Problem Solving and Search

Outline

- Introduction to Problem Solving
- Complexity
- Uninformed search
  - Problem formulation
  - Search strategies: depth-first, breadth-first
- Informed search
  - Search strategies: best-first, memory bounded
  - Heuristic functions

Building a Problem Solving Program

- Define the problem precisely
- Analyze the problem
- Represent the task knowledge
- Choose and apply representation and reasoning techniques

To Specify a Problem

- Define the state space
- Specify the initial states
- Specify the goal states
- Specify the operations

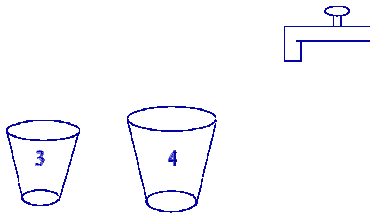
Production Systems

- A set of rules
- Knowledge/databases
- A control strategy
- A rule applier

Problem Characteristics

- Is the problem decomposable?
- Can solution steps be ignored or undone?
- Is the problem's universe predictable?
- Is a good solution absolute or relative?
- Is the desired solution a state of the world or a path to a state?
- Is a large amount of knowledge absolutely required to solve the problem, or is it important only to constrain the search?
- Must problem-solving be interactive?

The Water Jug Problem

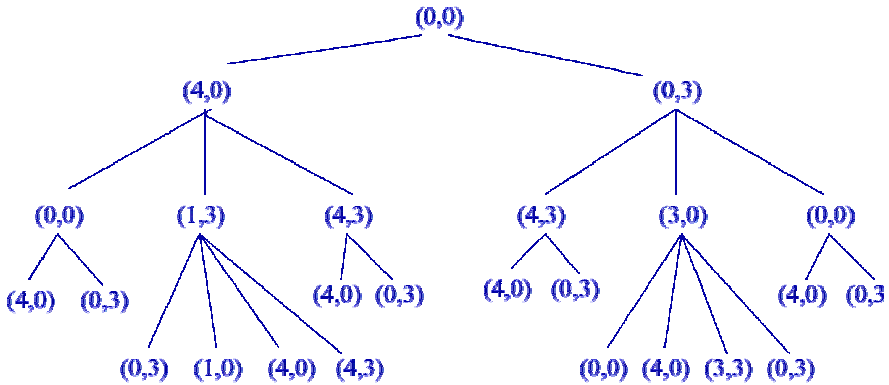


- **Given:**
  - one three- liter jug,
  - one four- liter jug
  - tap that can fill the jugs with water
- **Goal:** exactly two liters of water in the four- liter jug

Water Jug Problem

1. (X,Y: X < 4) -> (4,Y) Fill the 4-liter jug
2. (X,Y: Y < 3) -> (X,3) Fill the 3-liter jug
3. (X,Y: X > 0) -> (0,Y) Empty the 4-liter jug on the ground
4. (X,Y: Y > 0) -> (X,0) Empty the 3-liter jug on the ground
5. (X,Y: X+Y >= 4 and Y > 0) -> (4,Y-(4-X))
   Fill the 4-liter jug from the 3-liter jug
6. (X,Y: X+Y >= 3 and X > 0) -> (X-(3-Y),3))
   Fill the 3-liter jug from the 4-liter jug
7. (X,Y: X+Y <= 4 and Y > 0) -> (X+Y,0)
   Pour all water from the 3-liter jug into the 4-liter jug
8. (X,Y: X+Y <= 3 and X > 0) -> (0,X+Y))
   Pour all water from the 4-liter jug into the 3-liter jug
9. (X,Y: X > 0) -> (X-D,Y)
10. (X,Y: Y > 0) -> (X,Y-D)

Water Jug Problem



Examples of task environments and their characteristics.

Task Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Strategic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

Problem-Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq); seq ← REST(seq)
  return action
```

Note: This is *offline* problem-solving. *Online* problem-solving involves acting without complete knowledge of the problem and environment

Problem types

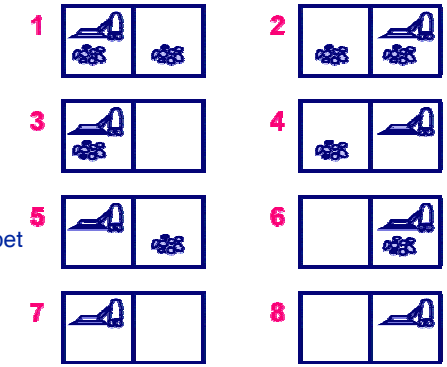
- Deterministic, fully observable ⇒ **single-state problem**
  - Agent knows exactly which state it will be in, solution is a sequence
- Non-observable ⇒ **sensorless problem** (conformant)
  - Agent may have no idea where it is, solution (if any) is a sequence
- Nondeterministic and/or partially observable ⇒ **contingency problem**
  - Percepts provide new information about current state
  - Solution is a contingent plan or a policy
  - Often interleave search, execution
- Unknown state space ⇒ **exploration problem ("online")**

Example: Vacuum world

Single-state, start in #5.
Solution?? [Right, Suck]

Sensorless, start in {1, 2, 3, 4, 5, 6, 7, 8}
e.g., Right goes to {2, 4, 6, 8}.
Solution?? [Right, Suck, Left, Suck]

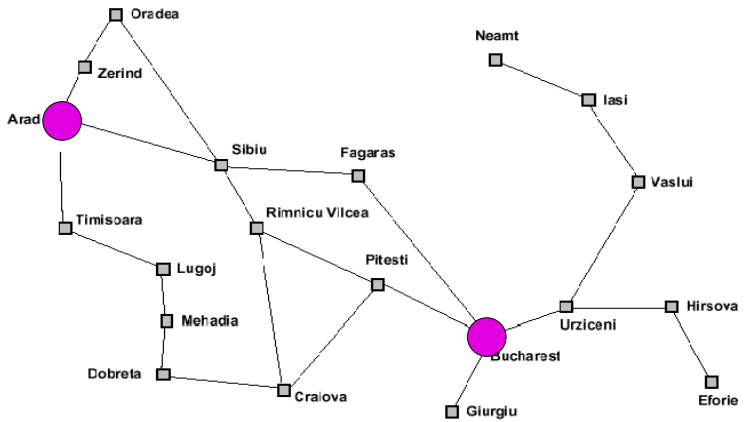
Contingency, start in #5
Murphy's Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution?? Initial belief state is {5, 7}
[Right, if dirt then Suck]



Example: Romania

- In Romania, on vacation. Currently in Arad.
- Flight leaves tomorrow from Bucharest.
- **Formulate goal:**
  - be in Bucharest
- **Formulate problem:**
  - states: various cities
  - operators: drive between cities
- **Find solution:**
  - sequence of cities, such that total driving distance is minimized.

Example: Traveling from Arad To Bucharest



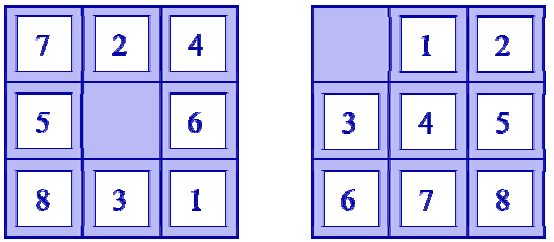
Well-defined problems

- The **initial state** that the agent starts in.
- A description of the possible **actions** available to the agent.
  - Given a particular state , SUCCESSOR-FN(x) returns a set of *action successor* ordered pairs, where each successor is a state that can be reached from x.
  - A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test**, which determines whether a given state is a goal state.
- A **path cost** function that assigns a numeric cost to each path.
- A **solution** to a problem is a path from the initial state to a goal state.
  - Solution quality is measured by the path cost function
  - **An optimal solution** has the lowest path cost among all solutions.

Selecting a state space

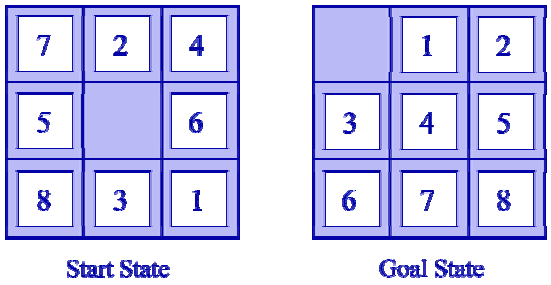
- Real world is absurdly complex
  - state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., “Arad ! Zerind" represents a complex set of possible routes, detours, rest stops, etc.
  - For guaranteed realizability, any real state “in Arad“ must get to some real state “in Zerind"
- (Abstract) solution
  - = sequence of abstract actions
  - = set of real paths that are solutions in the real world
- Each abstract action should be “easier" than the original problem!

Example: 8-puzzle



- **State:**
- **Operators:**
- **Goal test:**
- **Path cost:**

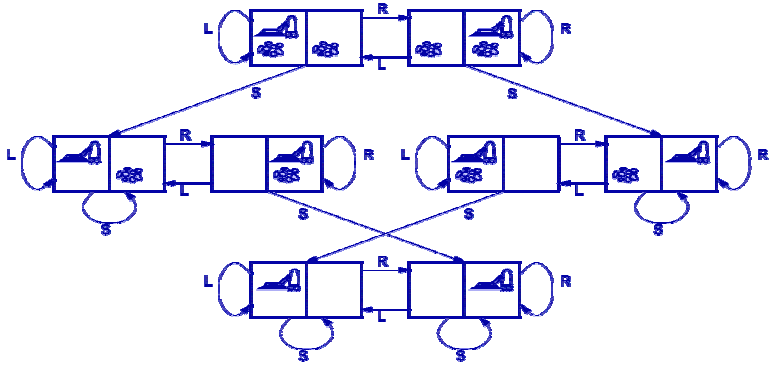
Example: 8-puzzle



- **State:** integer location of tiles (ignore intermediate locations)
- **Operators:** moving blank left, right, up, down (ignore jamming)
- **Goal test:** does state match goal state?
- **Path cost:** 1 per move

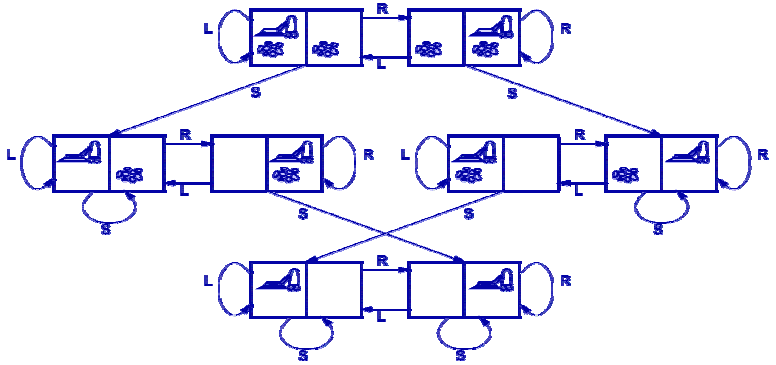
[Note: optimal solution of *n*-Puzzle family is NP-hard]

Back to Vacuum World



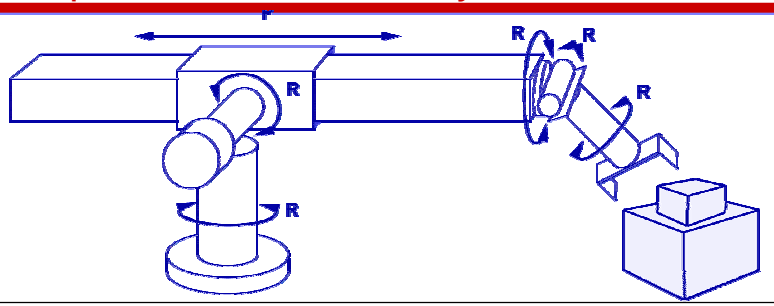
- **State:**
- **Operators:**
- **Goal test:**
- **Path cost:**

Back to Vacuum World



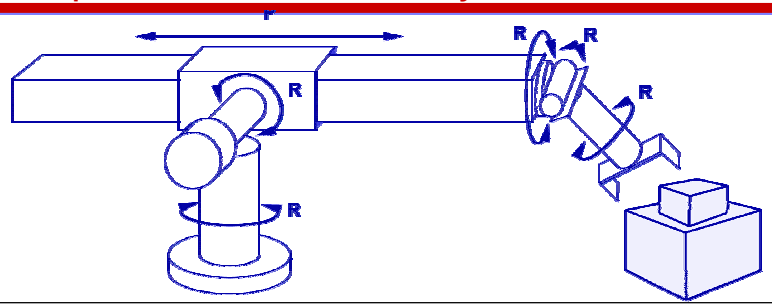
- **State:** integer dirt and robot locations (ignore dirt amounts etc.)
- **Operators:** *Left, Right, Suck, NoOp*
- **Goal test:** no dirt
- **Path cost:** 1 per action (0 for *NoOp*)

Example: Robotic Assembly



- **State:**
- **Operators:**
- **Goal test:**
- **Path cost:**

Example: Robotic Assembly

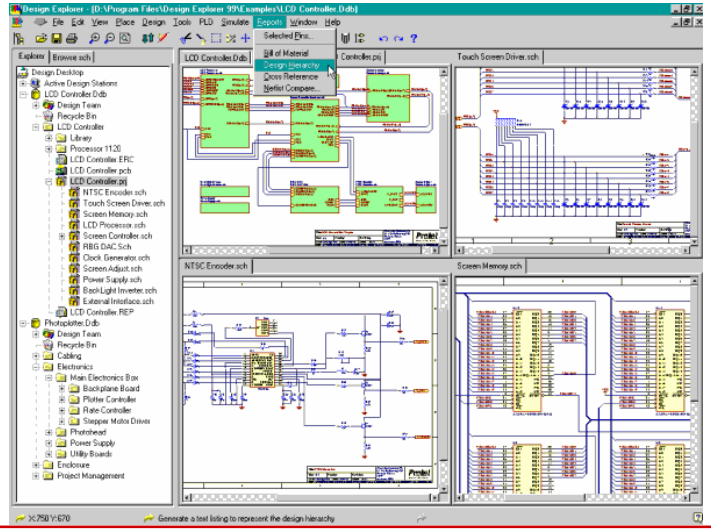


- **State:** real-valued coordinates of robot joint angles and parts of the object to be assembled
- **Operators:** continuous motions of robot joints
- **Goal test:** complete assembly with no robot included!
- **Path cost:** time to execute

Real-life example: VLSI Layout

- Given schematic diagram comprising components (chips, resistors, capacitors, etc) and interconnections (wires), find optimal way to place components on a printed circuit board, under the constraint that only a small number of wire layers are available (and wires on a given layer cannot cross!)
- "optimal way"??
  - minimize surface area
  - minimize number of signal layers
  - minimize number of connections from one layer to another
  - minimize length of some signal lines (e.g., clock line)
  - distribute heat throughout board
  - etc.

A Design Tool



Search algorithms

Basic idea: offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

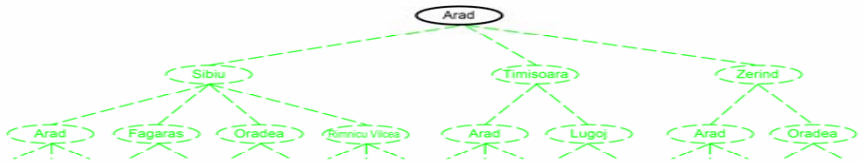
Function General-Search(*problem*, *strategy*) returns a *solution*, or failure
initialize the search tree using the initial state problem
loop do
if there are no candidates for expansion **then return** failure
choose a leaf node for expansion according to strategy
if the node contains a goal state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

Tree Search Algorithms

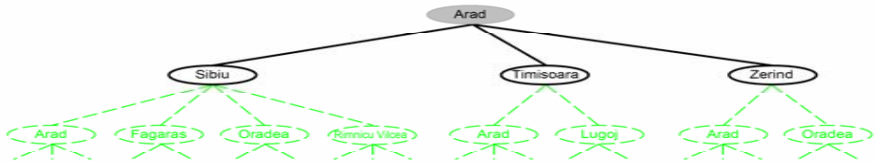
function TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
loop do
if *fringe* is empty **then return** failure
node ← REMOVE-FRONT(*fringe*)
if GOAL-TEST(*problem*, STATE(*node*)) **then return** SOLUTION(*node*)
fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

function EXPAND(*node*, *problem*) **returns** a set of nodes
successors ← the empty set; *state* ← STATE[*node*]
for each *action*, *result* **in** SUCCESSOR-FN(*problem*, *state*) **do**
s ← a new NODE
PARENT-NODE[*s*] ← *node*; ACTION[*s*] ← *action*; STATE[*s*] ← *result*
PATH-COST[*s*] ← PATH-COST[*node*]+STEP-COST(*state*, *action*, *result*)
DEPTH[*s*] ← DEPTH[*node*] + 1
add *s* to *successors*
return *successors*

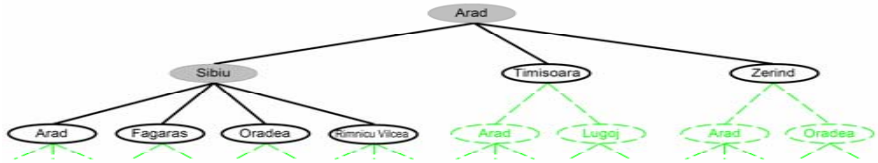
General search example



General search example

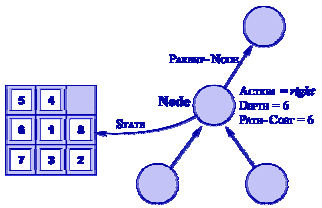


General search example



Implementation: states vs. nodes

A state is a (representation of) a physical conguration
A node is a data structure constituting part of a search tree



- STATE: the state in the state space to which the node corresponds.
- PARENT-NODE: the node in the search tree that generated this node.
- ACTION: the action that was applied to the parent to generate the node.
- PATH-COST: the cost of the path from the initial state to the node, as indicated by the parent pointers. The path cost is traditionally denoted by g(n)
- DEPTH: the number of steps along the path from the initial state.

Evaluation of search strategies

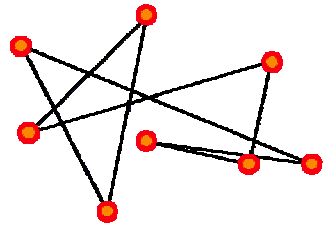
- A search strategy is defined by picking the order of node expansion.
- Search algorithms are commonly evaluated according to the following four criteria:
  - Completeness: does it always find a solution if one exists?
  - Time complexity: number of nodes generated/expanded
  - Space complexity: maximum number of nodes in memory
  - Optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of:
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - C*: path cost of the least-cost solution
  - m: maximum depth of the state space (may be infinity)

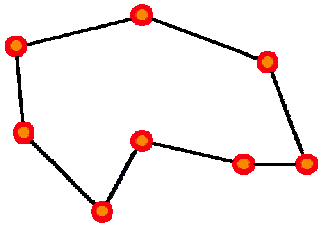
Complexity

- Why worry about complexity of algorithms?
  - because a problem may be solvable in principle but may take too long to solve in practice
- How can we evaluate the complexity of algorithms?
  - through asymptotic analysis, i.e., estimate time (or number of operations) necessary to solve an instance of size n of a problem when n tends towards infinity

Complexity example: Traveling Salesman Problem

- There are n cities, with a road of length Lij joining city i to city j.
- The salesman wishes to find a way to visit all cities that is optimal in two ways: each city is visited only once, and the total route is as short as possible.





- This is a *hard* problem: the only known algorithms (so far) to solve it have exponential complexity, that is, the number of operations required to solve it grows as e^n for n cities.

Why is exponential complexity "hard"?

It means that the number of operations necessary to compute the exact solution of the problem grows exponentially with the size of the problem (here, the number of cities).

- $e^{1} = 2.72$
- $e^{10} = 2.20\ 10^{4}$ (daily salesman trip)
- $e^{100} = 2.69\ 10^{43}$ (monthly salesman planning)
- $e^{500} = 1.40\ 10^{217}$ (music band worldwide tour)
- $e^{250,000} = 10^{108,573}$ (postal services)
- Fastest computer $= 10^{12}$ operations/second

So…

In general, exponential-complexity problems *cannot be solved for any but the smallest instances!*

Complexity

- Polynomial-time (P) problems: we can find algorithms that will solve them in a time (=number of operations) that grows polynomially with the size of the input.
  - for example: sort n numbers into increasing order: poor algorithms have n^2 complexity, better ones have $n \log(n)$ complexity.
- Since we did not state what the order of the polynomial is, it could be very large! Are there algorithms that require more than polynomial time?
- Yes (until proof of the contrary), for some algorithms, we do not know of any polynomial-time algorithm to solve them. These are referred to as non-polynomial-time (NP) algorithms.
  - for example: traveling salesman problem.
- In particular, exponential-time algorithms are believed to be NP.

Note on NP-hard problems

- The formal definition of NP problems is:
  - A problem is nondeterministic polynomial if there exists some algorithm that can guess a solution and then verify whether or not the guess is correct in polynomial time.
- (one can also state this as these problems being solvable in polynomial time on a nondeterministic Turing machine.)
- In practice, until proof of the contrary, this means that known algorithms that run on known computer architectures will take more than polynomial time to solve the problem.

Complexity and the human brain

- Are computers close to human brain power?
- Current computer chip (CPU):
  - 10^3 inputs (pins)
  - 10^7 processing elements (gates)
  - 2 inputs per processing element (fan-in = 2)
  - processing elements compute boolean logic (OR, AND, NOT, etc)
- Typical human brain:
  - 10^7 inputs (sensors)
  - 10^{10} processing elements (neurons)
  - fan-in = 10^3
  - processing elements compute complicated functions

Still a lot of improvement needed for computers, but computer clusters come close!

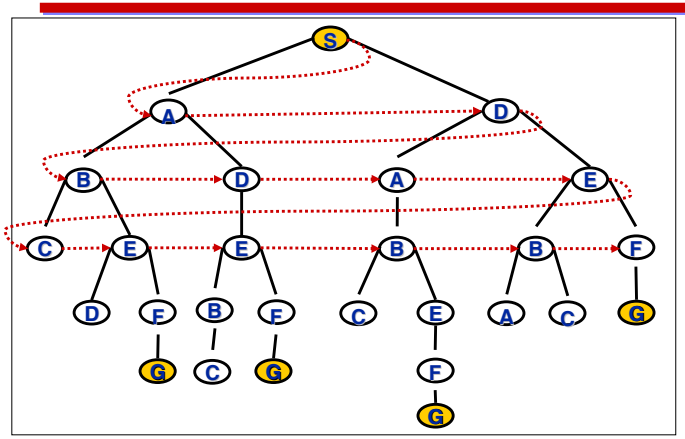
Uninformed search strategies

- Uninformed strategies use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

Breadth first Search

- **Breadth-first search** is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level.
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

Breadth-first search:



- Move downwards, level by level, until goal is reached.

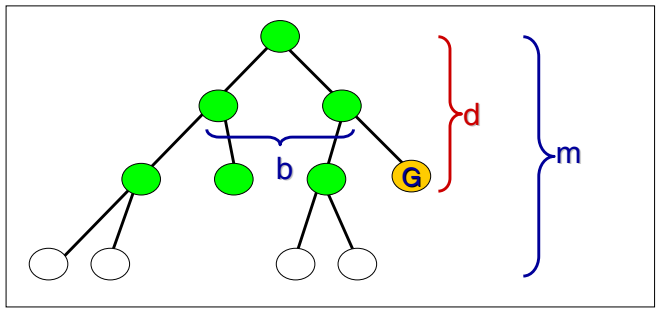
Properties of breadth-first search

- Completeness: Yes, if *b* is finite
- Time complexity: $1+b+b^2+\ldots+b^d = O(b^{d+1})$
- Space complexity: $O(b^{d+1})$ (keeps every node in memory)
- Optimality: No, unless step costs are constant

- Space is the big problem; can easily generate nodes at 100MB/sec
- so in 24hrs 8640GB can be generated.

Time complexity

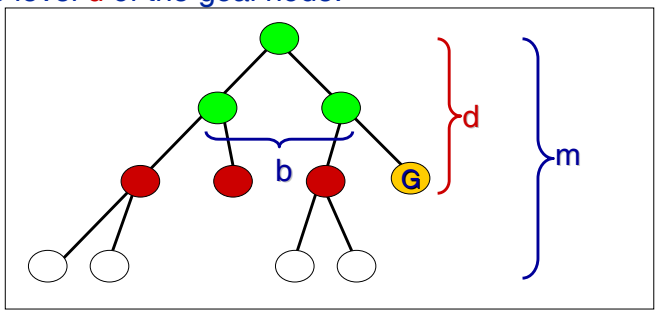
- If a goal node is found on depth d of the tree, all nodes up till that depth are created.



- Thus: $O(b^d)$

Space complexity

- Largest number of nodes in QUEUE is reached on the level d of the goal node.



- QUEUE contains all  and  nodes. (Thus: 4) .
- In General: b^d

Time and memory requirements for breadth-first search

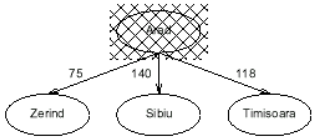
Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

The numbers shown assume branching factor b=10, 10,000 nodes/second, 1000 bytes/node.

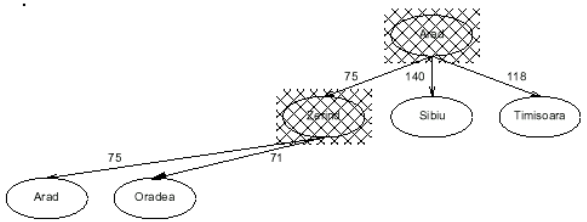
Uniform-cost search

- Uniform-cost search expands the node with the lowest path cost, instead of expanding the shallowest node.
- A refinement of the breadth-first strategy:
- Breadth-first = uniform-cost with path cost = node depth

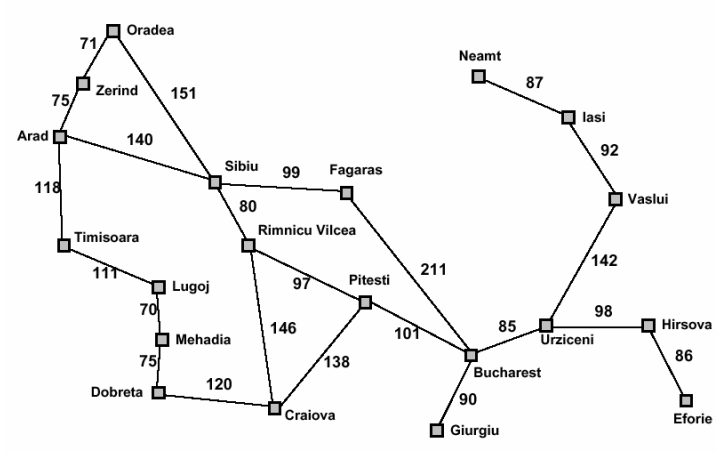
Uniform-cost search



Uniform-cost search

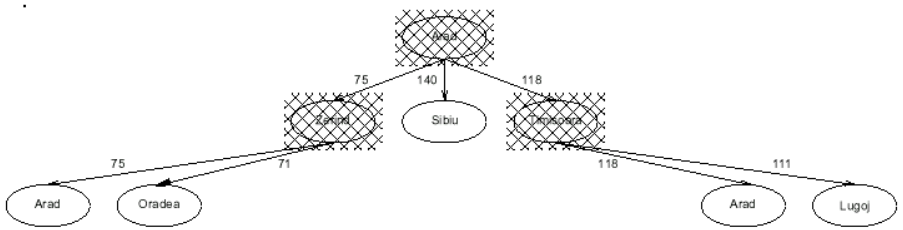


Romania with step costs in km



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

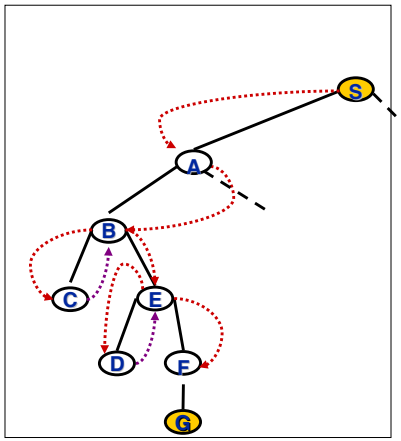
Uniform-cost search



Properties of uniform-cost search

- Completeness: Yes, if step cost $\geq \varepsilon > 0$
- Time complexity: # nodes with g ≤ cost of optimal solution, $\leq O(b^{\lceil C^*/\epsilon \rceil})$
  where C^* is the cost of theoptimal solution
- Space complexity: # nodes with g ≤ cost of optimal solution, $\leq O(b^{\lceil C^*/\epsilon \rceil})$
- Optimality: Yes, nodes expanded in increasing order of g(n)

Depth-first search



- Select a child
  - convention: left-to-right
- Repeatedly go to next child, as long as possible.
- Return to left-over alternatives (higher-up) only when needed.

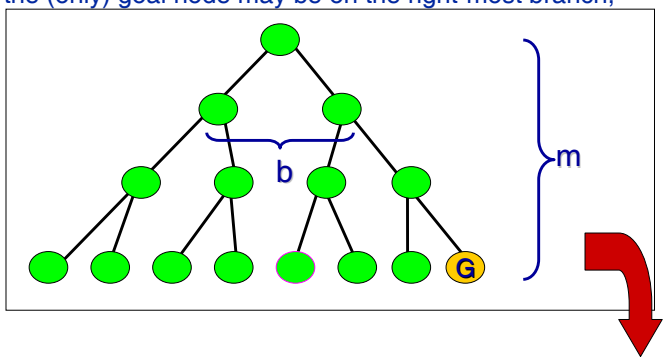
Implementation:
fringe = LIFO queue, i.e., put successors at front

Properties of depth-first search

- **Completeness:** No
  - fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
  - complete in finite spaces
- **Time complexity:** $O(b^m)$
  - terrible if *m* is much larger than *d*
  - but if solutions are dense, may be much faster than breadth-first
- **Space complexity:** O(bm)
  - i.e., linear space!
- **Optimality:** No

Time complexity

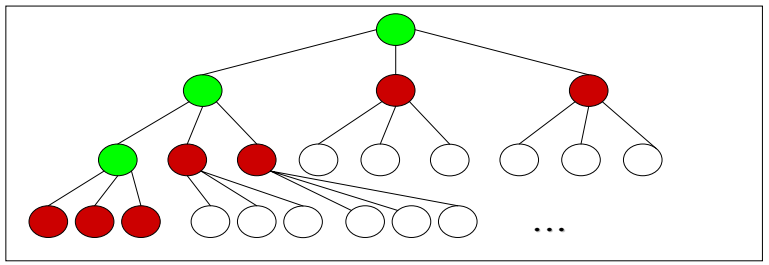
- In the worst case:
  - the (only) goal node may be on the right-most branch,



- Time complexity == $b^m + b^{m-1} + \ldots + 1 = \frac{b^{m+1} - 1}{b - 1}$
- Thus: $O(b^m)$

Space complexity

- Largest number of nodes in QUEUE is reached in bottom left-most node.
- Example: m = 3, b = 3 :



- QUEUE contains all  nodes. Thus: 7.
- In General: ((b-1) * m) + 1
- Order: O(bm)

Depth-limited search

- Is a depth-first search with depth limit λ
- Implementation: Nodes at depth λ have no successors.
- Complete: if cutoff chosen appropriately then it is guaranteed to find a solution.
- Optimal: it does not guarantee to find the least-cost solution

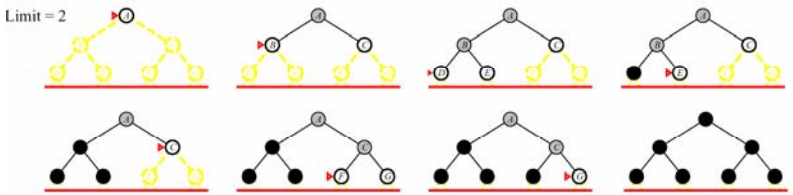
Depth limited search

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

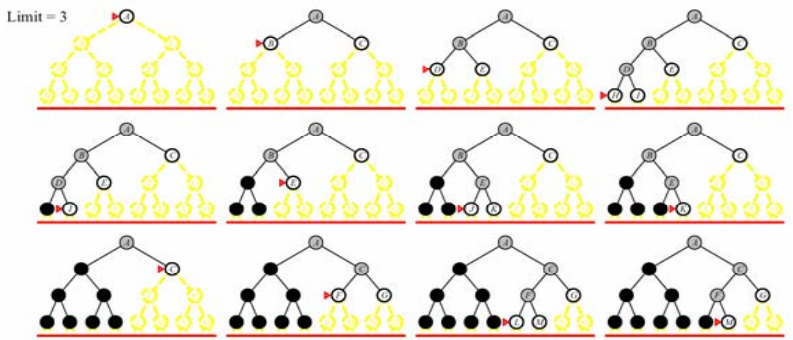

Depth limited search example

Depth limited search example

Depth limited search example



Depth limited search example



Iterative deepening search

Combines the best of breadth-first and depth-first search strategies.

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

- Completeness: Yes,
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$

Optimality: No, unless step costs are constant

Can be modied to explore uniform-cost tree

Iterative deepening complexity

- Iterative deepening search may seem wasteful because so many states are expanded multiple times.
- In practice, however, the overhead of these multiple expansions is small, because most of the nodes are towards leafs (bottom) of the search tree:
- thus, the nodes that are evaluated several times (towards top of tree) are in relatively small number.
- In iterative deepening, nodes at bottom level are expanded once, level above twice, etc. up to root (expanded d+1 times) so total number of expansions is:

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

- In general, iterative deepening is preferred to depth-first or breadth-first when search space large and depth of solution not known.

Bidirectional search

- Both search forward from initial state, and backwards from goal.
- Stop when the two searches meet in the middle.

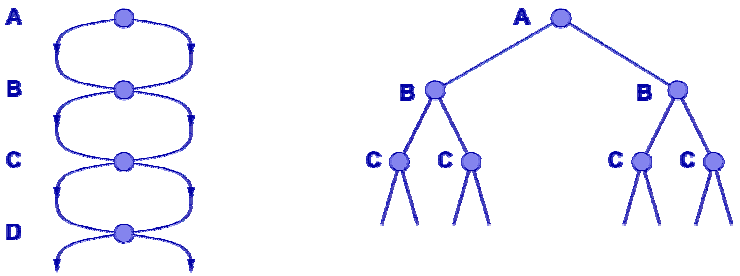
- Problem: how do we search backwards from goal??
  - predecessor of node n = all nodes that have n as successor
  - this may not always be easy to compute!
  - if several goal states, apply predecessor function to them just as we applied successor (only works well if goals are explicitly known, may be difficult if goals only characterized implicitly).

Evaluation of search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	No*	Yes	No	No	No*

Repeated states

- Failure to detect repeated states can cause exponentially more work!



Avoiding repeated states

In increasing order of effectiveness and computational overhead:

- **do not return to state we come from,** i.e., expand function will skip possible successors that are in same state as node's parent.
- **do not create paths with cycles,** i.e., expand function will skip possible successors that are in same state as any of node's ancestors.
- **do not generate any state that was ever generated before,** by keeping track (in memory) of every state generated.

Graph search

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

- Use hash table for *closed* — constant-time lookup!
- Makes all algorithms complete in finite spaces!!
- Makes all algorithms worst-case exponential space!!!
- But size of graph often much less than $O(b^d)$!!!!

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
- Graph search can be exponentially more efficient than tree search

Searching With Partial Information

- **Sensorless problems** (also called **conformant problems**):
  - If the agent has no sensors at all, then it may be in one of several possible initial states, and each action may therefore lead to one of several possible successor states.
- **Contingency problems**:
  - If there is uncertainty about action outcomes, or if the environment is partially observable, then the agent's percepts provide *new* information after each action.
  - Each possible percept defines a contingency that must be planned for.
  - A problem is called **adversarial** if the uncertainty is caused by the hostile actions of another agent.
- **Exploration problems**:
  - When the states and actions of the environment are unknown, the agent must act to discover them.
  - Exploration problems can be viewed as an extreme case of contingency problems.