Problem Solving and Search

Problem Solving

- Rational agents need to perform sequences of actions in order to achieve goals.
- Intelligent behavior can be generated by having a look-up table or reactive policy that tells the agent what to do in every circumstance, but:
  - Such a table or policy is difficult to build
  - All contingencies must be anticipated
- A more general approach is for the agent to have knowledge of the world and how its actions affect it and be able to simulate execution of actions in an internal model of the world in order to determine a sequence of actions that will accomplish its goals.
- This is the general task of **problem solving** and is typically performed by **searching** through an internally modelled space of world states.

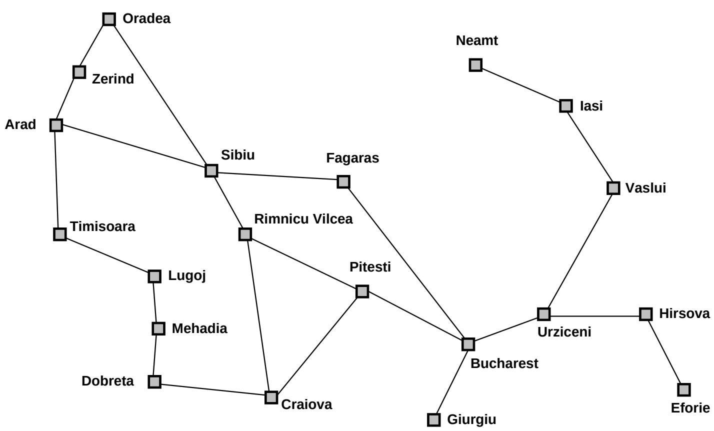
Problem Solving Task

- Given:
  - An **initial state** of the world
  - A set of possible possible actions or **operators** that can be performed.
  - A **goal test** that can be applied to a single state of the world to determine if it is a goal state.
- Find:
  - A **solution** stated as a **path** of states and operators that shows how to transform the initial state into one that satisfies the goal test.
- The initial state and set of operators implicitly define a **state space** of states of the world and operator transitions between them. May be infinite.

Measuring Performance

- **Path cost**: a function that assigns a cost to a path, typically by summing the cost of the individual operators in the path. May want to find minimum cost solution.
- **Search cost**: The computational time and space (memory) required to find the solution.
- Generally there is a trade-off between path cost and search cost and one must **satisfice** and find the best solution in the time that is available.

Sample Route Finding Problem



Initial state: Arad
Goal state: Bucharest

Path cost: Number of intermediate cities, distance traveled,
expected travel time

Sample “Toy” Problems

- 8-puzzle (sliding tile puzzle)

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

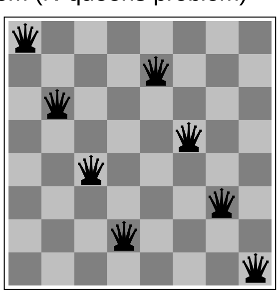
- Peg Puzzle (Hi-Q)

- Cryptarithmetic

```
  FORTY        29786       F=2, O=9, R=7, T=8,...
+   TEN      +   850
+   TEN      +   850
 ----------  ---------
  SIXTY        31486
```


“Toy” Problems (cont)

- 8-queens problem (N-queens problem)



- Missionaries and cannibals

Identity of individuals irrelevant,best to represent state as

(M,C,B) M = number of missionaries on left bank
C = number of cannibals on left bank
B = number of boats on left bank (0 or 1)

Operators to move: 1M, 1C, 2M, 2C, 1M1C

Goal state: (0,0,0)

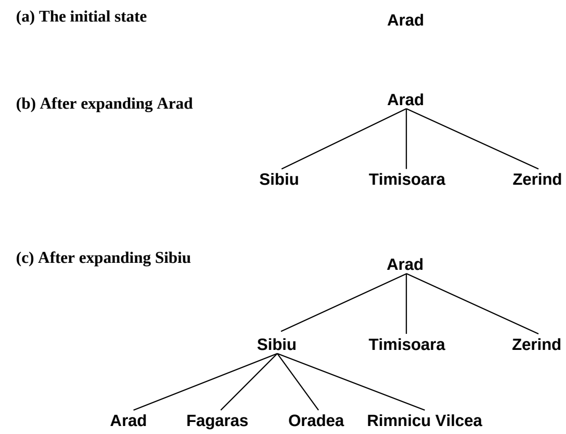
More Realistic Problems

- Route finding
- Travelling salesman problem
- VLSI layout
- Robot navigation
- Web searching

Searching Concepts

- A state can be **expanded** by generating all states that can be reached by applying a legal operator to the state.
- State space can also be defined by a **successor function** that returns all states produced by applying a single legal operator.
- A **search tree** is generated by generating search nodes by successively expanding states starting from the initial state as the root.
- A **search node** in the tree can contain
  - Corresponding state
  - Parent node
  - Operator applied to reach this node
  - Length of path from root to node (depth)
  - Path cost of path from initial state to node

Expanding Nodes and Search



function GENERAL-SEARCH(*problem*, *strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end

Search Algorithm

- Easiest way to implement various search strategies is to maintain a queue of unexpanded search nodes.
- Different strategies result from different methods for inserting new nodes in the queue.

function GENERAL-SEARCH(*problem*, QUEUING-FN) **returns** a solution, or failure

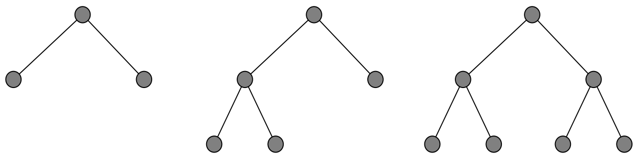
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))
  loop do
    if *nodes* is empty **then return** failure
    node ← REMOVE-FRONT(*nodes*)
    if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
    nodes ← QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))
  end

Search Strategies

- Properties of search strategies
  - Completeness
  - Time Complexity
  - Space Complexity
  - Optimality
- **Uniformed search** strategies (**blind, exhaustive, brute-force**) do not guide the search with any additional information about the problem.
- **Informed search** strategies (**heuristic, intelligent**) use information about the problem (estimated distance from a state to the goal) to guide the search.

Breadth-First Search

- Expands search nodes level by level, all nodes at level *d* are expanded before expanding nodes at level *d*+1



- Implemented by adding new nodes to the end of the queue (FIFO queue):

  GENERAL-SEARCH(problem, ENQUEUE-AT-END)

- Since eventually visits every node to a given depth, guaranteed to be complete.

- Also optimal provided path cost is a nondecreasing function of the depth of the node (e.g. all operators of equal cost) since nodes explored in depth order.

Breadth-First Complexity

- Assume there are an average of *b* successors to each node, called the **branching factor**.

- Therefore, to find a solution path of length *d* must explore

$$1 + b + b^2 + b^3 + \ldots + b^d$$

nodes.

- Plus need b^d nodes in memory to store leaves in queue.

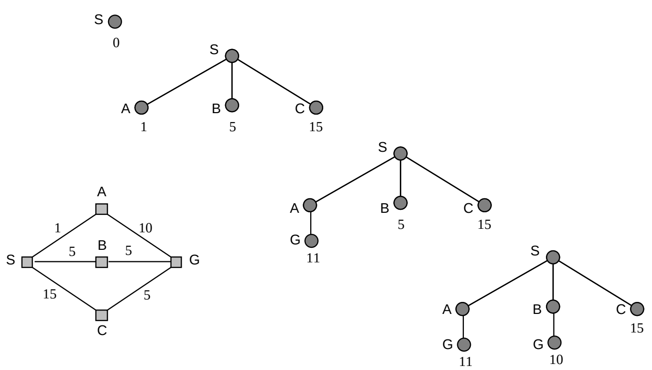
- Assuming can expand and check 1000 nodes/sec and need 100 bytes/node storage , *b*=10

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Note memory is a bigger problem than time.

Uniform Cost Search

- Like breadth-first except always expand node of least cost instead of of least depth (i.e. sort new queue by path cost).



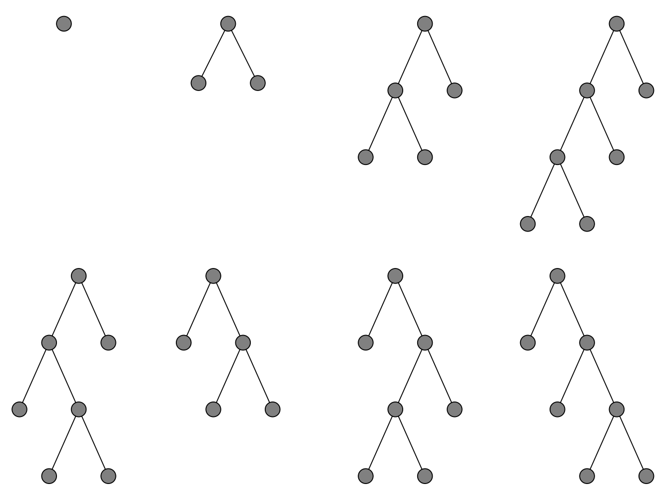
(a) (b)

- Do not recognize goal until it is the least cost node on the queue and removed for goal testing.

- Therefore, guarantees optimality as long as path cost never decreases as a path increases (non-negative operator costs).

Depth-First Search

- Always expand node at deepest level of the tree, i.e. one of the most recently generated nodes. When hit a dead-end, backtrack to last choice.



- Implemented by adding new nodes to front of the queue:

  GENERAL-SEARCH(problem, ENQUEUE-AT-FRONT)

Depth-First Properties

- Not guaranteed to be complete since might get lost following infinite path.
- Not guaranteed optimal since can find deeper solution before shallower ones explored.
- Time complexity in worst case is still O(b^d) since need to explore entire tree. But if many solutions exist may find one quickly before exploring all of the space.
- Space complexity is only O(bm) where m is maximum depth of the tree since queue just contains a single path from the root to a leaf node along with remaining sibling nodes for each node along the path.
- Can impose a **depth limit**, l, to prevent exploring nodes beyond a given depth. Prevents infinite regress, but incomplete if no solution within depth limit.

Iterative Deepening

- Conduct a series of depth-limited searches, increasing depth-limit each time.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
   inputs: problem, a problem

   for depth ← 0 to ∞ do
      if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
   end
   return failure
```

- Seems wasteful since work is repeated, but most work is at the leaves at each iteration and is not repeated.

Depth-first:

$$1 + b + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

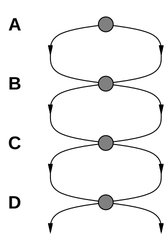
Iterative deepening:

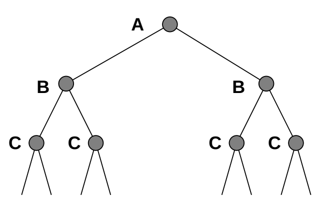
$$(d+1)1 + db + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

Time complexity is still O(b^d)
Space complexity O(bm)

Avoiding Repeated States

- Basic search methods may repeatedly search the same state if it can be reached via multiple paths.





- Three methods for reducing repeated work in order of effectiveness and computational overhead:
  - Do not follow self-loops (remove successors back to the same state).
  - Do no create paths with cycles (remove successors already on the path back to the root). O(d) overhead.
  - Do not generate any state that was already generated. Requires storing all generated states (O(b^d) space) and searching them (usually using a hash-table for efficiency).