Artificial Intelligence

Uninformed Search

Overview

- Searching for Solutions
- Uninformed Search
  - BFS
  - UCS
  - DFS
  - DLS
  - IDS

Searching for Solutions

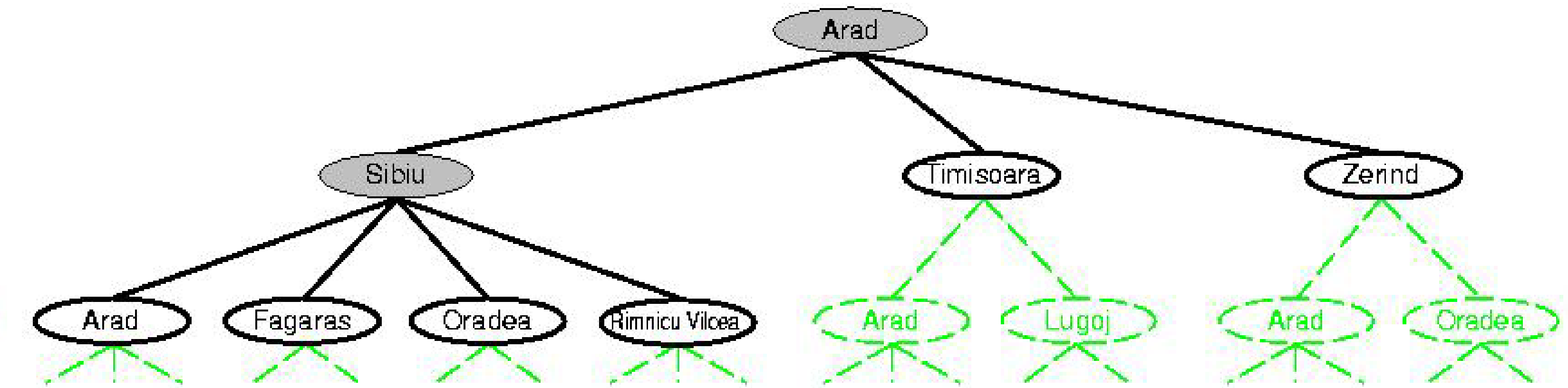
Tree Search Algorithms

Basic idea:

offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. *expanding* states)

```
function TREE-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
```

Tree Search Example



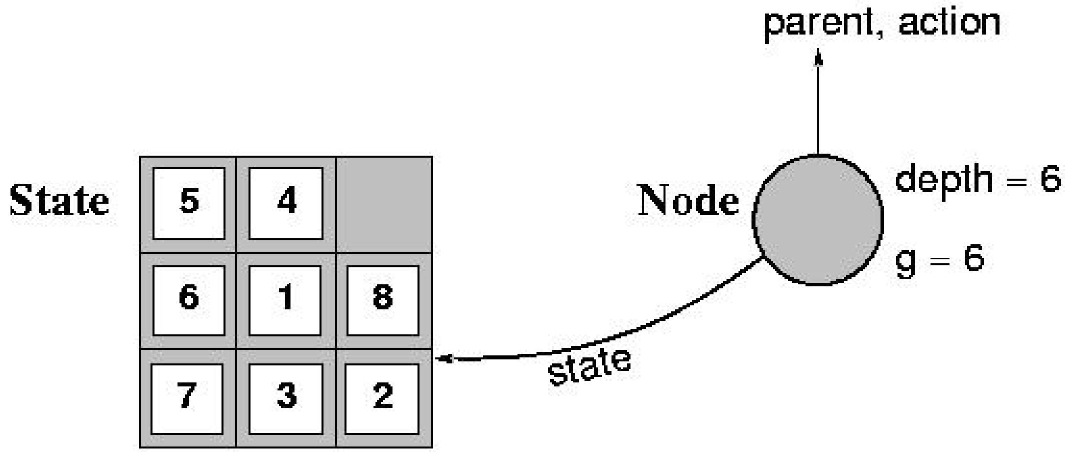
Implementation: State vs. Nodes

A *state* is a (representation of) a physical configuration

A *node* is a data structure constituting part of a search tree

includes *parent*, *children*, *depth*, *path cost* $g(x)$ ***and state***

States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFN of the problem to create the corresponding states.

General Tree Search

```
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] applied to STATE(node) succeeds return node
       fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

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

Search Strategies

A strategy is defined by picking the *order of node expansion*

Strategies are evaluated along the following dimensions:

- 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
- m—maximum depth of the state space (may be ∞)

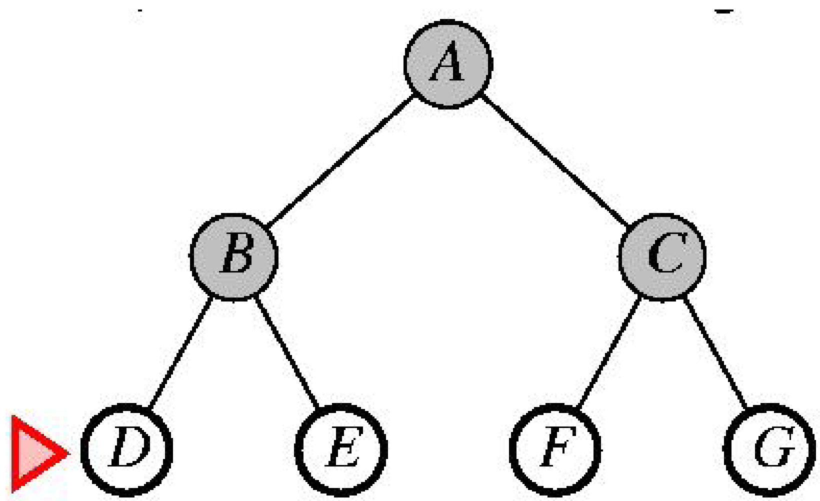
Uninformed Search

Uninformed Search Strategies

- Use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limit search
  - Iterative deepening search

Breadth-First Search

- Expand shallowest unexpanded node
- Implementation: fringe is a FIFO queue, that is new successors go at end



Properties of BFS

- Complete?
  - Yes. (if b is finite)
- Time?
  - $1+b+b^2+\ldots+b(b^d-1)=O(b^{d+1})$, exp in d
- Space?
  - $O(b^{d+1})$ (keeps every node in memory)
- Optimal?
  - Yes, if cost = 1 per step; not optimal in general when actions have different cost
- Space is the big problem; can easily generate nodes at 10MB/sec, so 24hrs = 860GB

Uniform-Cost Search

Expand least-cost unexpanded node

Implementation:

$\quad fringe =$ queue ordered by path cost

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

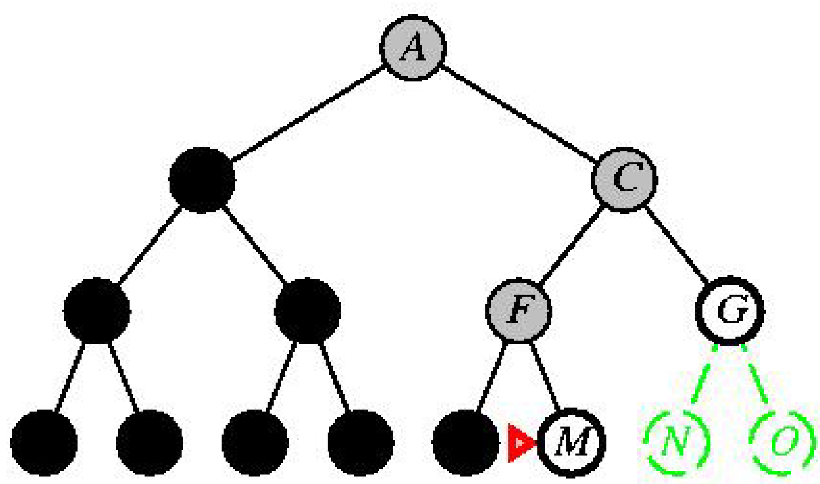
Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-First Search

- Expanded deepest unexpanded node
- Implementation:
  - Fringe = LIFO queue, i.e. put successors at front



Properties of DFS

- Complete?
  - No. Fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path => complete in finite space
- Time?
  - $O(b^m)$, terrible if m is much larger than d, but if solutions are dense, may be much faster than BFS
- Space?
  - $O(bm)$, linear space
- Optimal?
  - No

Depth-Limit Search

- Depth-first search with depth limit L, that is nodes at depth L have no successors

Depth-Limited search

- Is DF-search with depth limit *l*.
  - i.e. nodes at depth *l* have no successors.
  - Problem knowledge can be used
- Solves the infinite-path problem.
- If $l < d$ then incompleteness results.
- If $l > d$ then not optimal.
- Time complexity:
- Space complexity:

$$O(b^l)$$

$$O(bl)$$

Depth-Limit Search

- Algorithm

Recursive implementation:

```
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
```

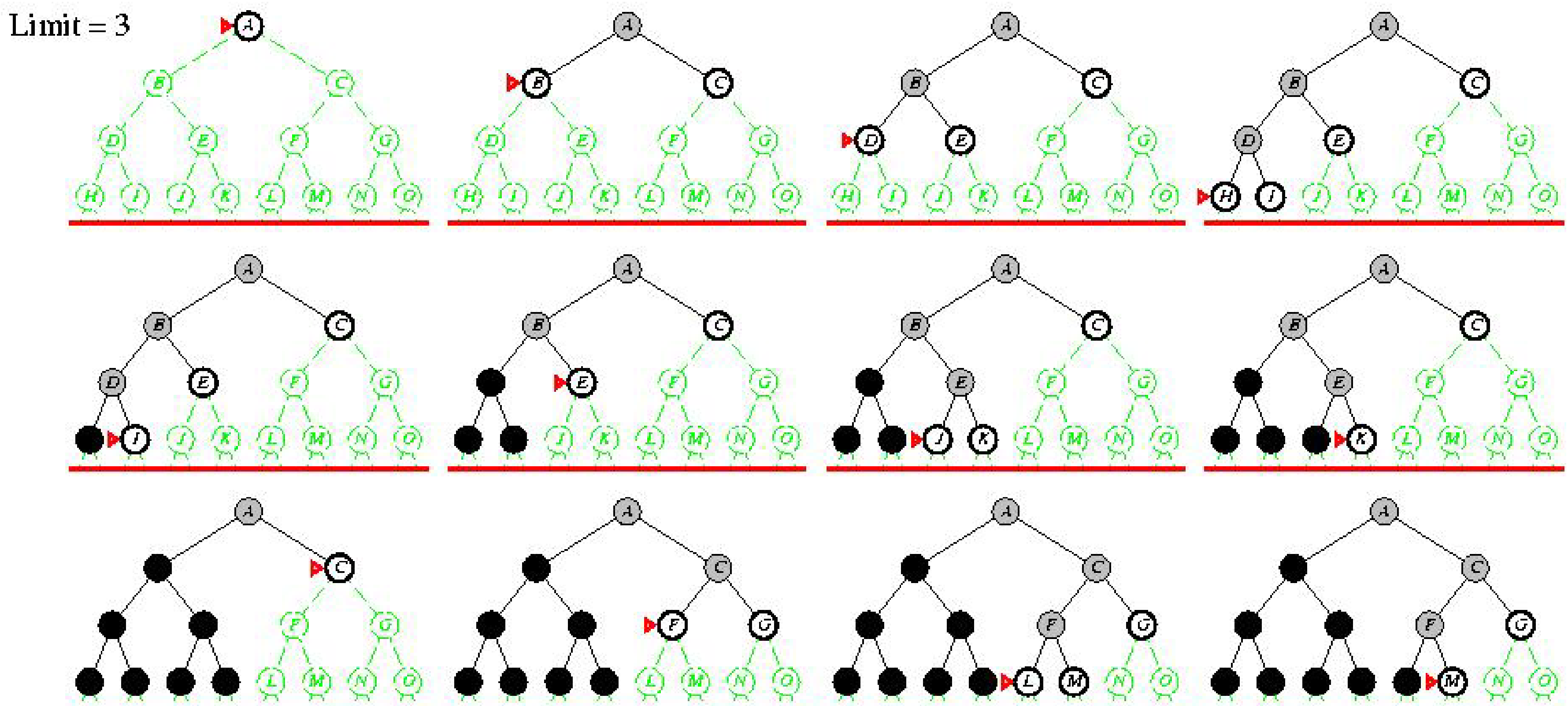
Iterative Deepening Search

- Iterative deepening search
  - A general strategy to find best depth limit *l*.
    - Goals is found at depth *d*, the depth of the shallowest goal-node.
  - Often used in combination with DF-search
- Combines benefits of DF- en BF-search

```
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
```

Iterative Deepening Search



Properties of IDS

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

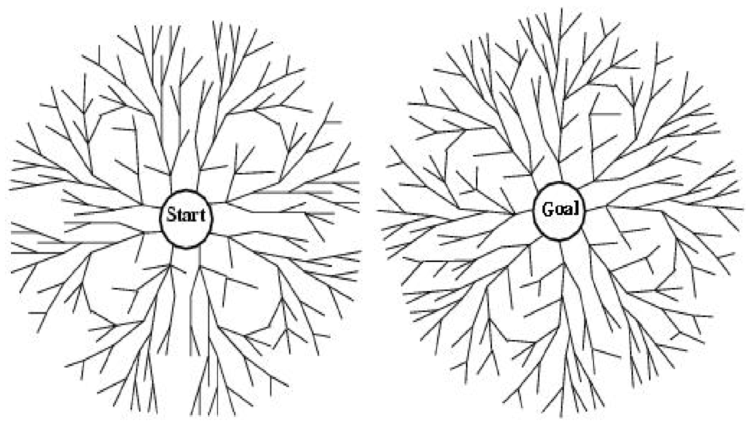
Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right:

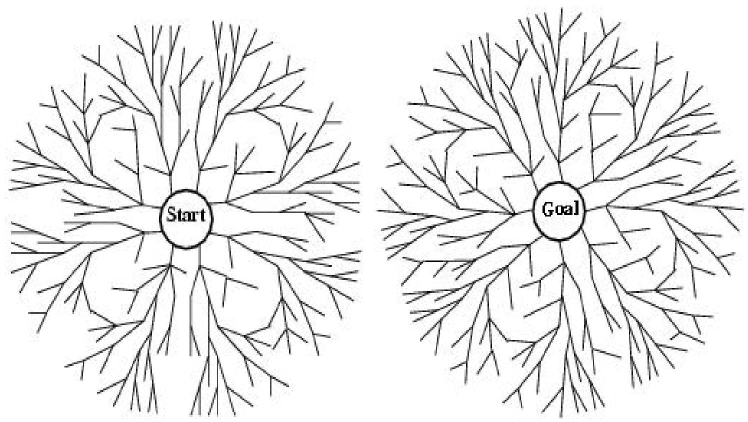
$$
\begin{aligned}
N(\mathsf{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\
N(\mathsf{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100
\end{aligned}
$$

Bidirectional search



- Two simultaneous searches from start an goal.
  - Motivation: $b^{d/2} + b^{d/2} \neq b^{d}$
- Check whether the node belongs to the other fringe before expansion.
- Space complexity is the most significant weakness.
- Complete and optimal if both searches are BF.

How to search backwards?



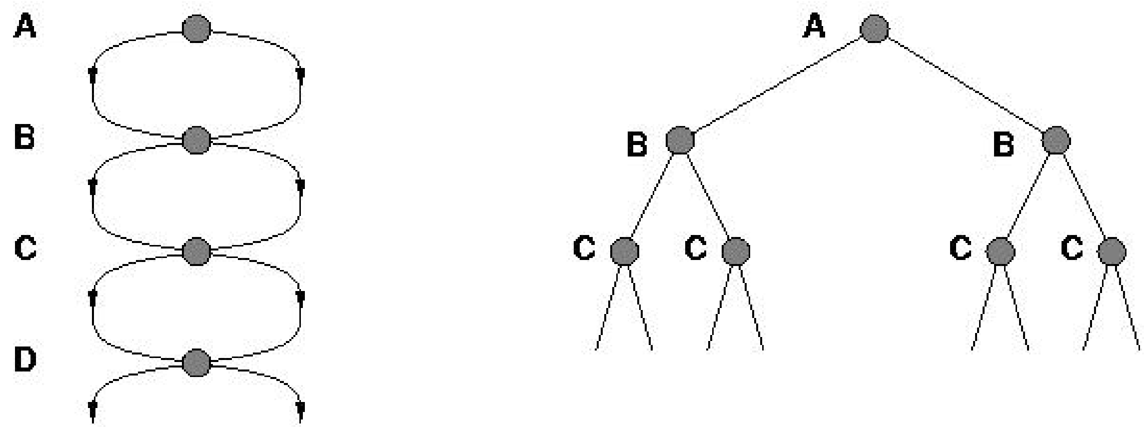
- The predecessor of each node should be efficiently computable.
  - When actions are easily reversible.

Summary of Algorithms

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete?	**YES***	**YES***	**NO**	**YES, if l $\geq$ d**	**YES**	**YES***
Time	b^{d+1}	$b^{C^*/e}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}	$b^{C^*/e}$	bm	bl	bd	$b^{d/2}$
Optimal?	**YES***	**YES***	**NO**	**NO**	**YES**	**YES**

Repeated States

Failure to detect repeated states can turn a linear problem into an exponential one!



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
```

Summary

- Searching for Solutions
- Uninformed Search
  - BFS
  - UCS
  - DFS
  - DLS
  - IDS