Artificial Intelligence: Search
Part 1: Uninformed graph search

Thomas Trappenberg

January 8, 2009

Based on the slides provided by Russell and Norvig, Chapter 3

Search outline

- Part 1: Uninformed search (tree search, graph search, etc)
- Part 2: Heuristic search (A*, etc)
- Part 3 Optimization search algorithms (gradient decent, GA, etc)

Selecting a state space

Real world is absurdly complex
$\Rightarrow$ state space must be **abstracted** for problem solving

(Abstract) state $\Leftrightarrow$ set of real states

(Abstract) action $\Leftrightarrow$ complex combination of real actions
e.g., "Halifax $\rightarrow$ Hawaii" represents a complex set
of possible routes, detours, rest stops, anticipatory emotional, etc.
(Abstract) solution $\Leftrightarrow$
set of real paths that are solutions in the real world

Each abstract action should be "easier" than the original problem!

Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

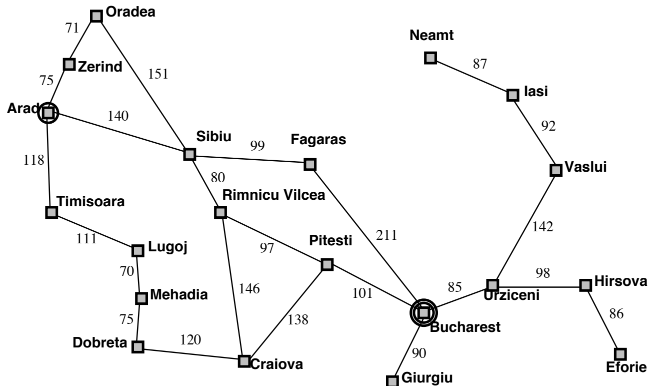
1	2	3
4	5	6
7	8	

Goal State

states: integer locations of tiles (ignore intermediate positions)
actions: move blank left, right, up, down (ignore unjamming etc.)
goal test: = goal state (given)
path cost: 1 per move

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

Example: Romania

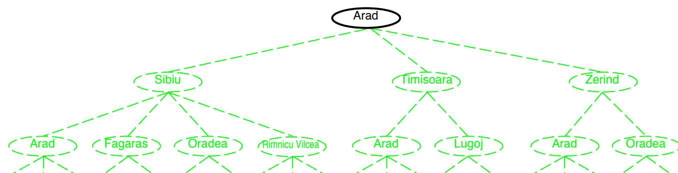


Tree search algorithms

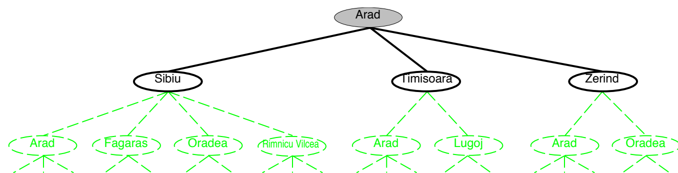
Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

```
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
```

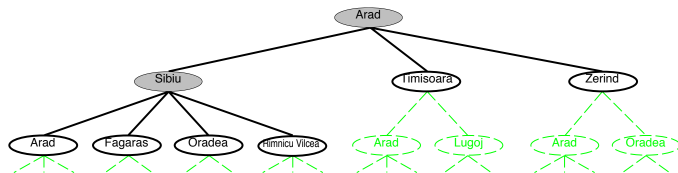
Tree search example



Tree search example

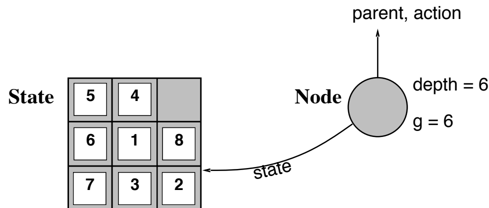


Tree search example



Implementation: states 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 $g(x)$
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.

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 strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

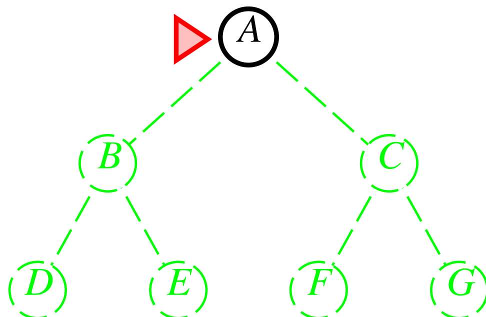
Depth-limited search

Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

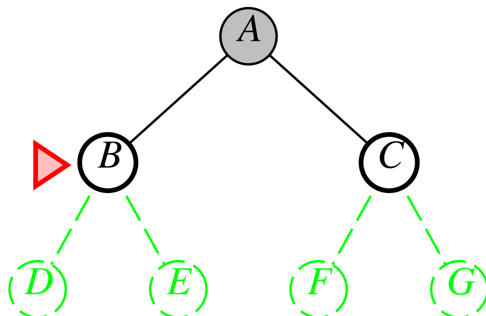
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



Breadth-first search

Expand shallowest unexpanded node

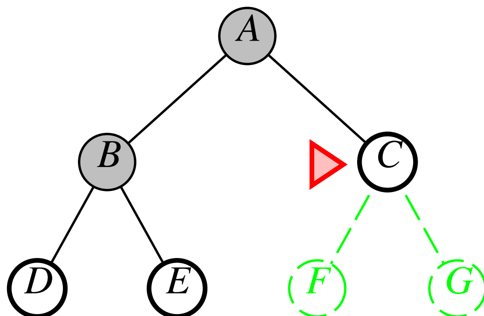
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



Breadth-first search

Expand shallowest unexpanded node

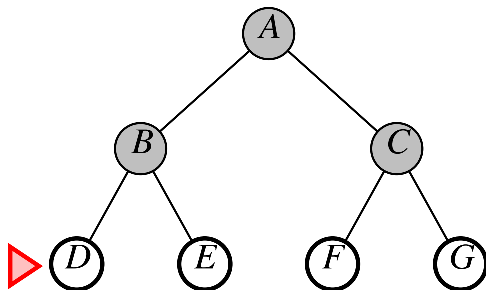
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



Breadth-first search

Expand shallowest unexpanded node

Implementation:
fringe is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

Complete Yes (if b is finite)

Time $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space $O(b^{d+1})$ (keeps every node in memory)

Optimal Yes (if cost = 1 per step); not optimal in general

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

Uniform-cost search

Expand least-cost unexpanded node

Implementation:
fringe = queue ordered by path cost, lowest first

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

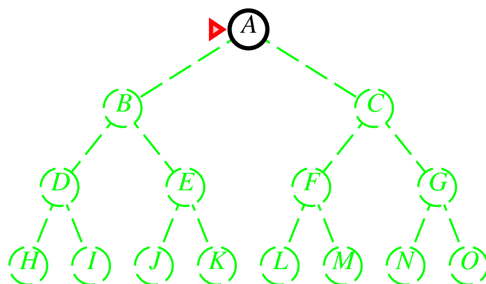
Space # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

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

Depth-first search

Expand deepest unexpanded node

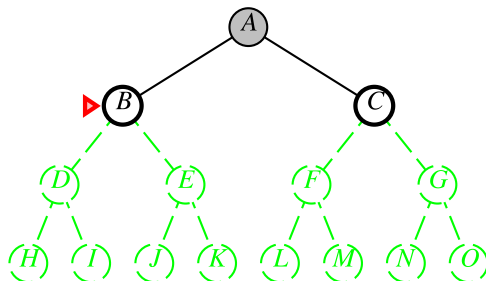
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

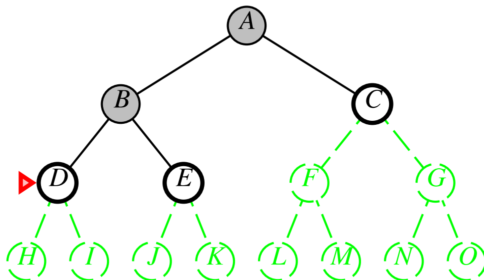
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

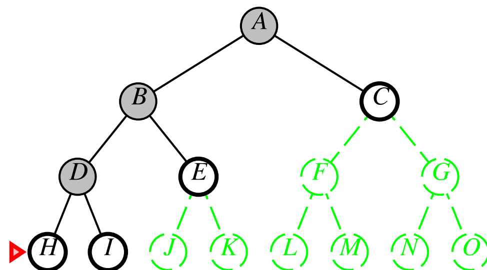
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

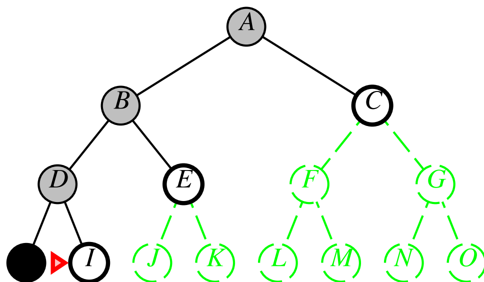
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

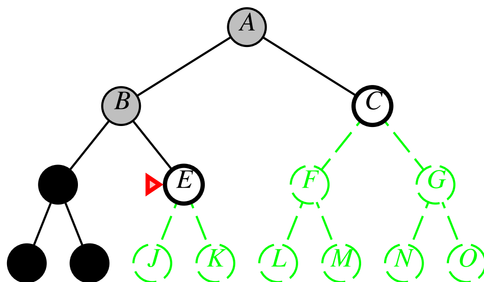
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

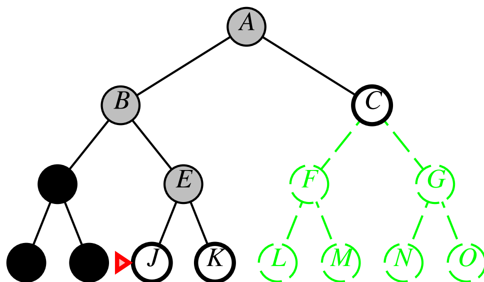
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

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

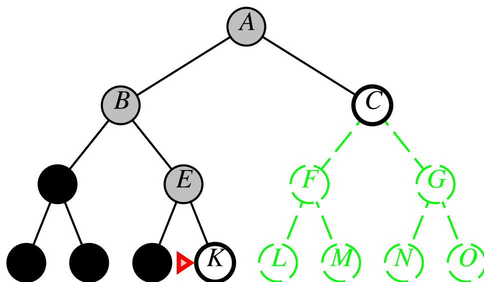


Depth-first search

Expand deepest unexpanded node

Implementation:

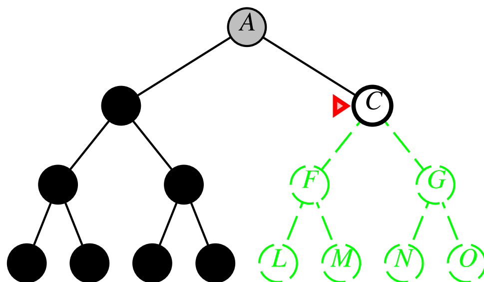
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

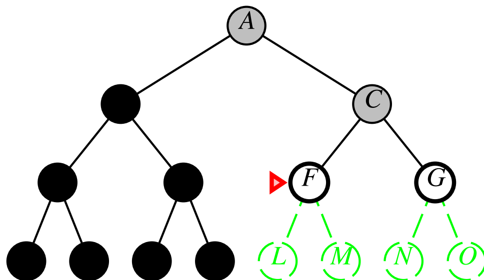
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

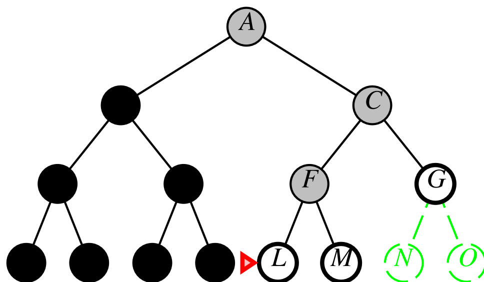
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

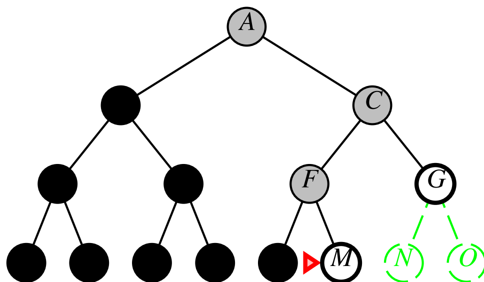
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

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



Properties of depth-first search

Complete No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ complete in finite spaces

Time $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space $O(bm)$, i.e., linear space!

Optimal No

Depth-limited search

= depth-first search with depth limit l,
i.e., nodes at depth l have no successors

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
```

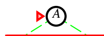
Iterative deepening search

```
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
```

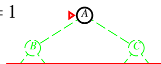
Iterative deepening search $l = 0$

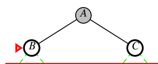
Limit = 0

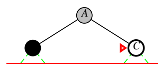


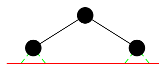
Iterative deepening search $l = 1$

Limit = 1

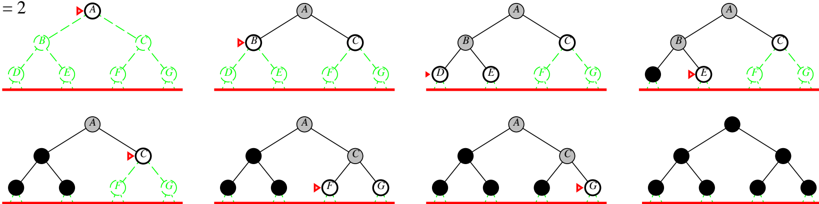




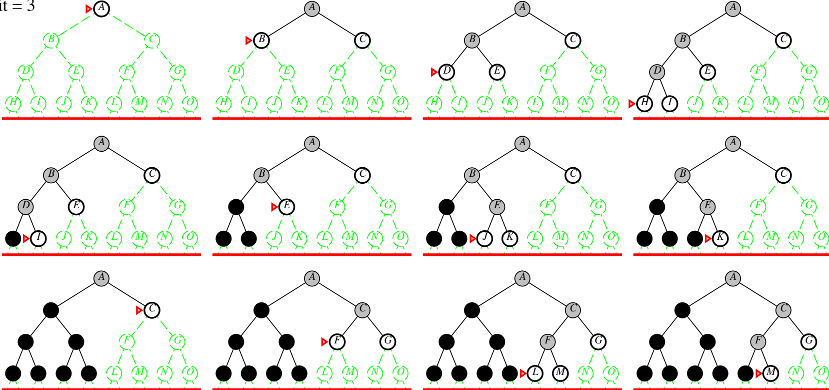




Iterative deepening search $l = 2$



Iterative deepening search $l = 3$



Properties of iterative deepening search

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 leaf:

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

IDS does better because other nodes at depth d are not expanded

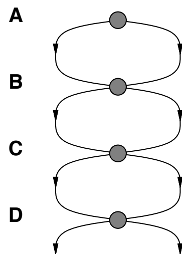
BFS can be modified to apply goal test when a node is **generated**

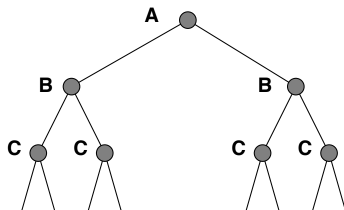
Summary of algorithms

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?	Yes*	Yes	No	No	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
```

More search algorithms

bi-directional search, ...

see http://en.wikipedia.org/wiki/Graph_traversal

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