3

Search Strategies

Claudia Chirita

School of Informatics, University of Edinburgh

Based on slides by: Jacques Fleuriot, Michael Rovatsos, Michael Herrmann, Vaishak Belle

3.a

Uninformed search. Breadth-first search

SEARCH STRATEGIES

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

EVALUATION

Strategies are evaluated along the following dimensions:

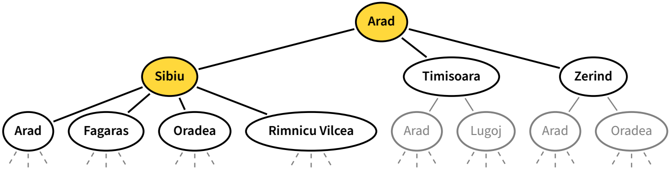
completeness	does it always find a solution if one exists?
time complexity	number of nodes generated/expanded
space complexity	maximum number of nodes in memory
optimality	does it always find a least-cost solution?

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 ∞)

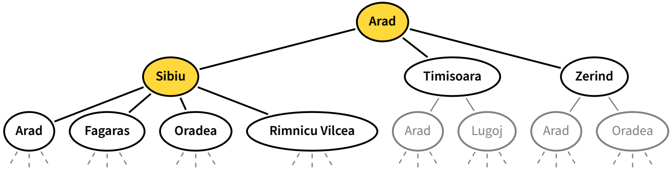
RECALL · TREE SEARCH

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```



RECALL · TREE SEARCH

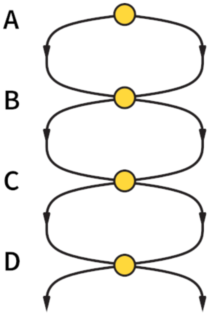
function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

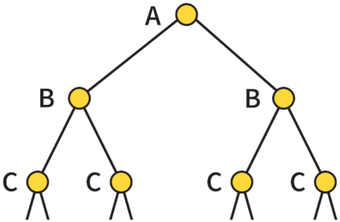


☠ Arad is a repeated state!

REPEATED STATES

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





GRAPH SEARCH

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

Augment TREE-SEARCH with a new data-structure:

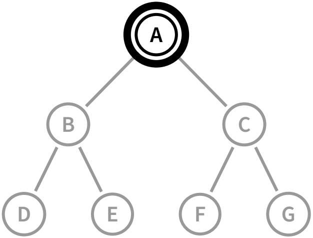
- the *explored set*, which remembers every expanded node
- newly expanded nodes already in explored set are discarded

BREADTH-FIRST SEARCH

Expand shallowest unexpanded node.

Implementation

frontier is a FIFO queue, i.e. new successors go at end

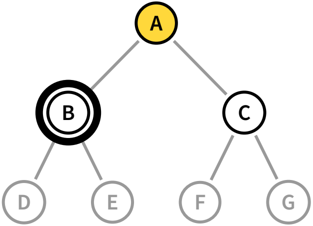


BREADTH-FIRST SEARCH

Expand shallowest unexpanded node.

Implementation

frontier is a FIFO queue, i.e. new successors go at end

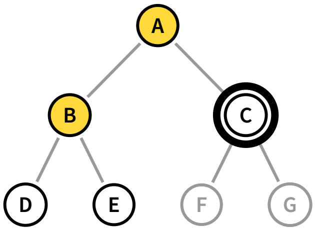


BREADTH-FIRST SEARCH

Expand shallowest unexpanded node.

Implementation

frontier is a FIFO queue, i.e. new successors go at end

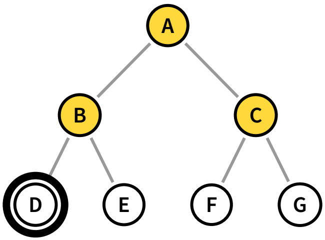


BREADTH-FIRST SEARCH

Expand shallowest unexpanded node.

Implementation

frontier is a FIFO queue, i.e. new successors go at end



BREADTH-FIRST SEARCH · ALGORITHM

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

BREADTH-FIRST SEARCH · PROPERTIES

- complete
- time
- space
- optimal

BREADTH-FIRST SEARCH · PROPERTIES

complete	Yes	(if b is finite)
time		
space		
optimal		

BREADTH-FIRST SEARCH · PROPERTIES

complete	Yes	(if b is finite)
time	$b + b^2 + b^3 + \cdots + b^d = O(b^d)$ (worst-case: regular b-ary tree of depth d)	
❓ space		
optimal		

BREADTH-FIRST SEARCH · PROPERTIES

complete	Yes (if b is finite)
time	$b + b^2 + b^3 + \cdots + b^d = O(b^d)$ (worst-case: regular b-ary tree of depth d)
space	$O(b^d)$ (keeps every node in memory)
optimal	

BREADTH-FIRST SEARCH · PROPERTIES

complete	Yes	(if b is finite)
time	$b + b^2 + b^3 + \cdots + b^d = O(b^d)$ (worst-case: regular b-ary tree of depth d)	
space	$O(b^d)$	(keeps every node in memory)
optimal	Yes	(if cost = 1 per step, then a solution is optimal if it is closest to the start node)

BREADTH-FIRST SEARCH · PROPERTIES

complete	Yes	(if b is finite)
time	$b + b^2 + b^3 + \cdots + b^d = O(b^d)$ (worst-case: regular b-ary tree of depth d)	
space	$O(b^d)$	(keeps every node in memory)
optimal	Yes	(if cost = 1 per step, then a solution is optimal if it is closest to the start node)

Space is the bigger problem (more than time).

UNIFORM-COST SEARCH

Expand least-cost unexpanded node.

Implementation

frontier is a queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs are all equal.

3.b

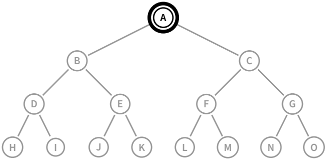
Depth-first search

DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

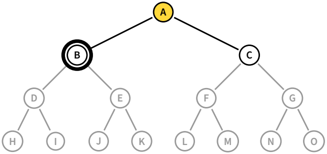


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

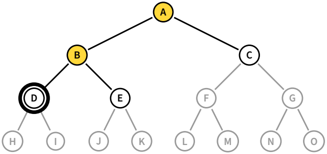


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

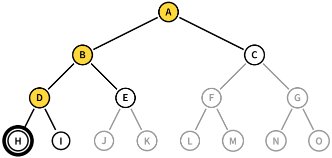


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

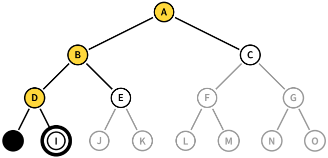


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

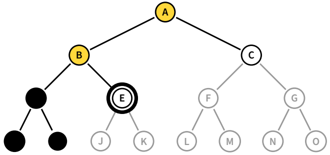


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

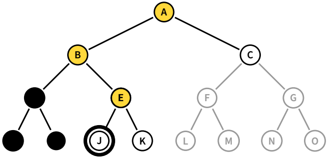


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

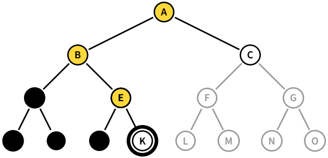


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

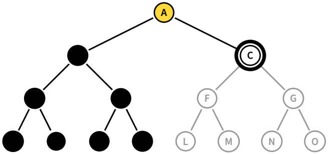


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

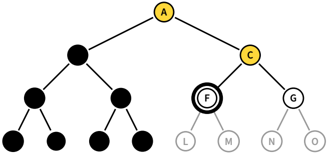


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

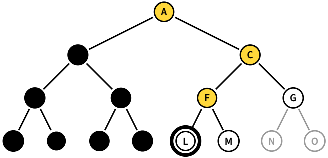


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front

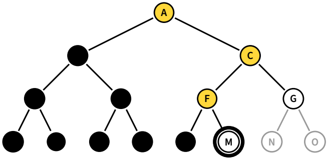


DEPTH-FIRST SEARCH

Expand deepest unexpanded node.

Implementation

frontier is a LIFO queue, i.e. put successors at front



DEPTH-FIRST SEARCH · PROPERTIES

- complete

time

space

optimal

complete	No	fails in infinite-depth spaces, spaces with loops
		Modify to avoid repeated states along path
		Complete in finite spaces
time		
space		
optimal		

DEPTH-FIRST SEARCH · PROPERTIES

complete	No fails in infinite-depth spaces, spaces with loops Modify to avoid repeated states along path 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	
optimal	

DEPTH-FIRST SEARCH · PROPERTIES

complete	No fails in infinite-depth spaces, spaces with loops Modify to avoid repeated states along path 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	$O(bm)$ linear space!
❓ optimal	

DEPTH-FIRST SEARCH · PROPERTIES

complete	No fails in infinite-depth spaces, spaces with loops Modify to avoid repeated states along path 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	$O(bm)$ linear space!
optimal	No

QUESTION TIME!

Compare breadth-first and depth-first search.

- When would breadth-first be preferable?
- When would depth-first be preferable?

ANSWER TIME!

Compare breadth-first and depth-first search.

- When would breadth-first be preferable?
  - when completeness is important.
  - when optimal solutions are important.

- When would depth-first be preferable?
  - when solutions are dense and
  - low-cost is important, especially space costs.

3.c

Improving depth-first search

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** a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  else if *limit* = 0 **then return** *cutoff*
  else
    cutoff_occurred? ← false
    for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
      child ← CHILD-NODE(*problem*, *node*, *action*)
      result ← RECURSIVE-DLS(*child*, *problem*, *limit* − 1)
      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, or failure
  for $depth$ = 0 **to** ∞ **do**
    $result \leftarrow$ DEPTH-LIMITED-SEARCH($problem, depth$)
    if $result \neq$ cutoff **then return** $result$

ITERATIVE DEEPENING SEARCH

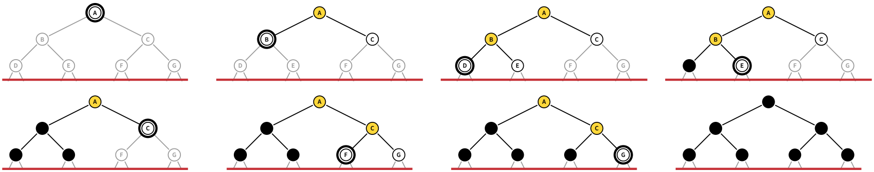
$l = 0$

ITERATIVE DEEPENING SEARCH

$l = 1$

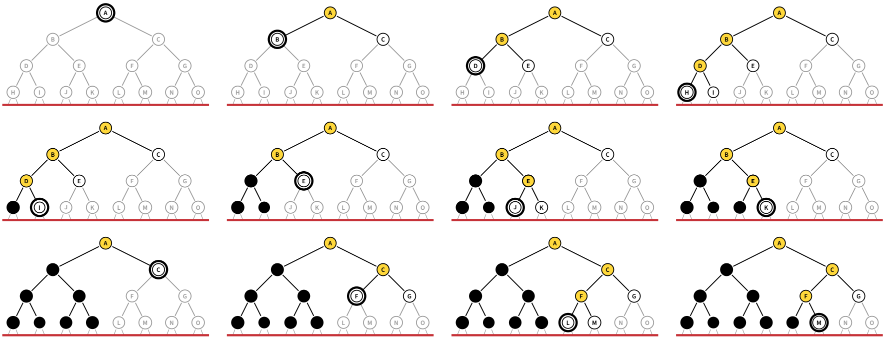
ITERATIVE DEEPENING SEARCH

$l = 2$



ITERATIVE DEEPENING SEARCH

$l = 3$



ITERATIVE DEEPENING SEARCH

Number of nodes generated in an iterative deepening search to depth d with branching factor b:

$$N_{IDS} = (d+1)b^0 + (d)b^1 + (d-1)b^2 + \cdots + (2)b^{d-1} + (1)b^d$$

Some cost associated with generating upper levels multiple times.

EXAMPLE

Comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N_{IDS} = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N_{BFS} = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

overhead $= (123,450 - 111,110)/111,110 = 11\%$

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

ITERATIVE DEEPENING SEARCH · PROPERTIES

- complete
- time
- space
- optimal

ITERATIVE DEEPENING SEARCH · PROPERTIES

complete	Yes
time	
space	
optimal	

ITERATIVE DEEPENING SEARCH · PROPERTIES

complete	Yes
time	$(d+1)b^0 + db + (d-1)b^2 + \cdots + b^d = O(b^d)$
space	
optimal	

ITERATIVE DEEPENING SEARCH · PROPERTIES

complete	Yes
time	$(d+1)b^0 + db + (d-1)b^2 + \cdots + b^d = O(b^d)$
space	$O(bd)$
optimal	

ITERATIVE DEEPENING SEARCH · PROPERTIES

complete	Yes
time	$(d+1)b^0 + db + (d-1)b^2 + \cdots + b^d = O(b^d)$
space	$O(bd)$
optimal	Yes if step cost = 1

SUMMARY

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*

TAKE-HOME MESSAGE

Uninformed search strategies use only the information available in the problem definition.

Graph search can be exponentially more efficient than tree search.

Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.