Uninformed Search

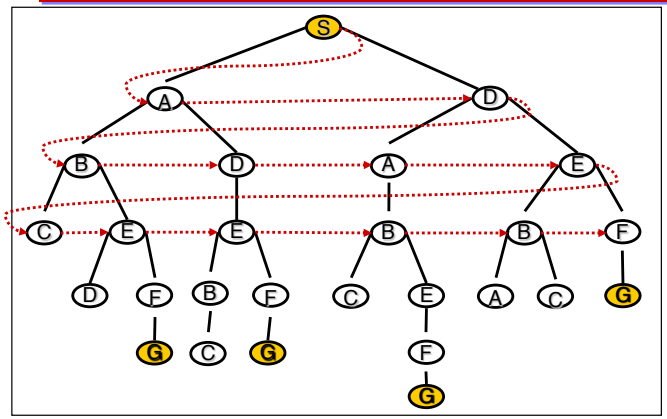
Uninformed Search Strategies

- 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 Example



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

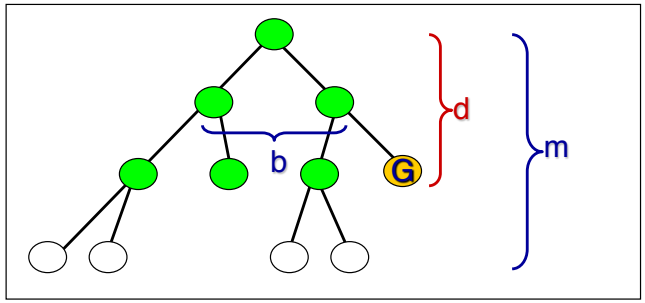
Properties of Breadth-first Search

- 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

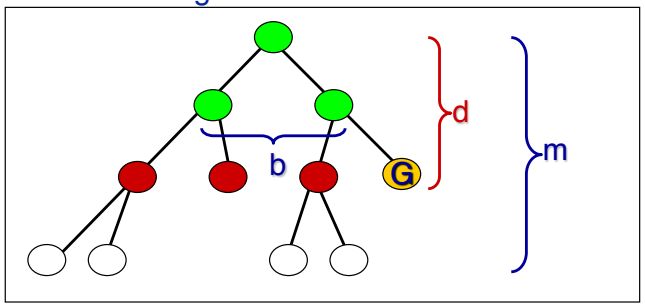
- If a goal node is found on depth d of the tree, all nodes up till that depth are created.



- 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 G nodes. (Thus: 4) .
- In General: b^d

Time and Memory Requirements for Breadth-first Search

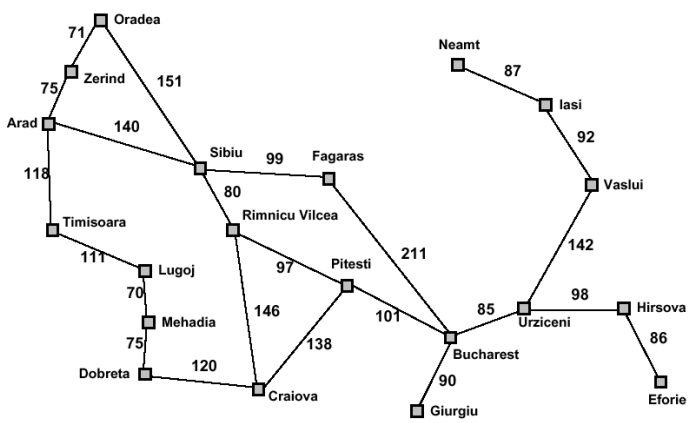
Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

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

Uniform-cost Search

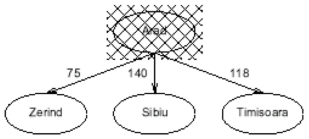
- Uniform-cost search expands the node with the lowest path cost, instead of expanding the shallowest node.
- A refinement of the breadth-first strategy:
  - Breadth-first = uniform-cost with path cost = node depth

Romania With Step Costs In Km

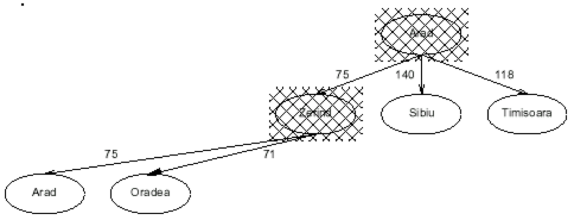


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

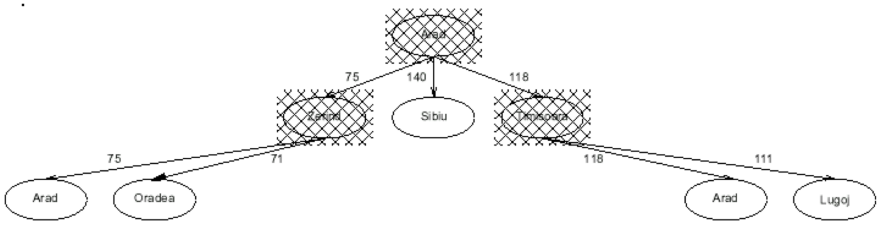
Uniform-cost Search



Uniform-cost Search



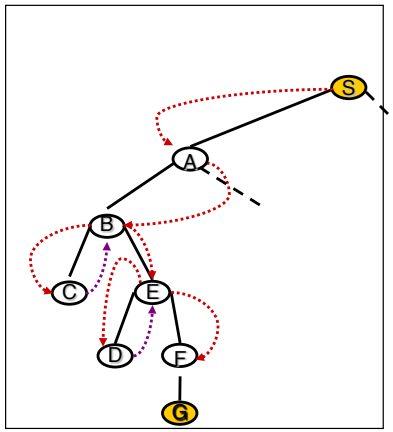
Uniform-cost Search



Properties Of Uniform-cost Search

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

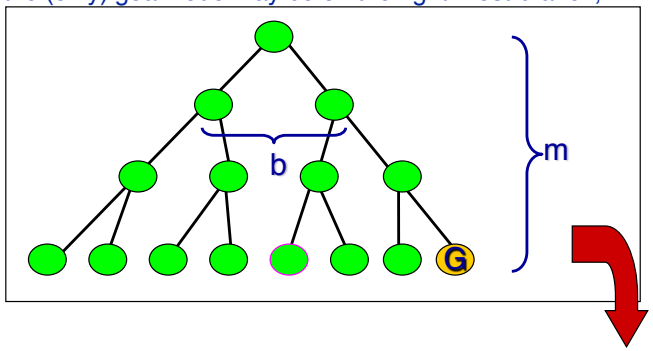
Implementation:
fringe = LIFO queue, i.e., put successors at front

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

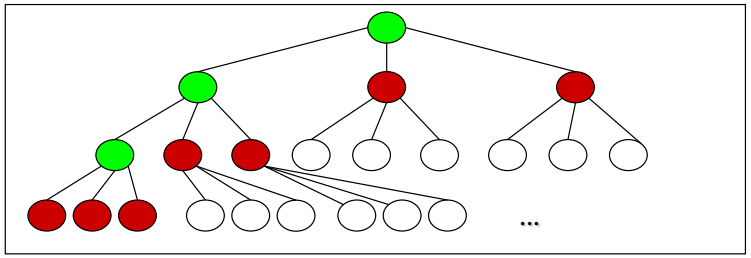
- In the worst case:
  - the (only) goal node may be on the right-most branch,



- 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

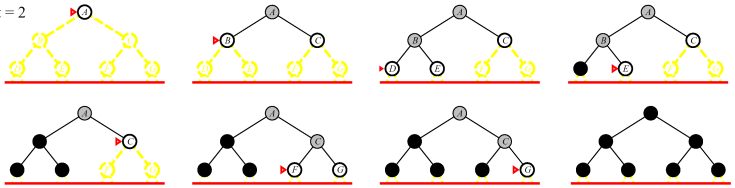
Limit = 0

Depth Limited Search Example

Limit = 1

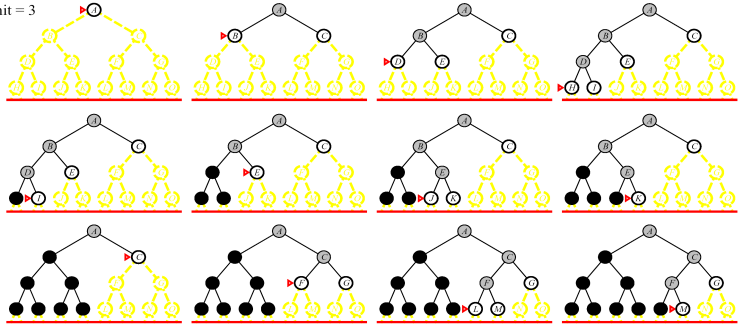
Depth Limited Search Example

Limit = 2



Depth Limited Search Example

Limit = 3



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 modified 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

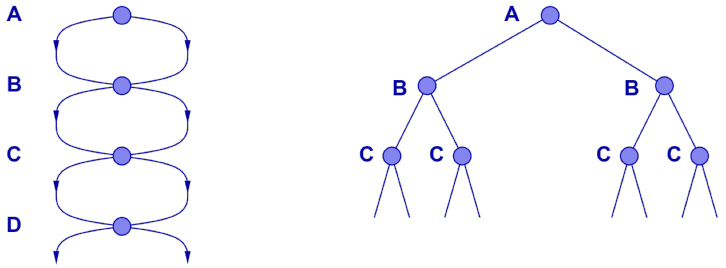
- Both search forward from initial state, and backwards from goal.
- Stop when the two searches meet in the middle.
- Problem: how do we search backwards from goal??
  - Predecessor of node n = all nodes that have n as successor
  - This may not always be easy to compute!
  - If several goal states, 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 $O(b^d)$!!!!