

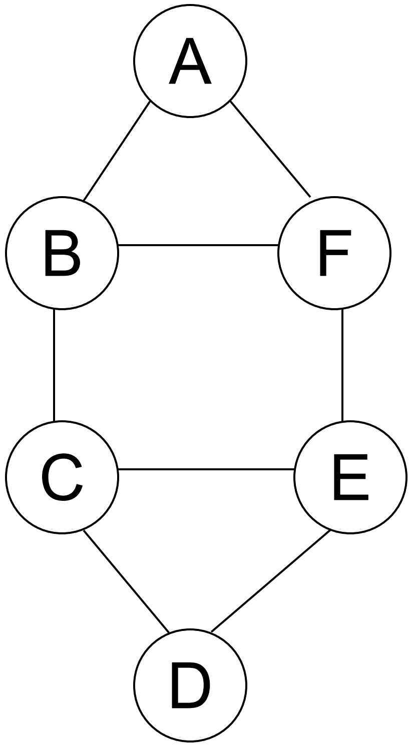
- Graph coloring is the problem of coloring each vertex in a graph such that no two adjacent vertices are the same color.
- Some direct examples include:
  - Map coloring
  - Register assignment

## Graph Coloring

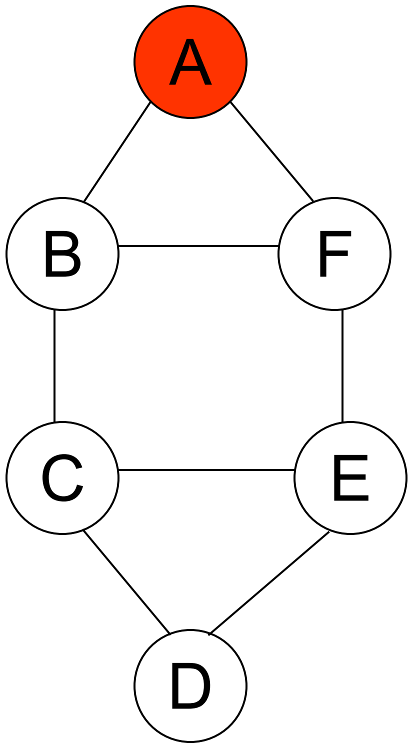
- The same issues apply as in N-queens.
  - We don’t want to simply pick all subsets.
    - There are way too many.
  - We want to prune the state-space tree as soon as we find something that won’t work.
    - This implies that we need a sequence of vertices to color.
    - As we color the next vertex, we need to make sure that it doesn’t conflict with any of its previously colored neighbors.
  - We may need to backtrack.

## Graph Coloring

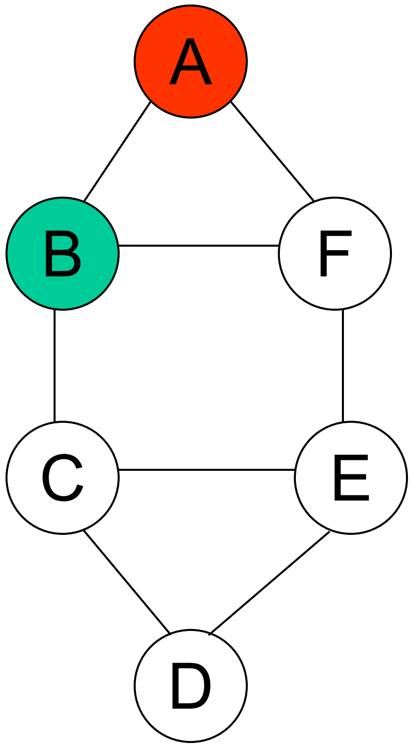
- As an example:
  - The vertices are enumerated in order A–F.
  - The colors are given in order R, G, B.



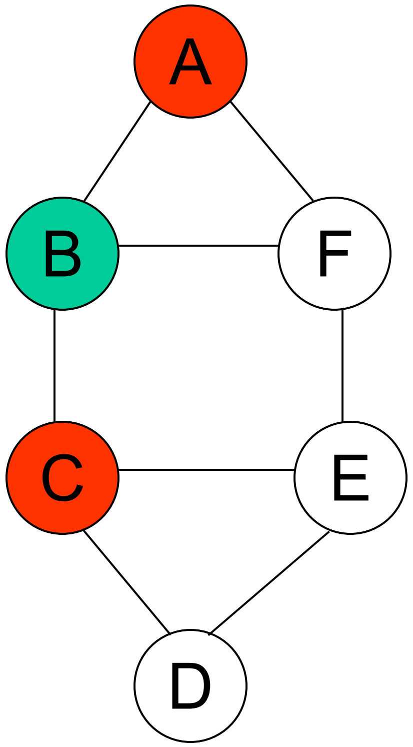
## Graph Coloring



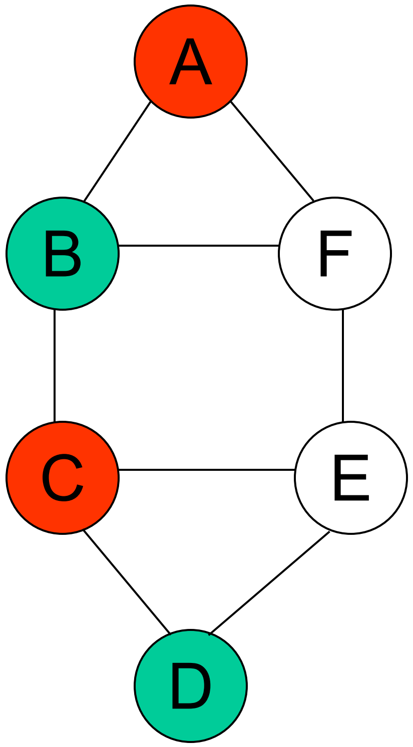
# Graph Coloring



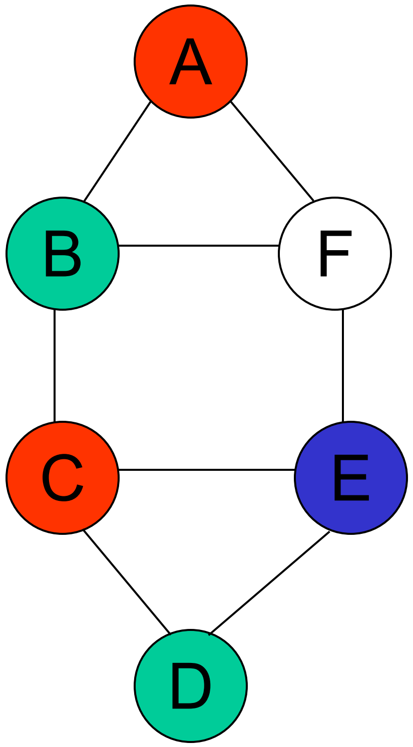
# Graph Coloring



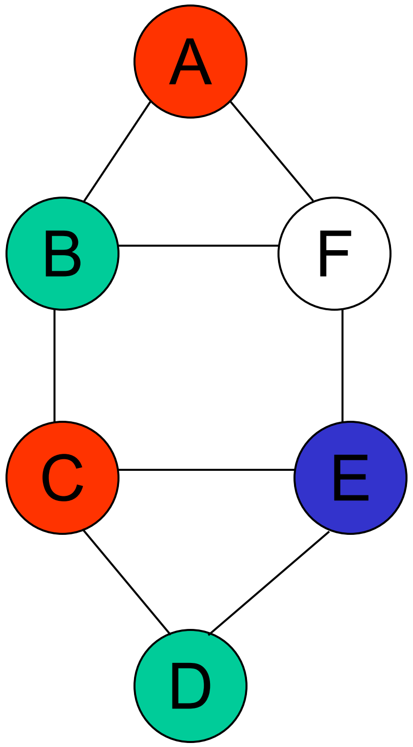
# Graph Coloring



# Graph Coloring

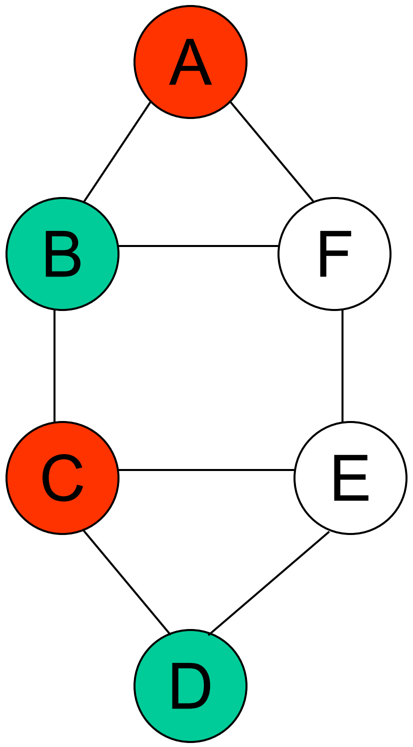


# Graph Coloring

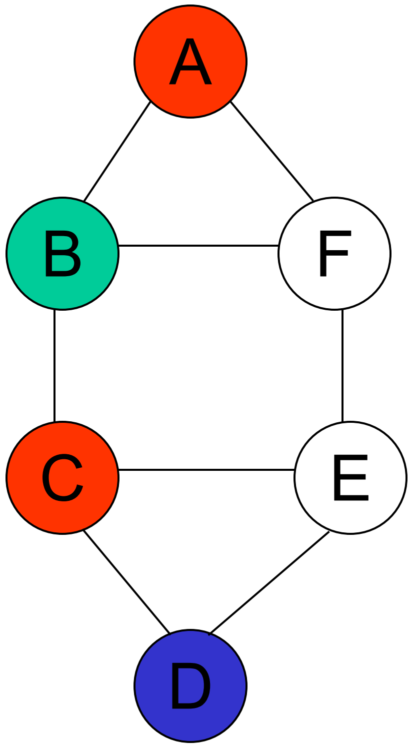


**Stuck!**

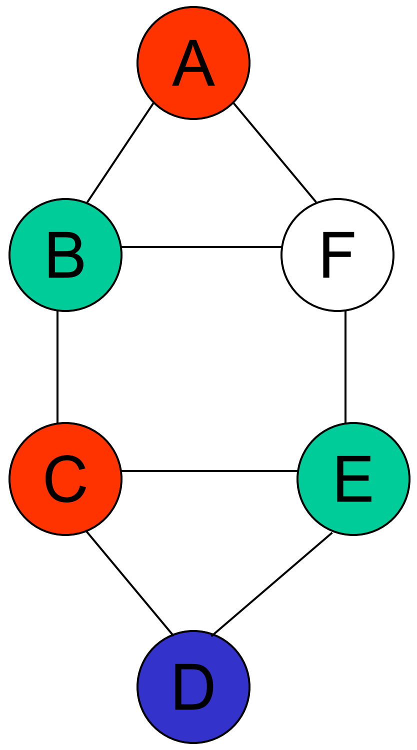
## Graph Coloring



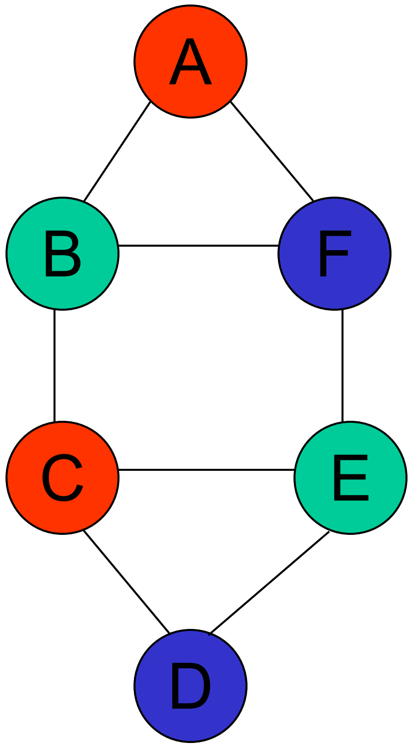
# Graph Coloring



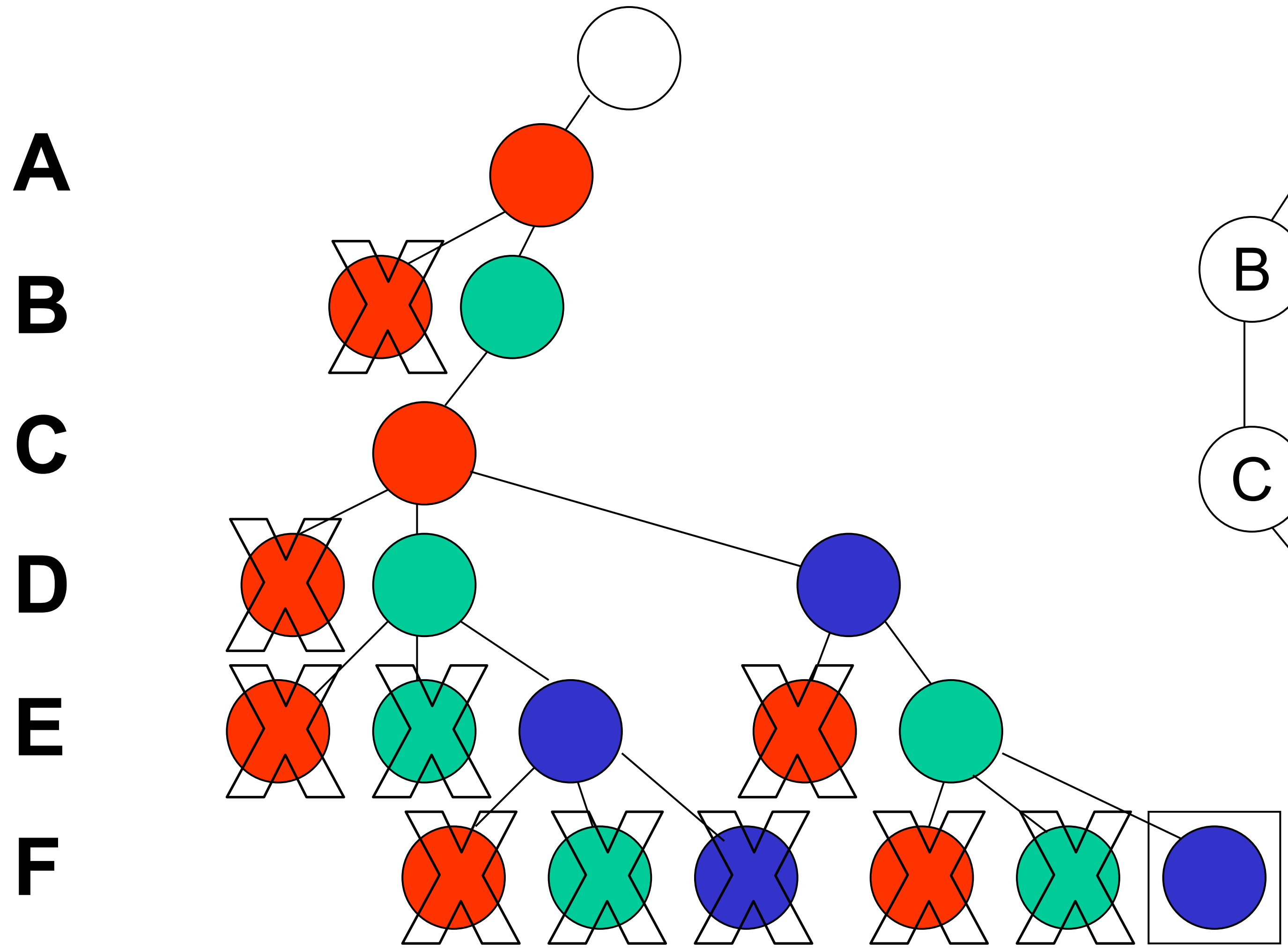
## Graph Coloring

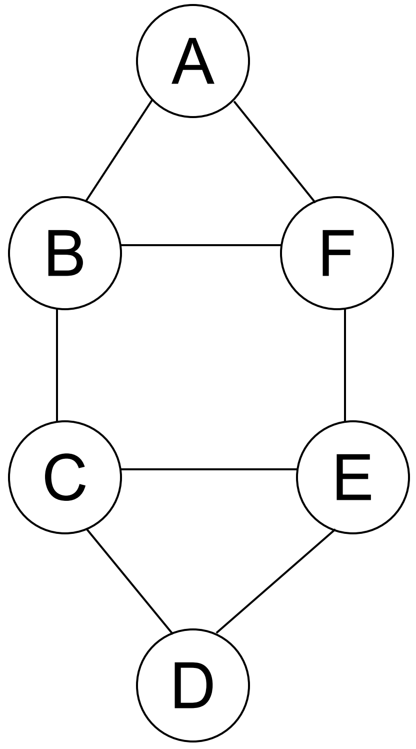


## Graph Coloring



# Graph Coloring





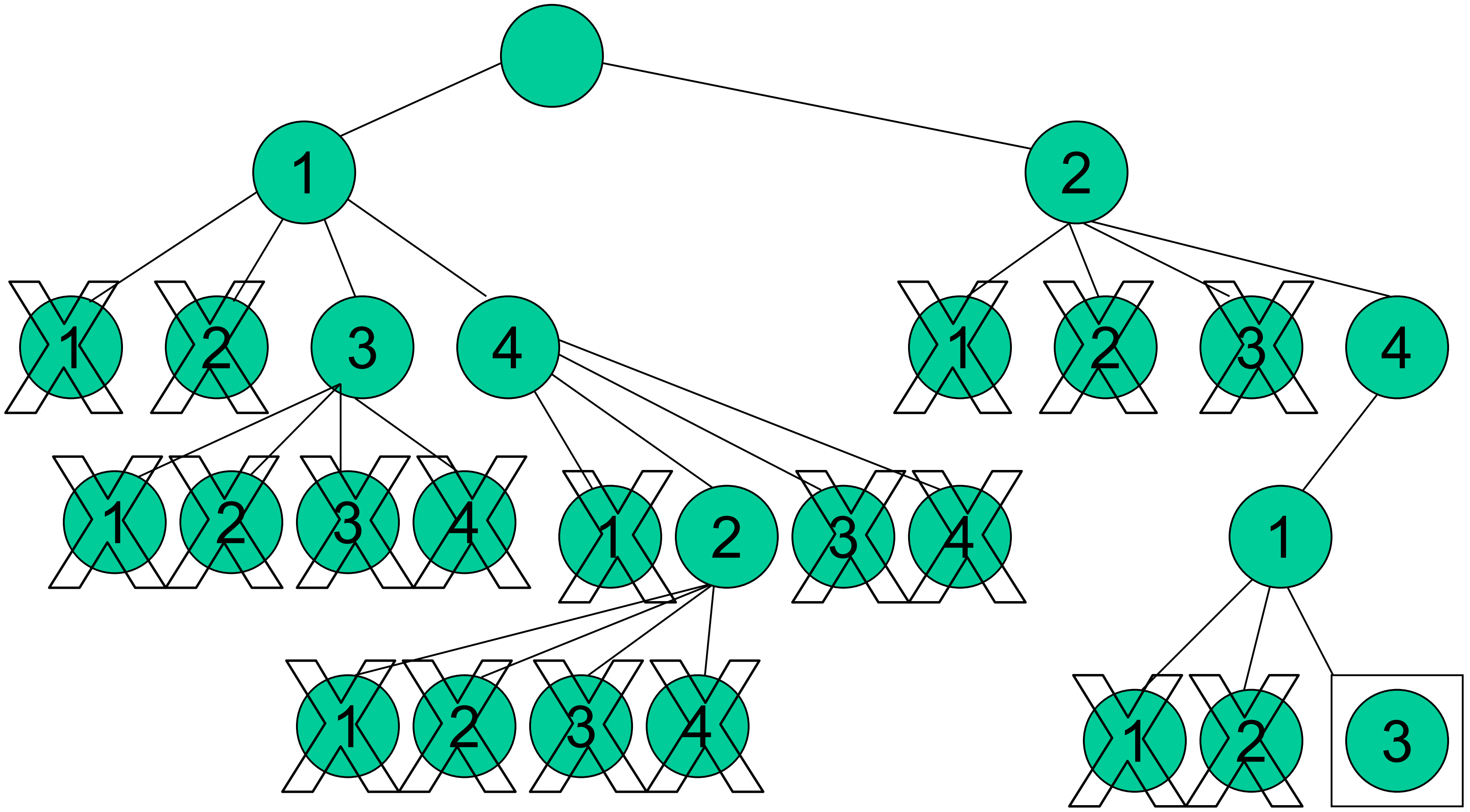
END

## Backtracking as Tree Search

# Depth-First Search

- One can view backtracking as a depth-first search through the state space.
  - We start our search at the root
  - We end our search when we find a valid leaf node
- A depth-first search expands the deepest node next.
  - A child children will be expanded before the child's siblings are expanded.
- This type of search gets you to the leaf nodes as fast as you can.
- How do you implement a depth-first search?

# 4-Queens



## Depth- vs. Breadth-First Search

- A breadth-first search expands all the children at a particular level before advancing to the next level.
  - How can a breadth-first search be implemented?
  - Which is better: a depth- or breadth-first search?

END

## Weighted Graph Coloring

# Optimization Problems

- The previous examples simply found all the solutions (N-queens) or any single solution (graph coloring).
- What if we have a cost associated with each solution and we want to find the optimal solution?
  - As an example, take weighed graph coloring
    - R → \$2, G → \$3, B → \$5, Y → \$2
    - A fully connected graph of three vertices

## Weighted Graph Coloring

- Obviously, this approach will take a long time since we need to look at every possible combination.
- Is there any way we can prune even more nodes from the tree?
  - Besides the nonfeasible ones that we already prune
- What about an upper or lower bound on a node?
  - An estimate of the potential cost of expanding the node

## Bounding

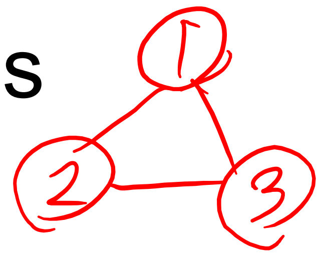
- A bound on a node is a guarantee that any solution obtained from expanding the node will be:
    - Greater than some number (lower bound)
    - Or less than some number (upper bound)
- If we are looking for a minimal optimal, as we are in weighted graph coloring, then we need a lower bound.
    - For example, if the best solution we have found so far has a cost of 12 and the lower bound on a node is 15, then there is no point in expanding the node
        - The node cannot lead to anything better than a 15.

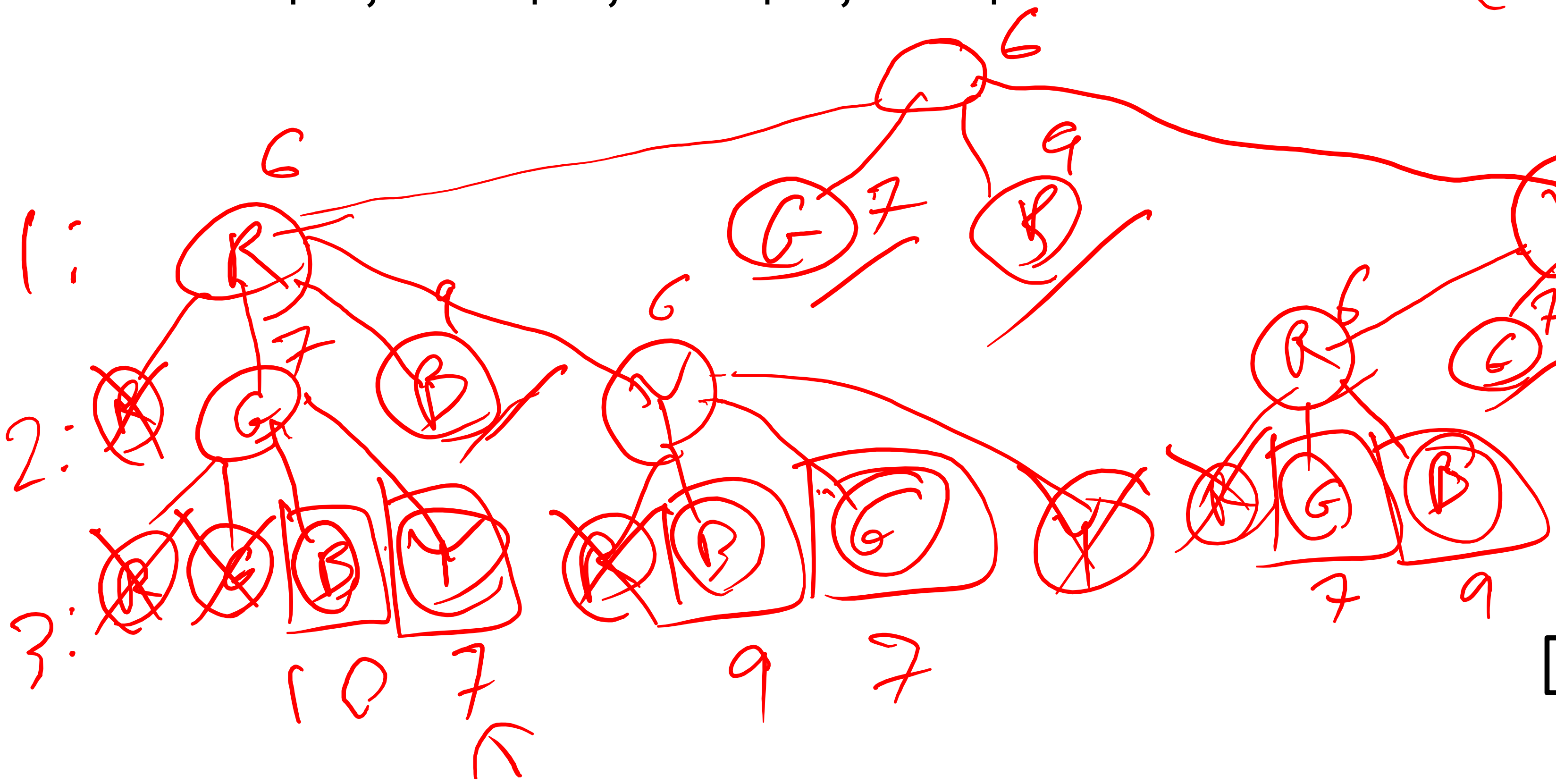
## Bounding

- We can compute a lower bound for weighted graph color in the following way:
    - The actual cost of getting to the node
    - Plus a bound on the future cost
        - Minimal weight color * number of nodes still to color
            - That is, the future cost cannot be any better than this.

# State-Space Tree Example (DFS)

- Fully connected graph of three nodes
- R: \$2, G: \$3, B: \$5, Y: \$2





END

## Best-First Search

# Bounding

- We prune (via bounding) when: (currentBestSolutionCost <= nodeBound)
- This tells us that we get more pruning if:
  - The currentBestSolution is low
  - And the nodeBound is high
- So, we want to find a low solution quickly and we want the highest possible lower bound.
  - One has to factor in the extra computation cost of computing higher lower bounds vs. the expected pruning savings

## Best-First Search

- Depth-first search found *a* solution quickly.
- Breadth-first search found all solutions at about the same time (but slowly).
- Best-first search finds *a good* solution *fairly* quickly.
  - Not as quickly as depth-first
  - But the solution is often better, which will increase the pruning via bounding

## Best-First Search

- Best-first search expands the node with the best bounds next.
- How would you implement a best-first search?
  - Depth-first is a stack.
  - Breadth-first is a queue.
  - Best-first is a priority queue.

## State-Space Tree (Best-First)

- Fully connected graph of three nodes
- R: $2, G: $3, B: $5, Y: $2

END

## Traveling Salesperson

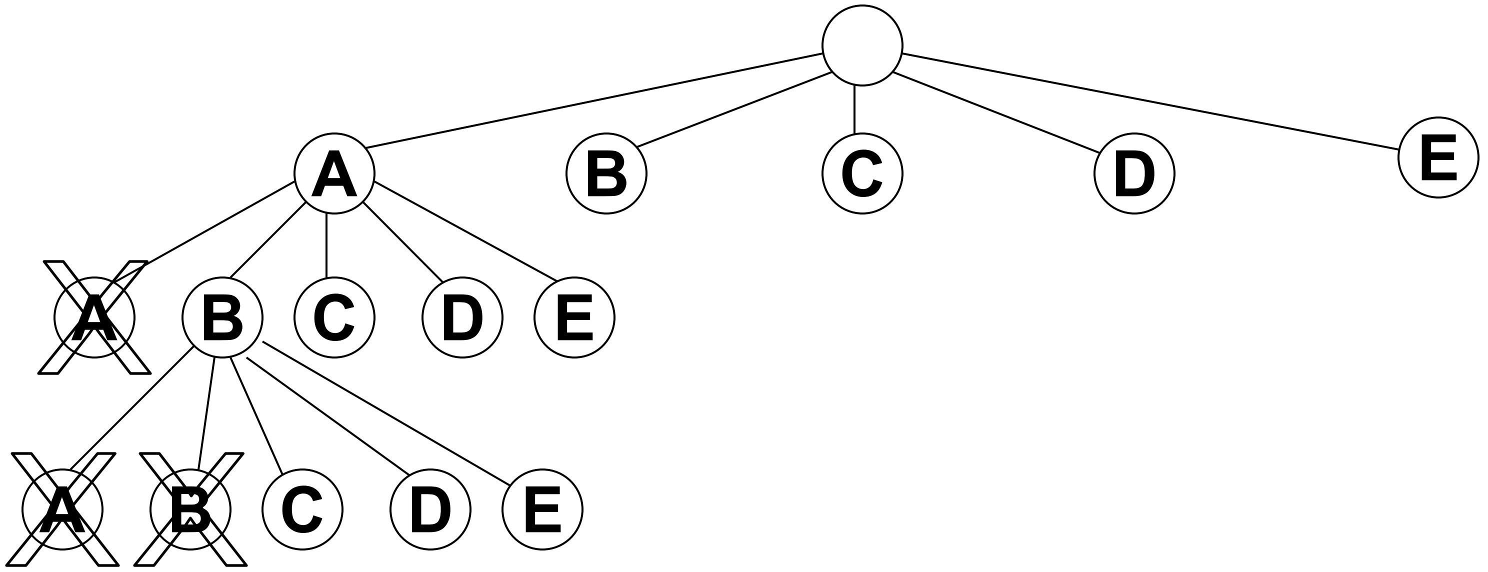
# Traveling Salesperson Problem

- This is a classic CS problem.
- Given a graph (cities) and weights on the edges (distances), find a minimum weight tour of the cities.
  - Start in a particular city.
  - Visit all other cities (*exactly* once each).
  - Return to the starting city.
- It cannot be done by brute force as this is worst-case exponential or worse running time.
  - So, we will look to backtracking with pruning to make it run in a reasonable amount of time in most cases.

# Traveling Salesperson Problem

- We will build our state space by:
  - Having our children be all the potential cities we can go to next
  - Having the depth of the tree be equal to the number of cities in the graph
    - We need to visit each city exactly once.
- So, given a fully connected set of five nodes, we have the following state space:
  - Only partially completed

# Traveling Salesperson Problem



# Traveling Salesperson Problem

- Now we need to add bounding to this problem
  - It is a minimization problem, so we need to find a lower bound
- We can use:
  - The current cost of getting to the node
  - Plus an underestimate of the future cost of going through the rest of the cities
    - The obvious choice is to find the minimum weight edge in the graph and multiply that edge weight by the number of remaining nodes through which to travel

# Traveling Salesperson Problem

- As an example, assume that we have the given adjacency matrix.
- If we started at node A and have just traveled to node B, then we need to compute the bound for node B.
    - Cost 14 to get from A to B
    - Minimum weight in matrix is 2 $\times$ 4 more legs to go to get back to node A = 8
    - For a grand total of 14 + 8 = 22

| | | | | |
|---|---|---|---|---|
| 0 | 14 | 4 | 10 | 20 |
| 14 | 0 | 7 | 8 | 7 |
| 4 | 5 | 0 | 7 | 16 |
| 11 | 7 | 9 | 0 | 2 |
| 18 | 7 | 17 | 4 | 0 |

# Traveling Salesperson Problem

- Recall that, if we can make the lower bound higher, then we will get more pruning.
- Note that, in order to complete the tour, we need to leave nodes B, C, D, and E.
  - The minimum edge we can take leaving B is min(14, 7, 8, 7) = 7.
  - Similarly, C = 4, D = 2, E = 4.
- This implies that, at best, the future underestimate can be 7 + 4 + 2 + 4 = 17.
- 17 + current cost of 14 = 31.
  - This is much higher than 8 + 14 = 22.

| | | | | |
|---|---|---|---|---|
| 0 | 14 | 4 | 10 | 20 |
| 14 | 0 | 7 | 8 | 7 |
| 4 | 5 | 0 | 7 | 16 |
| 11 | 7 | 9 | 0 | 2 |
| 18 | 7 | 17 | 4 | 0 |

END

## Best-First Search

# Bounding

- We prune (via bounding) when: (currentBestSolutionCost <= nodeBound)
- This tells us that we get more pruning if:
  - The currentBestSolution is low
  - And the nodeBound is high
- So, we want to find a low solution quickly and we want the highest possible lower bound.
  - One has to factor in the extra computation cost of computing higher lower bounds vs. the expected pruning savings

## Best-First Search

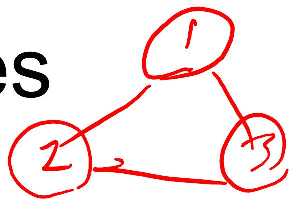
- Depth-first search found *a* solution quickly.
- Breadth-first search found all solutions at about the same time (but slowly).
- Best-first search finds *a good* solution *fairly* quickly.
    - Not as quickly as depth-first
    - But the solution is often better, which will increase the pruning via bounding

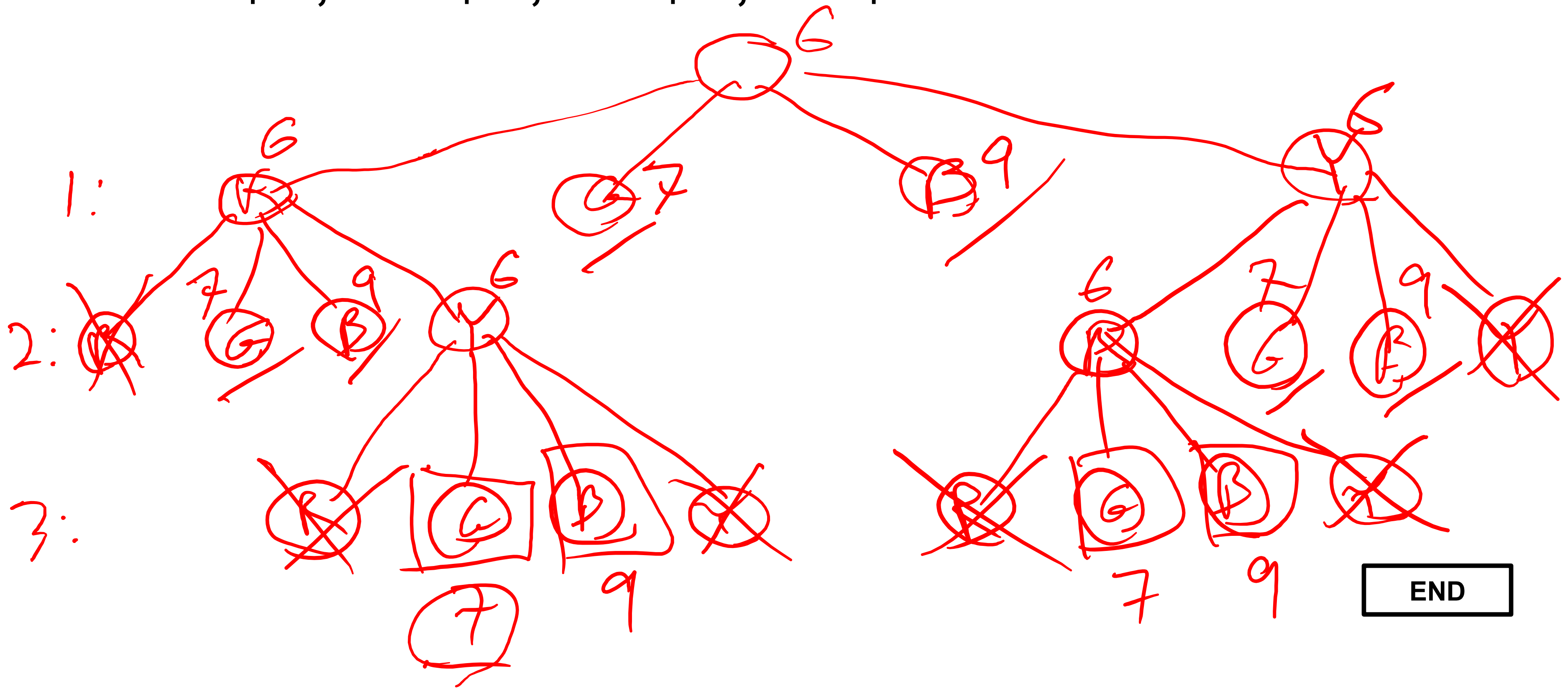
## Best-First Search

- Best-first search expands the node with the best bounds next.
- How would you implement a best-first search?
    - Depth-first is a stack.
    - Breadth-first is a queue.
    - Best-first is a priority queue.

# State-Space Tree (Best-First)

- Fully connected graph of three nodes
- R: $2, G: $3, B: $5, Y: $2





END

## Traveling Salesperson

# Traveling Salesperson Problem

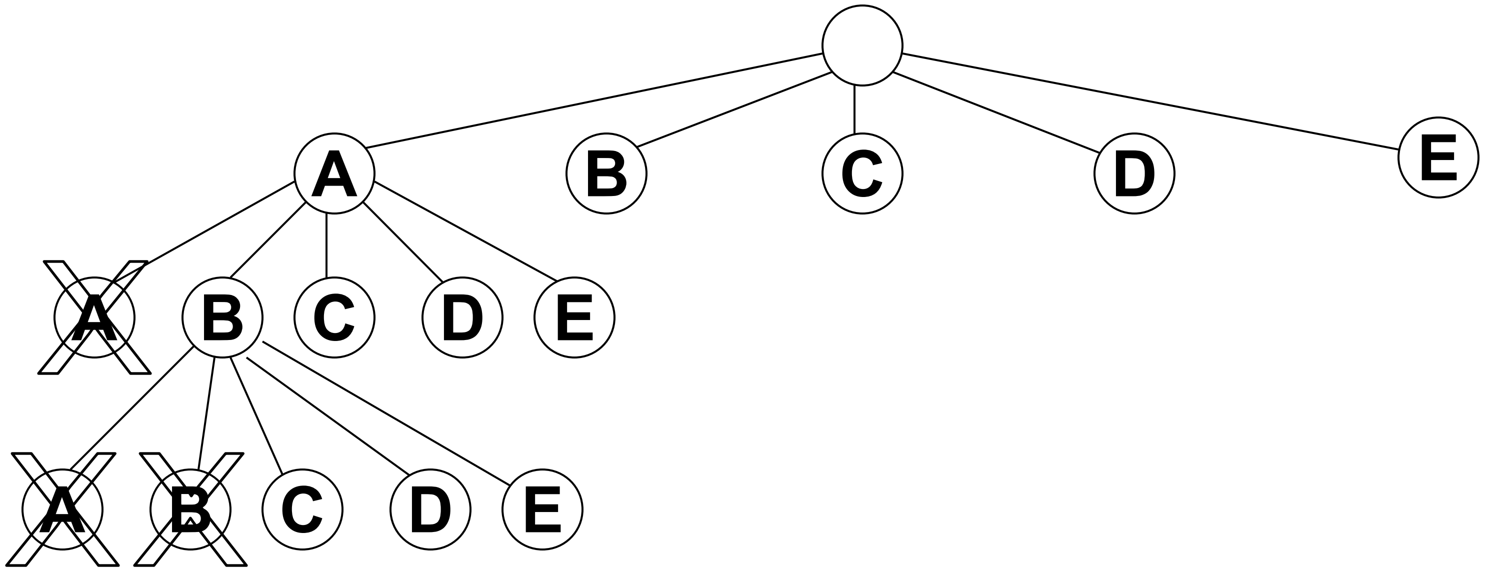
- This is a classic CS problem.
- Given a graph (cities) and weights on the edges (distances), find a minimum weight tour of the cities.
  - Start in a particular city.
  - Visit all other cities (*exactly* once each).
  - Return to the starting city.
- It cannot be done by brute force as this is worst-case exponential or worse running time.
  - So, we will look to backtracking with pruning to make it run in a reasonable amount of time in most cases.

$$\underline{n-1,\ n-2,\ n-3,\ \ldots\ 1}$$
$$n!$$

## Traveling Salesperson Problem

- We will build our state space by:
  - Having our children be all the potential cities we can go to next
  - Having the depth of the tree be equal to the number of cities in the graph
    - We need to visit each city exactly once.
- So, given a fully connected set of five nodes, we have the following state space:
  - Only partially completed

# Traveling Salesperson Problem



## Traveling Salesperson Problem

- Now we need to add bounding to this problem
  - It is a minimization problem, so we need to find a lower bound
- We can use:
  - The current cost of getting to the node
  - Plus an underestimate of the future cost of going through the rest of the cities
    - The obvious choice is to find the minimum weight edge in the graph and multiply that edge weight by the number of remaining nodes through which to travel

# Traveling Salesperson Problem

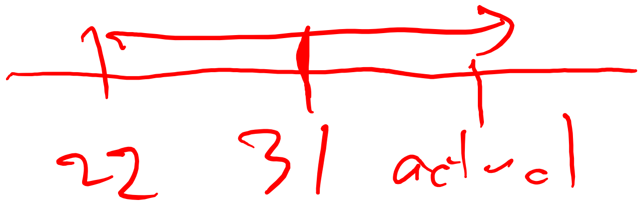
- As an example, assume that we have the given adjacency matrix.
- If we started at node A and have just traveled to node B, then we need to compute the bound for node B.
  - Cost 14 to get from A to B
  - Minimum weight in matrix is 2 × 4 more legs to go to get back to node A = 8
  - For a grand total of 14 + 8 = 22

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 14 & 4 & 10 & 20 \\ B & 14 & 0 & 7 & 8 & 7 \\ C & 4 & 5 & 0 & 7 & 16 \\ D & 11 & 7 & 9 & 0 & 2 \\ E & 18 & 7 & 17 & 4 & 0 \end{array}$$

# Traveling Salesperson Problem

- Recall that, if we can make the lower bound higher, then we will get more pruning.
- Note that, in order to complete the tour, we need to leave nodes B, C, D, and E.
  - The minimum edge we can take leaving B is min(14, 7, 8, 7) = 7.
  - Similarly, C = 4, D = 2, E = 4.
- This implies that, at best, the future underestimate can be 7 + 4 + 2 + 4 = 17.
- 17 + current cost of 14 = 31.
  - This is much higher than 8 + 14 = 22.

| | | | | |
|---|---|---|---|---|
| 0 | 14 | 4 | 10 | 20 |
| 14 | 0 | 7 | 8 | 7 |
| 4 | 5 | 0 | 7 | 16 |
| 11 | 7 | 9 | 0 | 2 |
| 18 | 7 | 17 | 4 | 0 |



END