Backtracking

General Concepts

- Algorithm strategy
  - *Approach to solving a problem*
  - *May combine several approaches*
- Algorithm structure
  - *Iterative ⇒ execute action in loop*
  - *Recursive ⇒ reapply action to subproblem(s)*
- Problem type
  - *Satisfying ⇒ find any satisfactory solution*
  - *Optimization ⇒ find best solutions (vs. cost metric)*

A short list of categories

- Many Algorithm types are to be considered:
  - *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"

Backtracking Algorithm

- Based on depth-first recursive search
- Approach
  1. *Tests whether solution has been found*
  2. *If found solution, return it*
  3. *Else for each choice that can be made*
     a) Make that choice
     b) Recur
     c) If recursion returns a solution, return it
  4. *If no choices remain, return failure*
- Some times called "search tree"

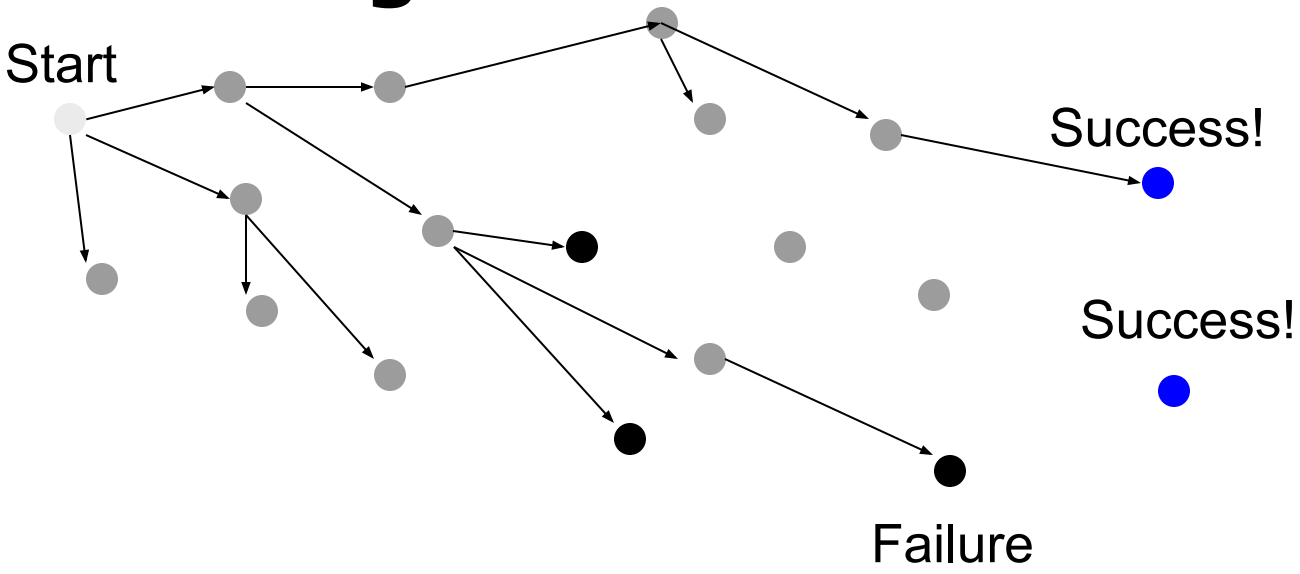
Backtracking Algorithm – Example

- Find path through maze
  - *Start at beginning of maze*
  - *If at exit, return true*
  - *Else for each step from current location*
    - Recursively find path
    - Return with first successful step
    - Return false if all steps fail

Backtracking Algorithm – Example

- Color a map with no more than four colors
  - *If all countries have been colored return success*
  - *Else for each color c of four colors and country n*
    - If country n is not adjacent to a country that has been colored c
      - Color country n with color c
      - Recursively color country n+1
      - If successful, return success
  - *Return failure*

Backtracking



Problem space consists of states (nodes) and actions (paths that lead to new states). When in a node can can only see paths to connected nodes

If a node only leads to failure go back to its "parent" node. Try other alternatives. If these all lead to failure then more backtracking may be necessary.

Recursive Backtracking

Pseudo code for recursive backtracking algorithms

If at a solution, return success
for(every possible choice from current state / node)
  Make that choice and take one step along path
  Use recursion to solve the problem for the new node / state
  If the recursive call succeeds, report the success to the next high level
  Back out of the current choice to restore the state at the beginning of the loop.
Report failure

Backtracking

- Construct the state space tree:
  - *Root represents an initial state*
  - *Nodes reflect specific choices made for a solution's components.*
    - Promising and nonpromising nodes
    - leaves

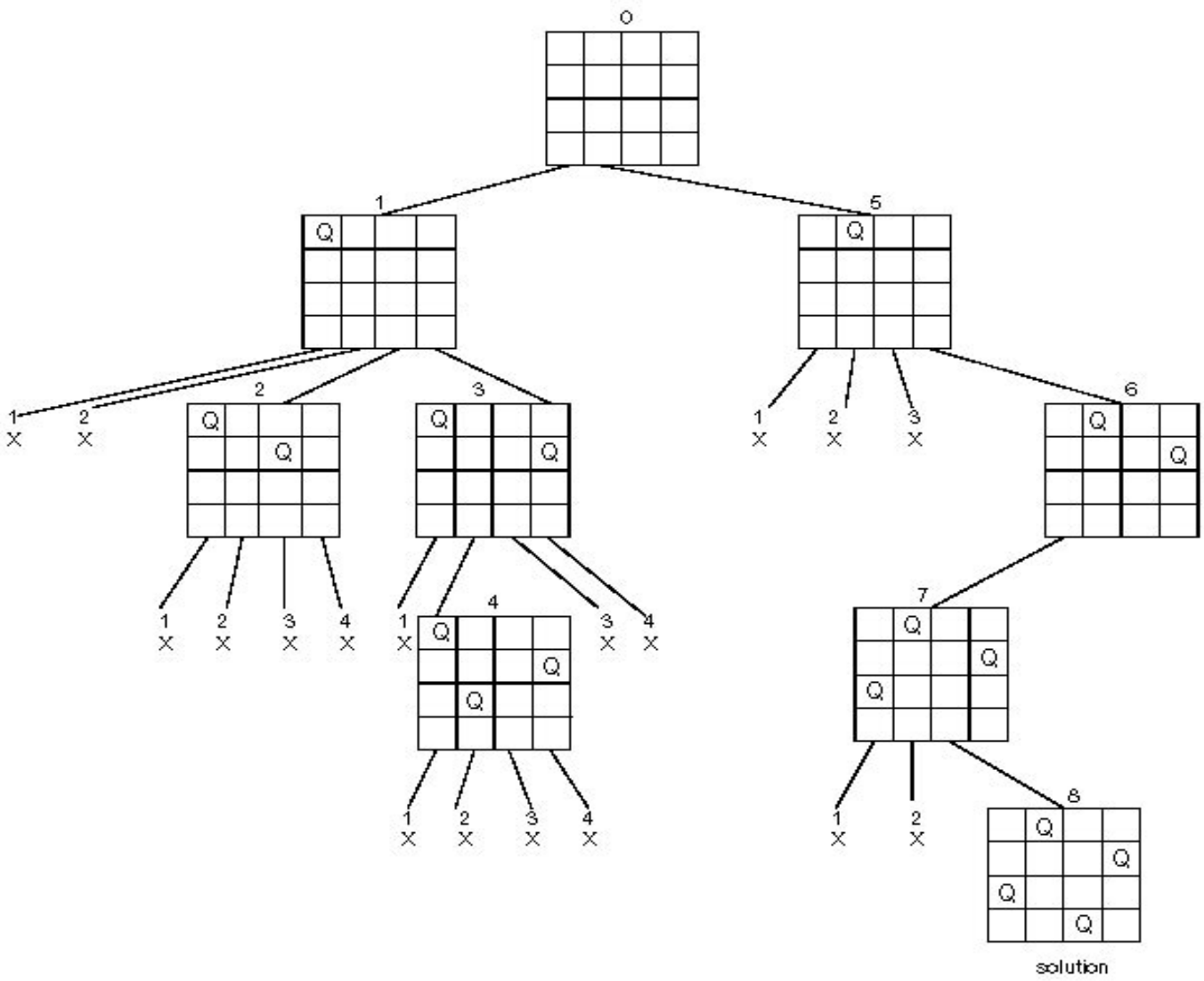
- Explore the state space tree using depth-first search

- "Prune" non-promising nodes
  - *dfs stops exploring subtree rooted at nodes leading to no solutions and...*
  - *"backtracks" to its parent node*

Example: The *n*-Queen problem

- Place *n* queens on an *n* by *n* chess board so that no two of them are on the same row, column, or diagonal

State Space Tree of the Four-queens Problem



The backtracking algorithm

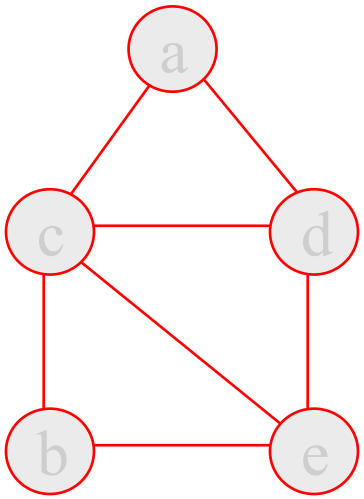
- Backtracking is really quite simple--we "explore" each node, as follows:
- To "explore" node N:
  - *1. If N is a goal node, return "success"*
  - *2. If N is a leaf node, return "failure"*
  - *3. For each child C of N,*
    - 3.1. Explore C
      - 3.1.1. If C was successful, return "success"
  - *4. Return "failure"*

Exercises

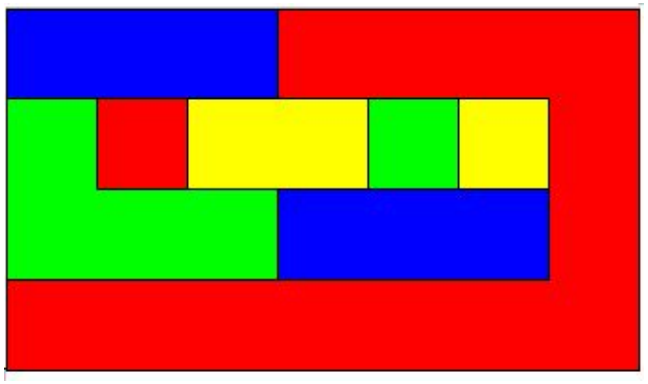
- Continue the backtracking search for a solution to the four-queens problem to find the second solution to the problem.
- A trick to use: the board is symmetric, obtain another solution by reflections.
- Get a solution to the 5-queens problem found by the back-tracking algorithm?
- Can you (quickly) find at least 3 other solutions?

The *m*-Coloring Problem and Hamiltonian Problem

- 2-color
- 3-color
- Hamiltonian Circuit
  (use alphabet order to break the ties)



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

Comments

- *Backtracking provides the hope to solve some problem instances of nontrivial sizes by pruning non-promising branches of the state-space tree.*
- *The success of backtracking varies from problem to problem and from instance to instance.*
- *Backtracking possibly generates all possible candidates in an exponentially growing state-space tree.*

Other Backtracking Problems *

- 8 Queens
- Knight's Tour
- Knapsack problem / Exhaustive Search
  - *Filling a knapsack. Given a choice of items with various weights and a limited carrying capacity find the optimal load out. 50 lb. knapsack. items are 1 40 lb, 1 32 lb. 2 22 lbs, 1 15 lb, 1 5 lb. A greedy algorithm would choose the 40 lb item first. Then the 5 lb. Load out = 45lb. Exhaustive search 22 + 22 + 5 = 49.*

Branch and Bound

Branch and Bound (B& B)

- An enhancement of backtracking
  - ***Similarity***
    - A state space tree is used to solve a problem.
  - ***Difference***
    - The branch-and-bound algorithm does not limit us to any particular way of traversing the tree.
    - Used only for optimization problems (since the backtracking algorithm requires the using of DFS traversal and is used for non-optimization problems.

Branch and Bound

- The idea:

  Set up a **bounding function**, which is used to compute a **bound** (for the value of the objective function) **at a node** on a state-space tree and determine **if it is promising**

  - ***Promising*** *(if the bound is better than the value of the best solution so far): expand beyond the node.*
  - ***Nonpromising*** *(if the bound is no better than the value of the best solution so far): not expand beyond the node (pruning the state-space tree).*

Traveling Salesman Problem

Construct the state-space tree:

- A node = a vertex: a vertex in the graph.
- A node that is not a leaf represents all the tours that start with the path stored at that node; each leaf represents a tour (or non-promising node).
- Branch-and-bound: we need to determine a **lower** bound for each node
  - For example, to determine a lower bound for node [1, 2] means to determine a lower bound on the length of any tour that starts with edge 1—2.
- Expand each promising node, and stop when all the promising nodes have been expanded. During this procedure, prune all the nonpromising nodes.
  - Promising node: the node's lower bound is less than current minimum tour length.
  - Non-promising node: the node's lower bound is NO less than current minimum tour length.

Traveling Salesman Problem—Bounding Function 1

- Because a tour must leave every vertex exactly once, **a lower bound** on the length of a tour is **b (lower bound) minimum cost of leaving every vertex**.
  - *The lower bound on the cost of leaving vertex v_1 is given by the minimum of all the nonzero entries in row 1 of the adjacency matrix.*
  - *…*
  - *The lower bound on the cost of leaving vertex v_n is given by the minimum of all the nonzero entries in row n of the adjacency matrix.*
- **Note**: This is not to say that there is a tour with this length. Rather, it says that there can be no shorter tour.
- Assume that the tour starts with v1.

Traveling Salesman Problem—Bounding Function 2

- Because every vertex must be entered and exited exactly once, a lower bound on the length of a tour is **the sum of the minimum cost of entering and leaving every vertex.**
  - *For a given edge (u, v), think of half of its weight as the exiting cost of u, and half of its weight as the entering cost of v.*
  - *The total length of a tour = the total cost of visiting(entering and exiting) every vertex exactly once.*
  - *The lower bound of the length of a tour = the lower bound of the total cost of visiting (entering and exiting) every vertex exactly once.*
    - Calculation:
      - for each vertex, pick top two shortest adjacent edges (their sum divided by 2 is the lower bound of the total cost of entering and exiting the vertex);
      - add up these summations for all the vertices.
- Assume that the tour starts with vertex a and that b is visited before c.

Traveling salesman example 2