Intelligent Systems

Ինտելեկտուալ տեղեկ. համակարգեր

NUACA/ՃՇՀԱՀ

2017

LECTURE 3

Big O notation (repetition)

- Big O notation characterises functions according to their growth rates.
- When analyzing an algorithm one might find that the time (or the number of steps) it takes to complete a problem of size *n* is given by *T*(*n*)

Let $\mathrm{T}(n)$ be a function on n = 1, 2, 3,…

Question: When is $\mathrm{T}(n) = \mathrm{O}(f(n))$?

Answer: Eventually (for all sufficiently large n) T(n) is bounded above by a constant multiple of $f(n)$.

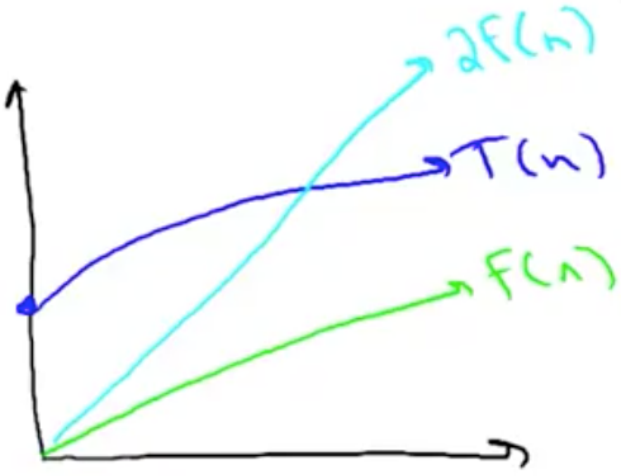
Big O notation
Formal Definition

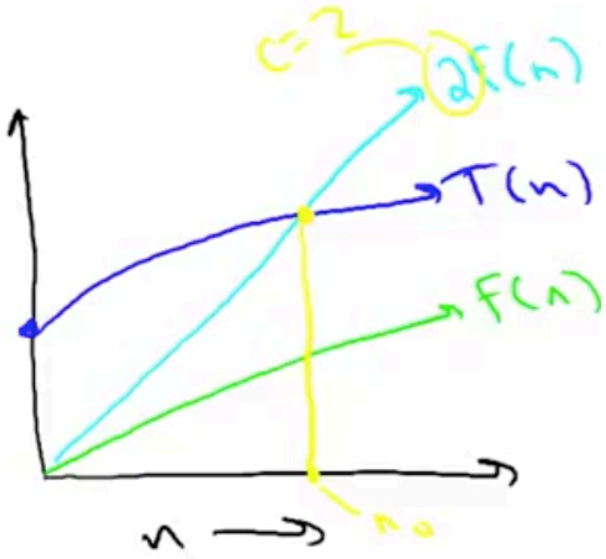
- Formal Definition:

$\mathrm{T}(n) = \mathrm{O}(f(n))$ if and only if there exists constants c, n_0 > o such that

$\mathrm{T}(n) \leq c \times f(n)$ for all $n \geq n_0$

Warning: c, n_0 are independent of n.





Big O notation
Example: One Loop

Given A (array of length n) and t (an integer)

Problem: Does the array A contain the integer t?

```
for i = 1 to n
   if A[i] == t return TRUE
return FALSE
```

What is the running time?

1) O(1)

2) O(logn)

3) O(n)

4) O(n^2)

Big O notation
Example: Two Loops

Given A, B (arrays of length n) and t (an integer)

Problem: Do the arrays A or B contain the integer t?

```
for i = 1 to n
   if A[i] == t return TRUE
for i = 1 to n
   if B[i] == t return TRUE
return FALSE
```

What is the running time?

1) O(1)

2) O(logn)

3) O(n)

4) O(n^2)

Big O notation
Example: Two Loops

Given A, B (arrays of length n) Problem: Do arrays A and B have a number in common?

```
for j = 1 to n
   for j = 1 to n
      if A[i] == B[j]
         return TRUE
return FALSE
```

What is the running time?

1) O(1)

2) O(logn)

3) O(n)

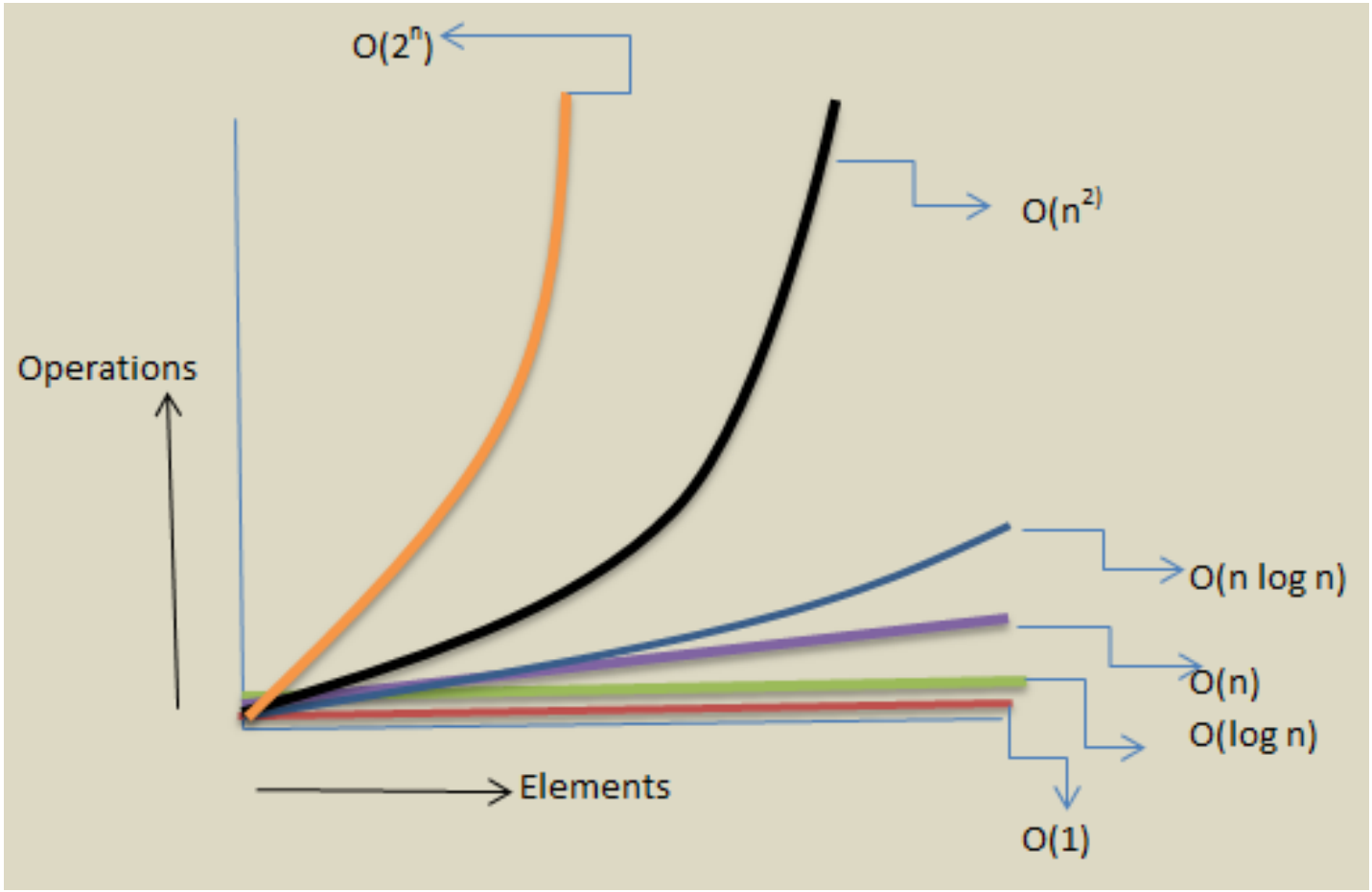
4) O(n^2)

Common complexities

Here is a list of classes of functions that are commonly encountered when analyzing algorithms.

notation	name
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Common complexities



Other notations

Notation	Analogy
$f(n) = O(g(n))$	$\leq$
$f(n) = o(g(n))$	$<$
$f(n) = \Omega(g(n))$	$\geq$
$f(n) = \omega(g(n))$	$>$
$f(n) = \theta(g(n))$	$=$

Uninformed Search Techniques

AIMA: CHAPTER 3.4

Problems

A **problem** can be formulated by specifying:

–The **state space** *("the set of all possible states we might get into") - վիճակների բազմություն*

–The **initial state** *("where we are") - սկզբնական վիճակ*

–A test for the **goal** *("are we where we want to be?") - արդյոք հասել ենք նպատակին*

- *Formally, we define a Boolean function* Goal *on the set of states*

–Available **actions** at any state *("what can we do?") - հասանելի գործողությունները բազմություն (ամեն վիճակում)*

- *Formally, we define a set-valued function* Actions *on the set of states*

–A **transition model** *("what will that achieve?") - անցում նոր վիճակի*

- *Formally, we define a state-valued function* Result *on the set of (state,action) pairs*

–A **cost measure** for sequences of actions *("is it worth it?") - գործողության արժեք*

- *Formally, we define a real-valued function* Cost *on the set of actions, or sequences*

Problems

From a problem specification we can derive:

–A **solution** *("a sequence of actions that take us to a goal state") - լուծում*

- *Formally, this is a **path** (ճանապարհ) in the state space, a sequence* $s_0, a_1, s_1, a_2, s_2, \ldots, a_n, s_n$ *where:*
- $s_0, s_1, s_2, \ldots, s_n$ *are **states**,* s_0 *is the **initial state**,* s_n *is a **goal state*** ($\text{Goal}(s_n)$ = TRUE)
- $a_1, a_2, \ldots, a_n$ *are* **actions**,
- $a_i \in \text{Actions}(s_{i-1})$ and $s_i = \text{Result}(s_{i-1}; a_i)$ for each $1 \leq i \leq n$

– An **optimal solution** *("a solution with minimal cost") – օպտիմալ լուծում*

Search
Որոնում

The basic idea is to explore the state space of a problem by generating the states that are reachable from the current state (known as expanding the state) and systematically examining them in some order

Հիմնական գաղափարն այն է ուսումնասիրել խնդրի վիճակների բազմությունը ստեղծելով վիճակներ, որոնք հասանելի են ներկա վիճակից (վիճակի ընդարձակում) եւ կանոնավոր կերպով ուսումնասիրել դրանք որոշակի հերթականությամբ:

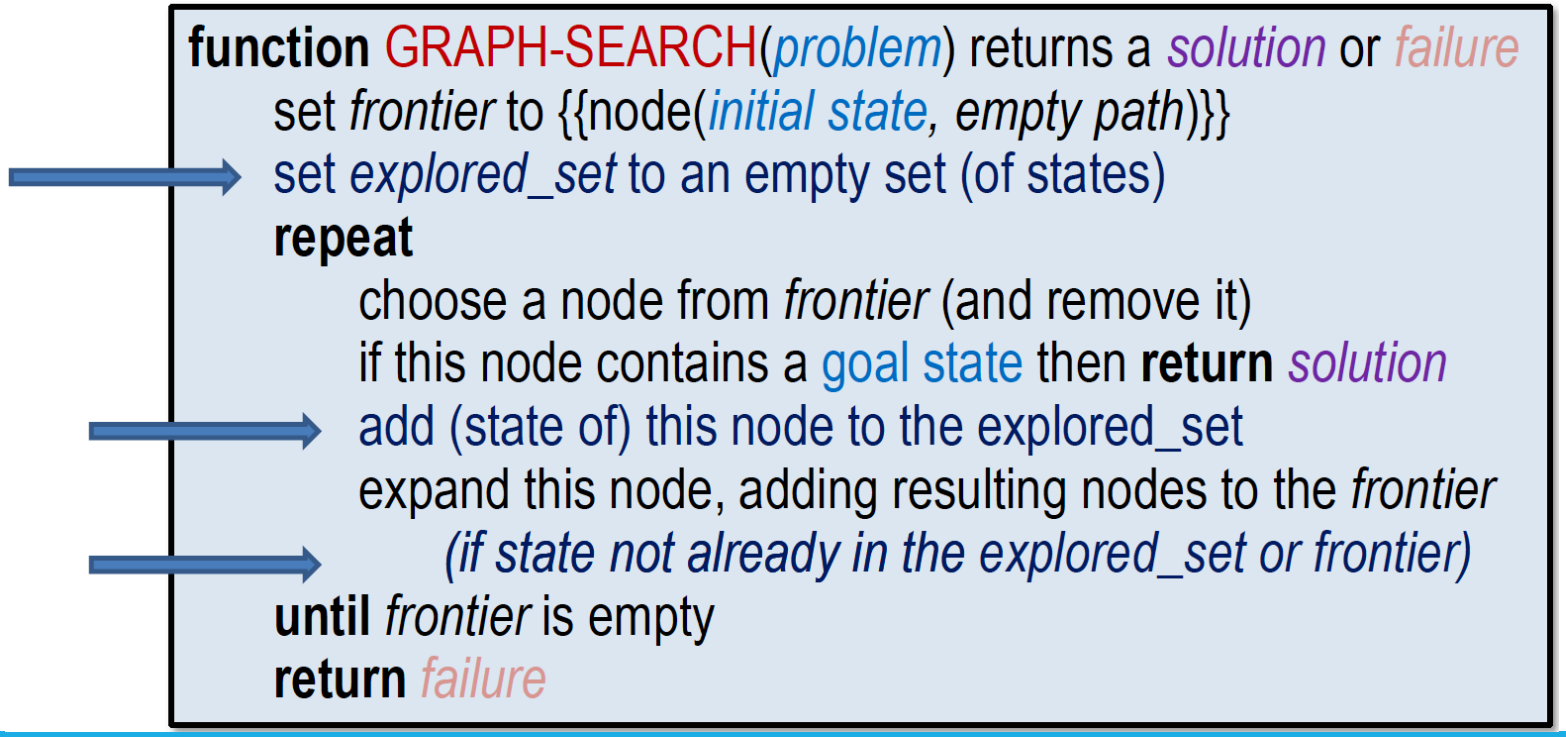
Tree-search

The basic idea is to explore the state space of a problem by generating the states that are reachable from the current state (known as expanding the state) and systematically examining them in some order

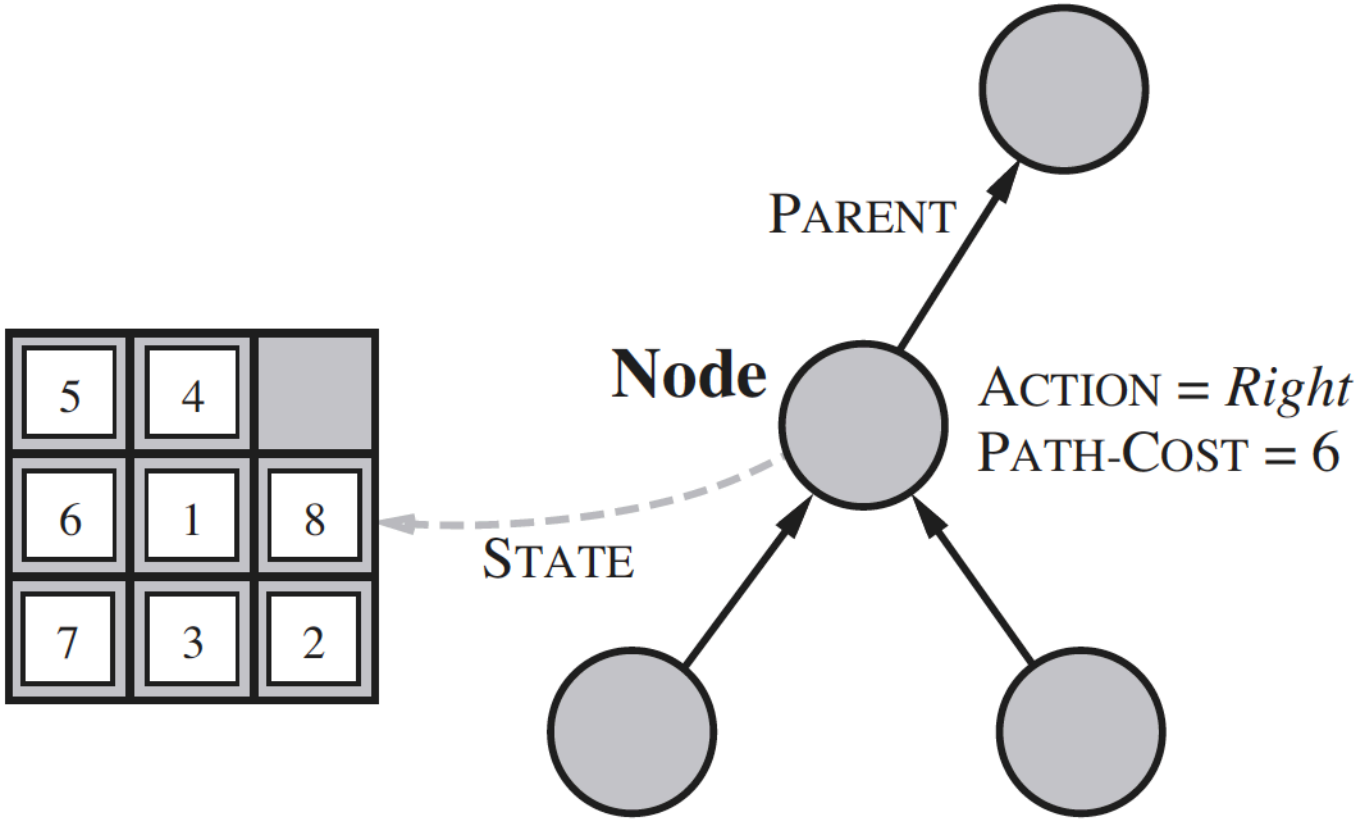
```
function TREE-SEARCH(problem) returns a solution or failure
    set frontier to {{node(initial state, empty path)}}

    repeat
        choose a node from frontier (and remove it)
        if this node contains a goal state then return solution

        expand this node, adding resulting nodes to the frontier

    until frontier is empty
    return failure
```

Graph-search

The basic idea is to explore the state space of a problem by generating **new** states that are reachable from the current state (known as expanding the state) and systematically examining them in some order



Node/հաևգnւյg



Issues in Graph Search

A graph search may explore exponentially fewer nodes, but:

- *space requirements* grow with state space
- it may be costly to determine equality of states
- note we only compare *states*, not other information in the nodes such as cost or path
- we may want “approximate” equality
- can use a hash table to implement *explored_set*

Measuring problem-solving performance

1. **Completeness**: Is the algorithm guaranteed to find a solution when there is one? - Լրիվություն
2. **Optimality**: Does the strategy find the optimal solution? - Օպտիմալություն
3. **Time complexity**: How long does it take to find a solution? - Ժամանակի բարդություն
4. **Space complexity**: How much memory is needed to perform the search? – Հիշողության բարդություն

Uninformed Search Techniques

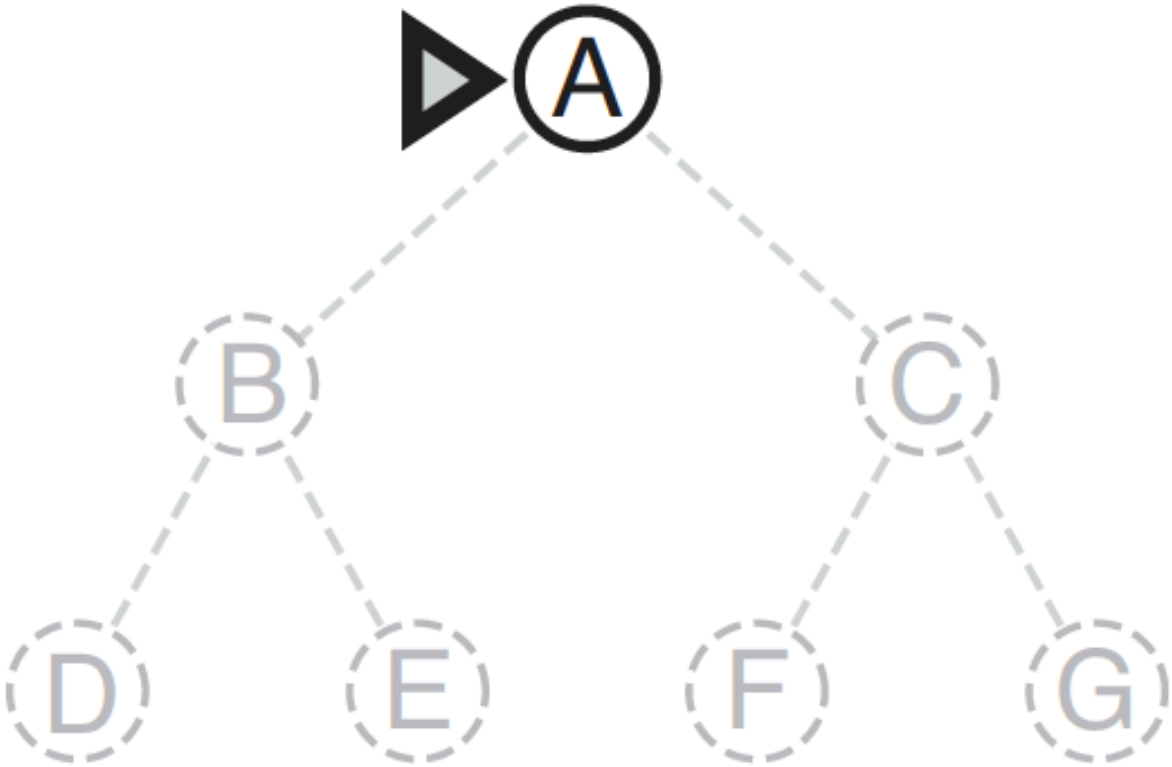
- **Uninformed search** (also called blind search) indicates that the strategies have no additional information about states beyond that provided in the problem definition
- All they can do is generate successors and distinguish a goal state from a non-goal state.
- All search strategies are distinguished by the order in which order the nodes are expanded.
- Strategies that know whether one non-goal state is "*more promising*" than another are called **informed search** strategies

Breadth-first search (BFS) Փնտրում դեպի լայնություն

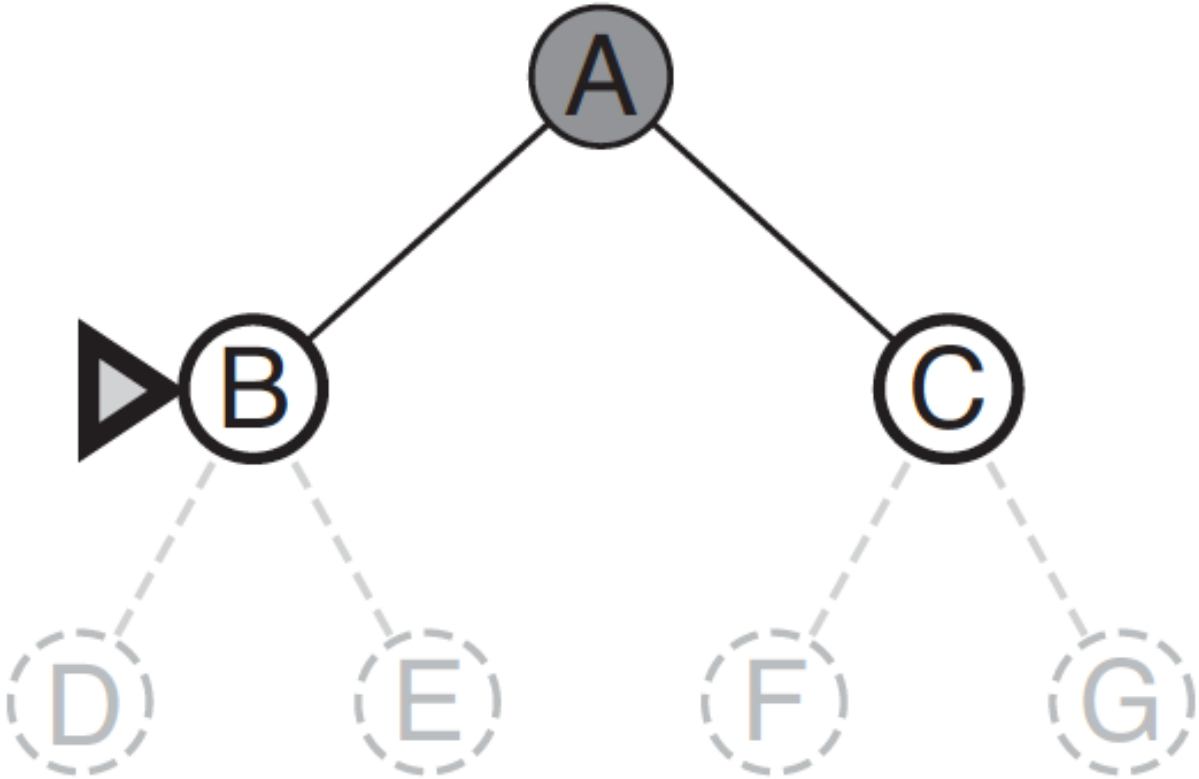
- the root node is expanded first
- all the successors of the root node are expanded next
- then their successors, and so on.

In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

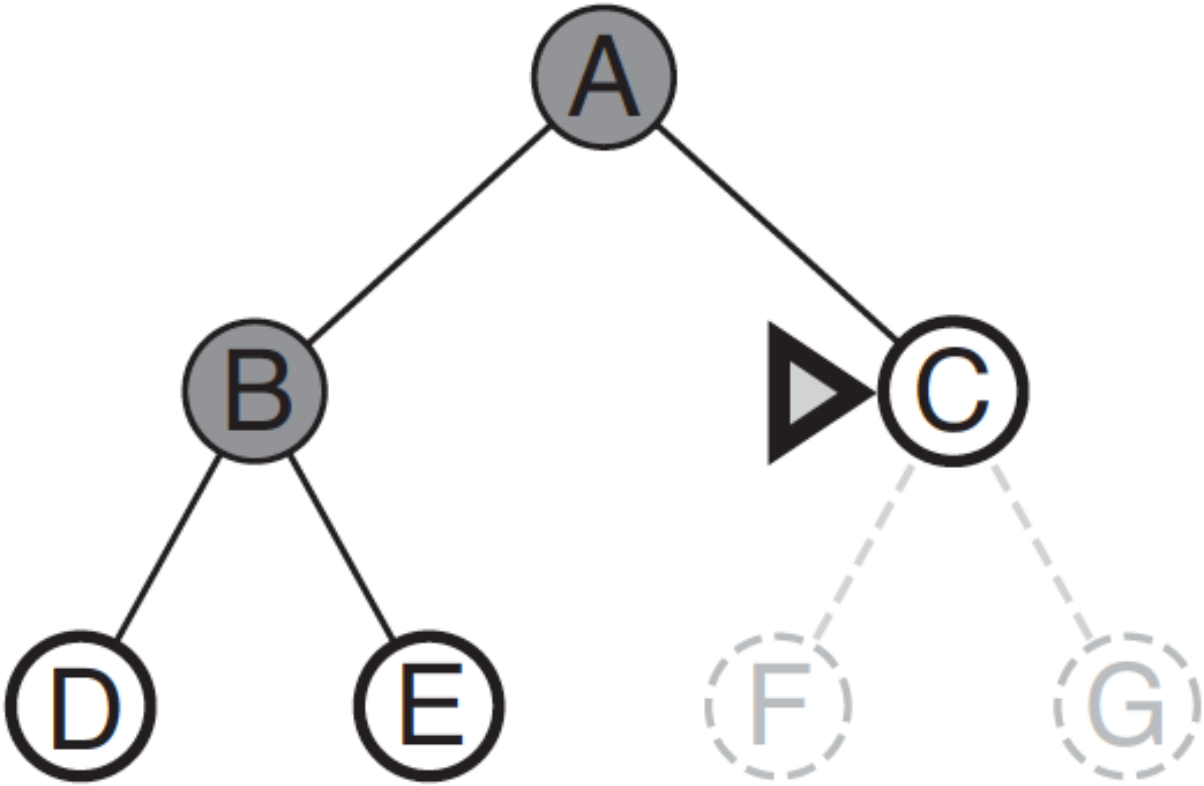
BFS on a simple binary tree



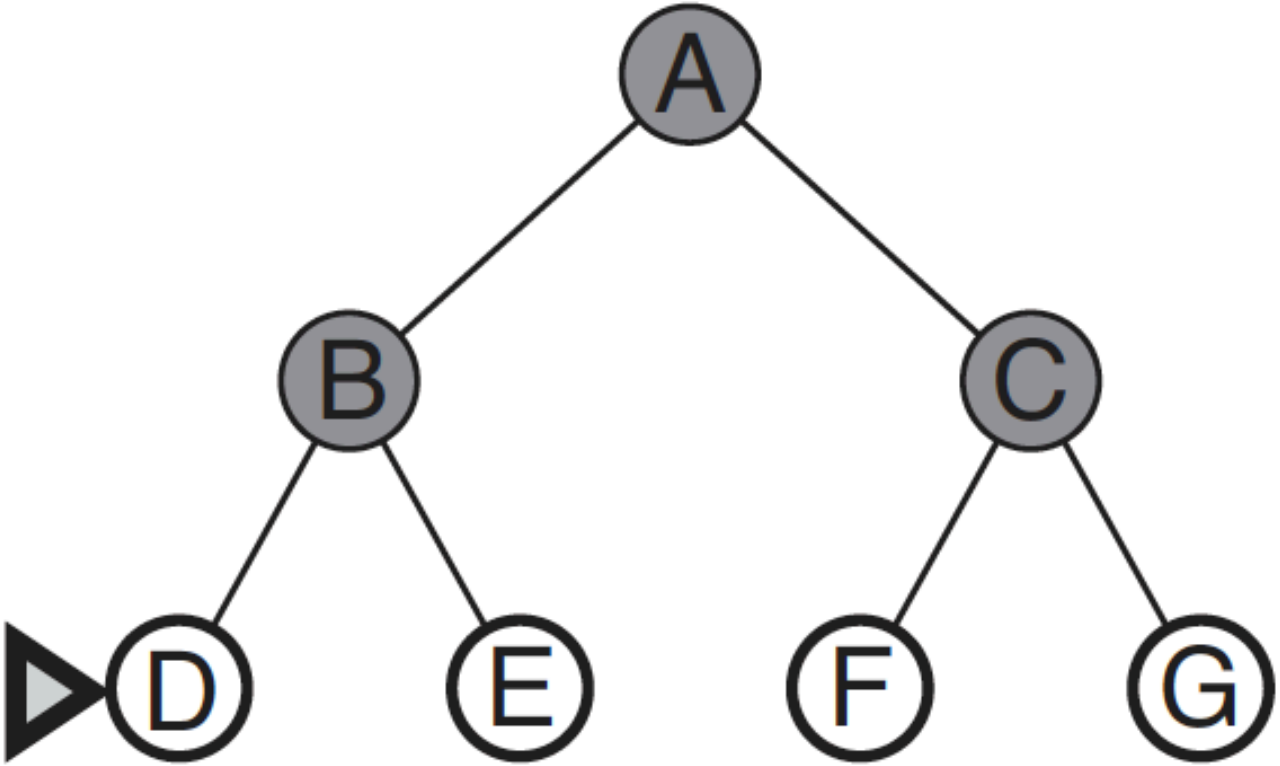
BFS on a simple binary tree



BFS on a simple binary tree



BFS on a simple binary tree

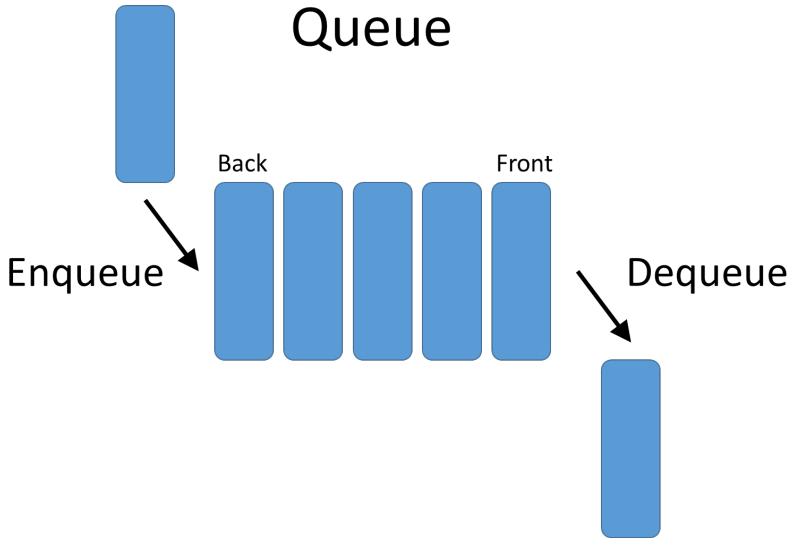


Breadth-first search Փնտրում դեպի լայնություն

Breadth-first search is an instance of the general graph-search algorithm in which the **shallowest** unexpanded node is chosen for expansion.

Question: which data structure should be used as the frontier?

Answer: FIFO (First In First Out) Queue



BFS on a graph

```
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)
```

Completeness
Լրիվություն

Question: Does BFS always reach a goal state?

Answer: Yes! (provided the branching factor b is finite).

if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes

Optimality
Օպտիմալություն

Question: Does BFS always find the optimal solution?

Answer: Yes and No.

- BFS is optimal if the path cost is a nondecreasing function of the depth of the node.
- The most common such scenario is that all actions have the same cost.

Time complexity
ժամանակային բարդություն

- 0 level – 1
- 1 level – b
- 2 level – b^2
- 3 level – b^3

…

if the shallowest solution is at depth *d,* we would have generated

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

If goal test is performed when the node is selected for expansion, then depth d would have been generated too, resulting in $O(b^{d+1})$

Space complexity

- For breadth-first graph search in particular, every node generated remains in memory.
- There will be $O(b^{d-1})$ nodes in the explored set and $O(b^{d})$ nodes in the frontier
- The space complexity thus is $O(b^{d})$
- Switching to a tree search would not save much space, and in a state space with many redundant paths, switching could cost a great deal of time.

BFS Complexity

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

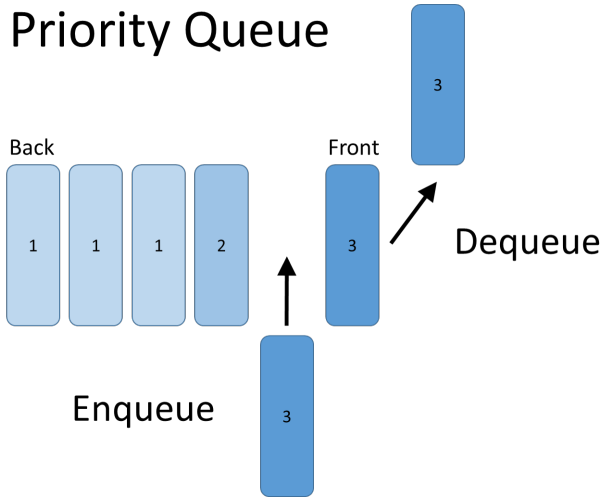
Uniform-cost search
Միօրինակ արժեքով որոնում

When all step costs are equal, BFS is optimal because it always expands the shallowest unexpanded node.

Instead of expanding the shallowest node, UCS expands the node n with the lowest path cost g(n)

Question: which data structure should be used as the frontier?

Answer: Priority Queue



Uniform-cost search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    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
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Notes for graph search

Goal test is applied to a node when it is *selected for expansion* rather than when it is first generated.

1) The reason is that the first goal node that is generated may be on a suboptimal path.

2) Test is added in case a better path is found to a node currently on the frontier.

Completeness

Question: Does BFS always reach a goal state?

Answer: Yes! if step costs $\geq \epsilon > 0$

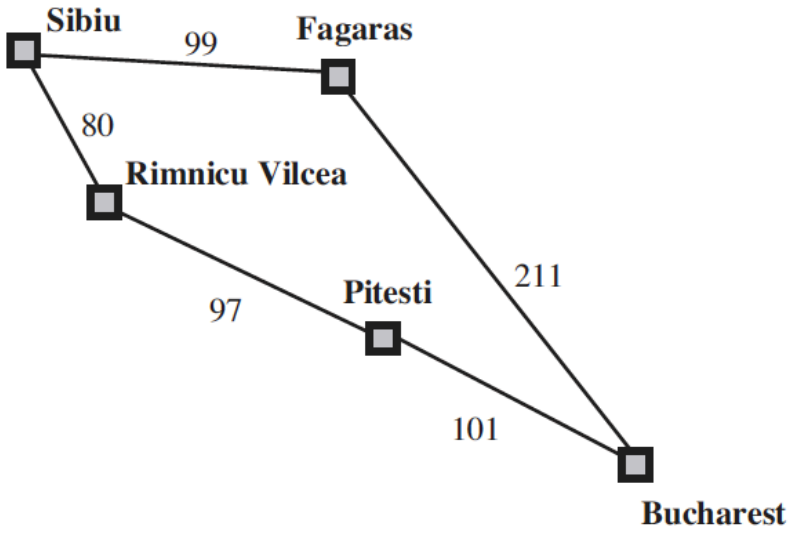
Optimality

Question: Does UCS always find the optimal solution?

Answer: Yes!

1) whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found.

2) step costs are nonnegative, paths never get shorter as nodes are added.



Time/space complexity

- Let C^* be the cost of the optimal solution and every action costs at least ϵ.
- Then the algorithm's worst-case time and space complexity is $O(b^{1+\lceil \frac{C^*}{\epsilon} \rceil})$ (can be much greater than $O(b^d)$)
- When all costs are the same, then $O(b^{1+\lceil \frac{C^*}{\epsilon} \rceil}) = O(b^d)$

UCS vs Dijkstra's Algorithm

Algorithm 1: Dijkstra's algorithm

Input: Graph $G = (V, E)$

1 $(\forall x \neq$
2 $dist[s]$
3 $S = \emptyset$
4 $Q = V$
5 **while**
6 $u =$
7 $S =$
8 **for**
9
10

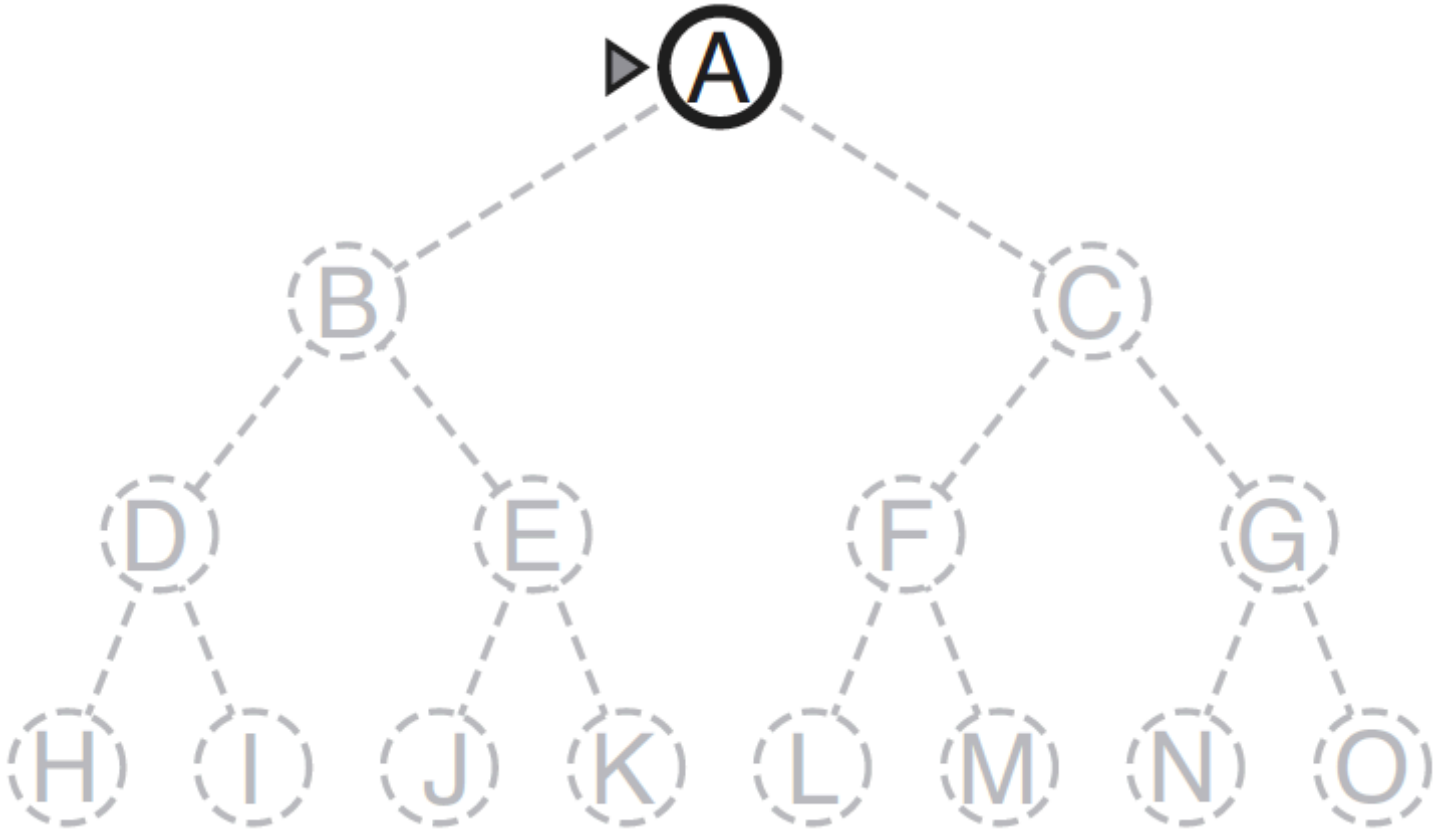
"I claim that UCS is superior to DA in almost all aspects. It is easier to understand and implement. Its time and memory needs are also smaller. The reason that DA is taught in universities and classes around the world is probably only historical. I encourage people to stop using and teaching DA, and focus on UCS only."

Position paper: Dijkstra's Algorithm Vs. Uniform Cost Search or A Case Against Dijkstra's Algorithm
Ariel Felner

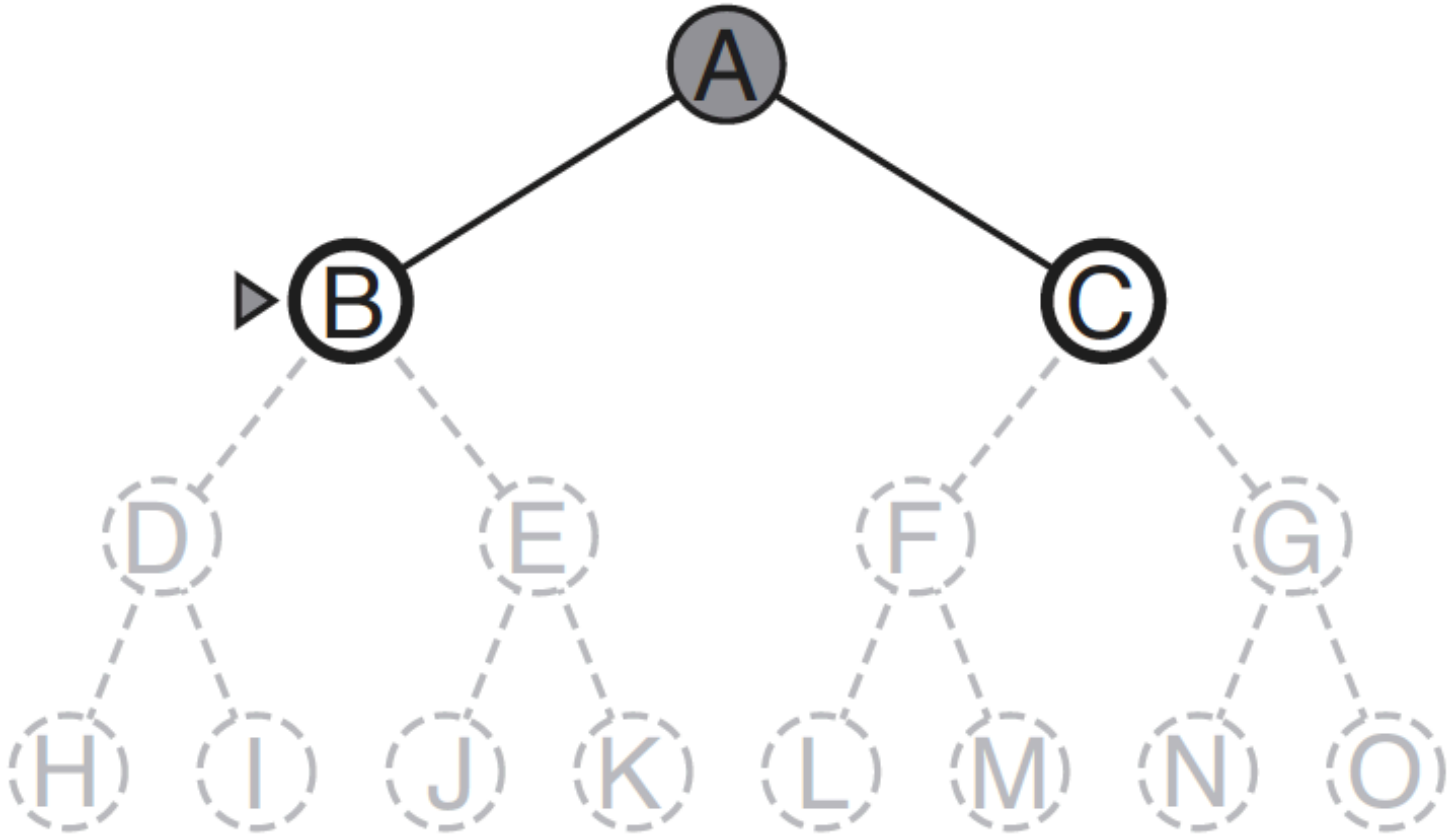
Depth-first search (DFS) Փնտրում դեպի խորություն

- Depth-first search always expands the **deepest** node in the current frontier

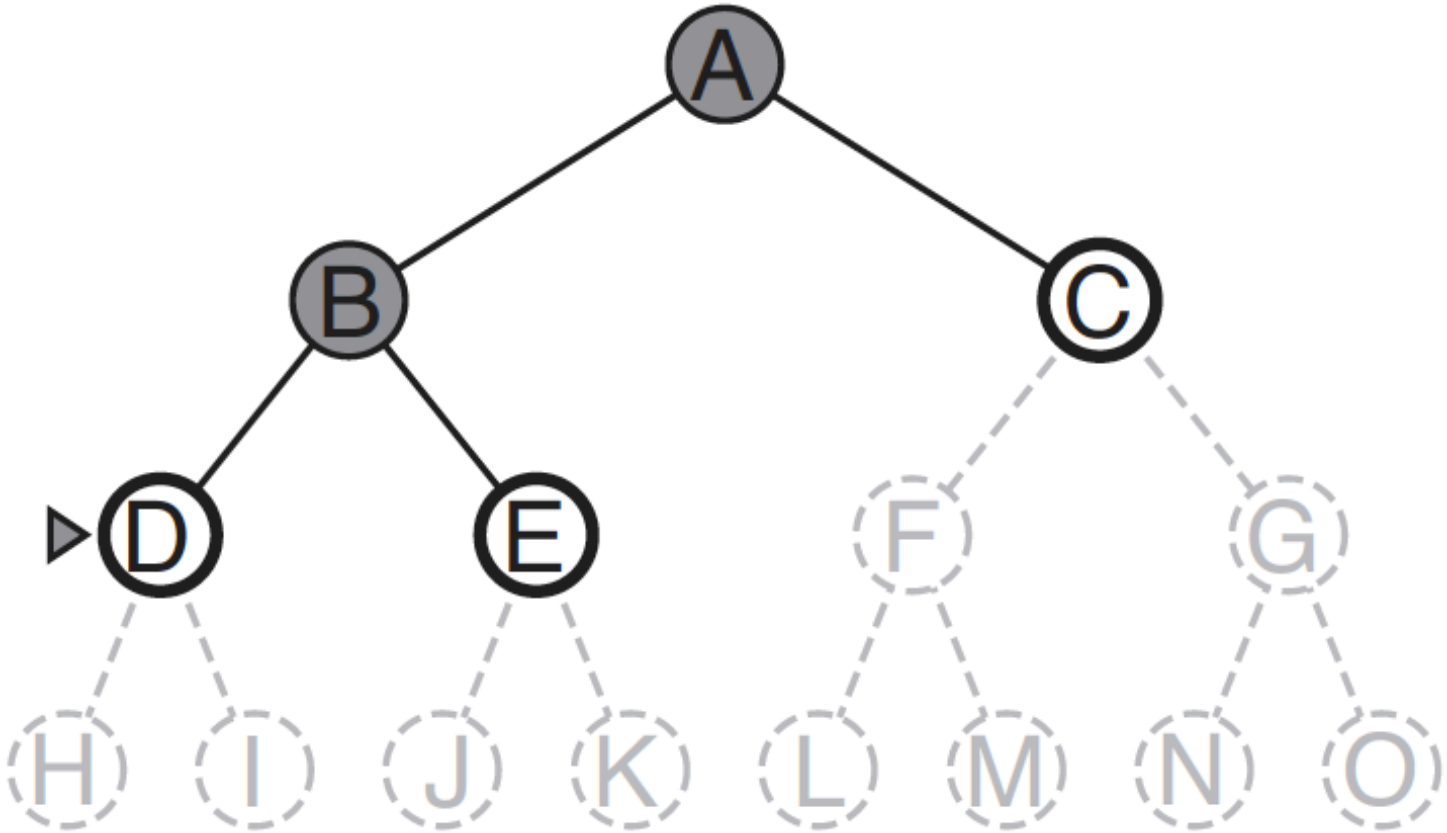
DFS on a binary tree.



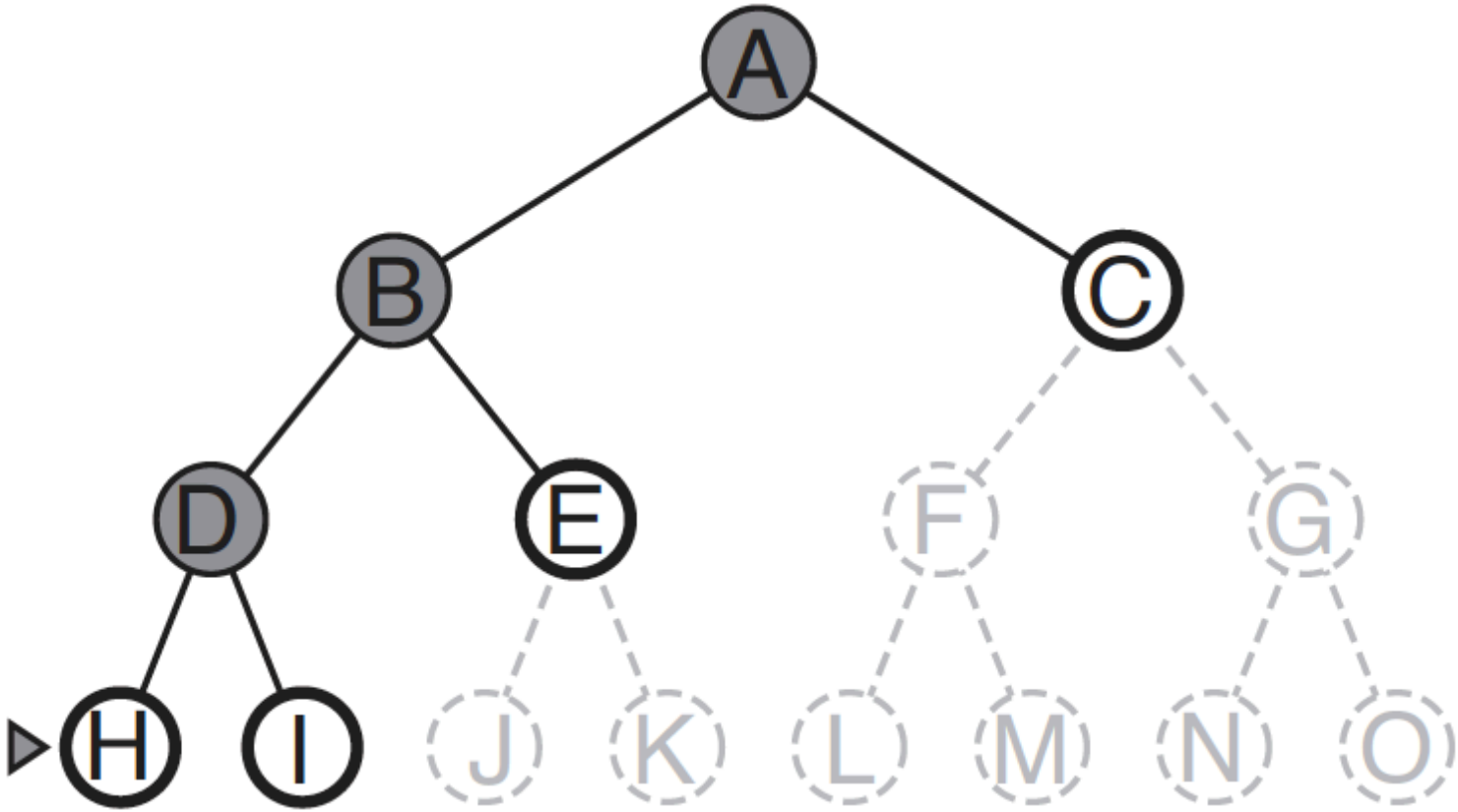
DFS on a binary tree.



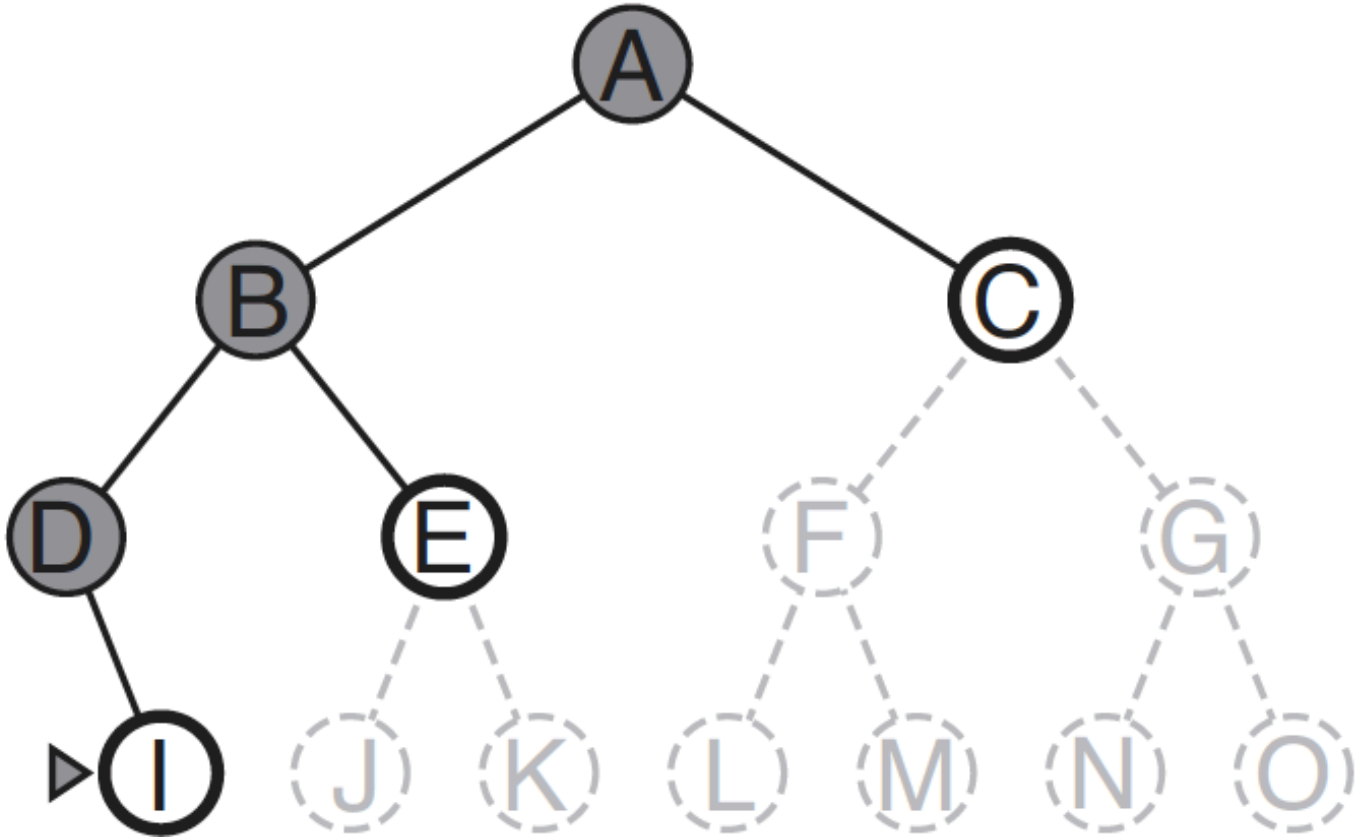
DFS on a binary tree.



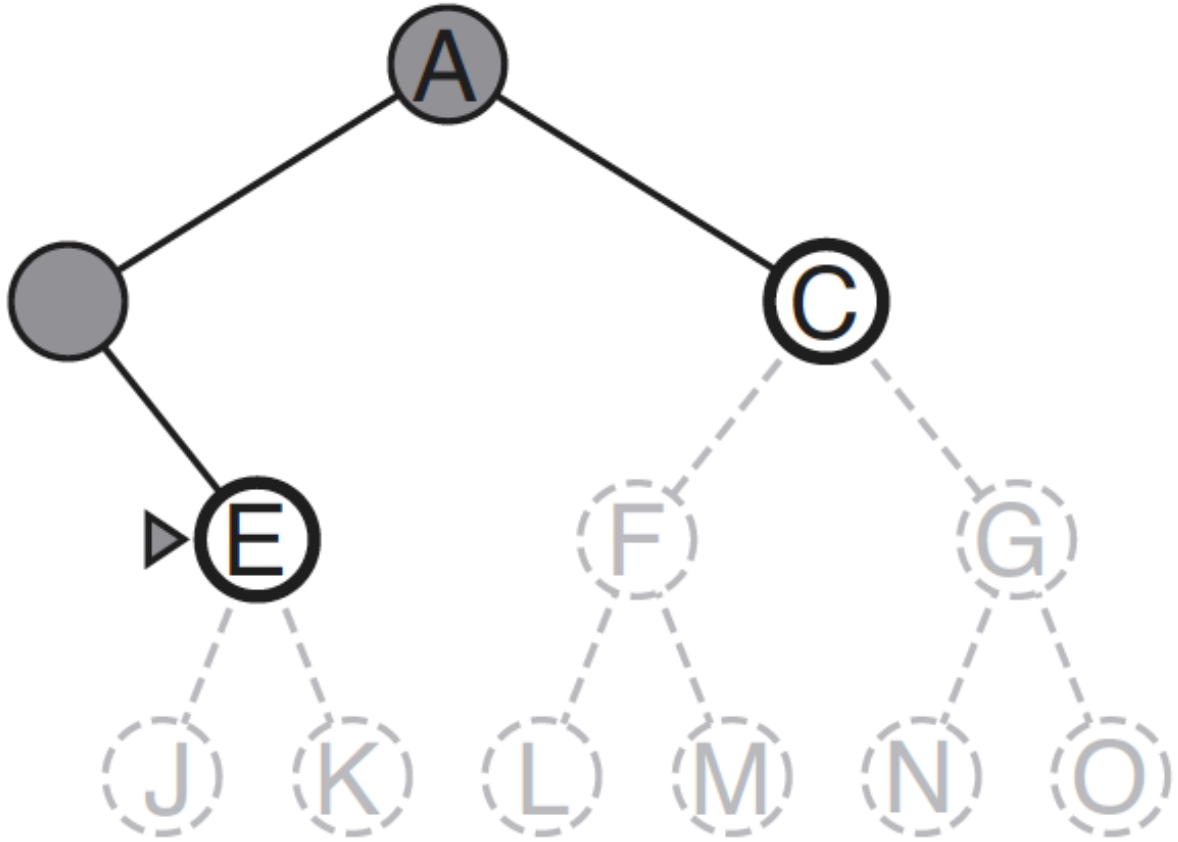
DFS on a binary tree.



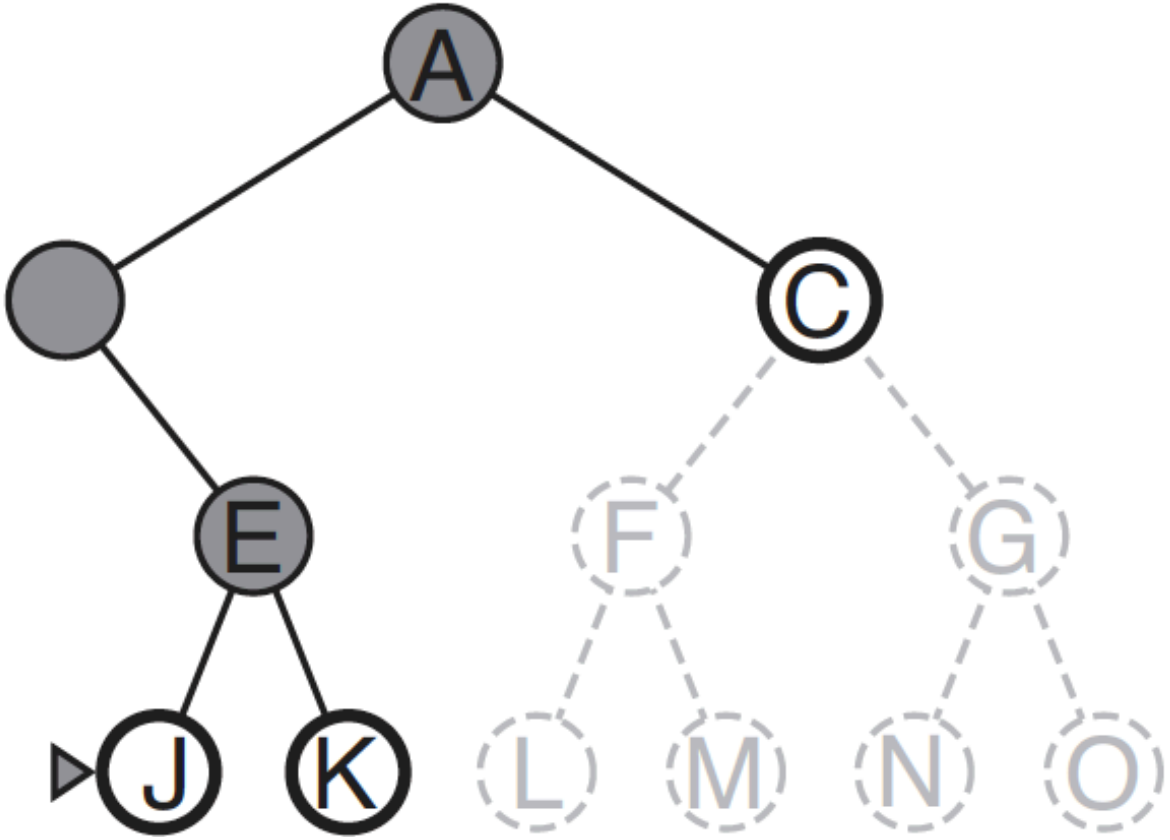
DFS on a binary tree.



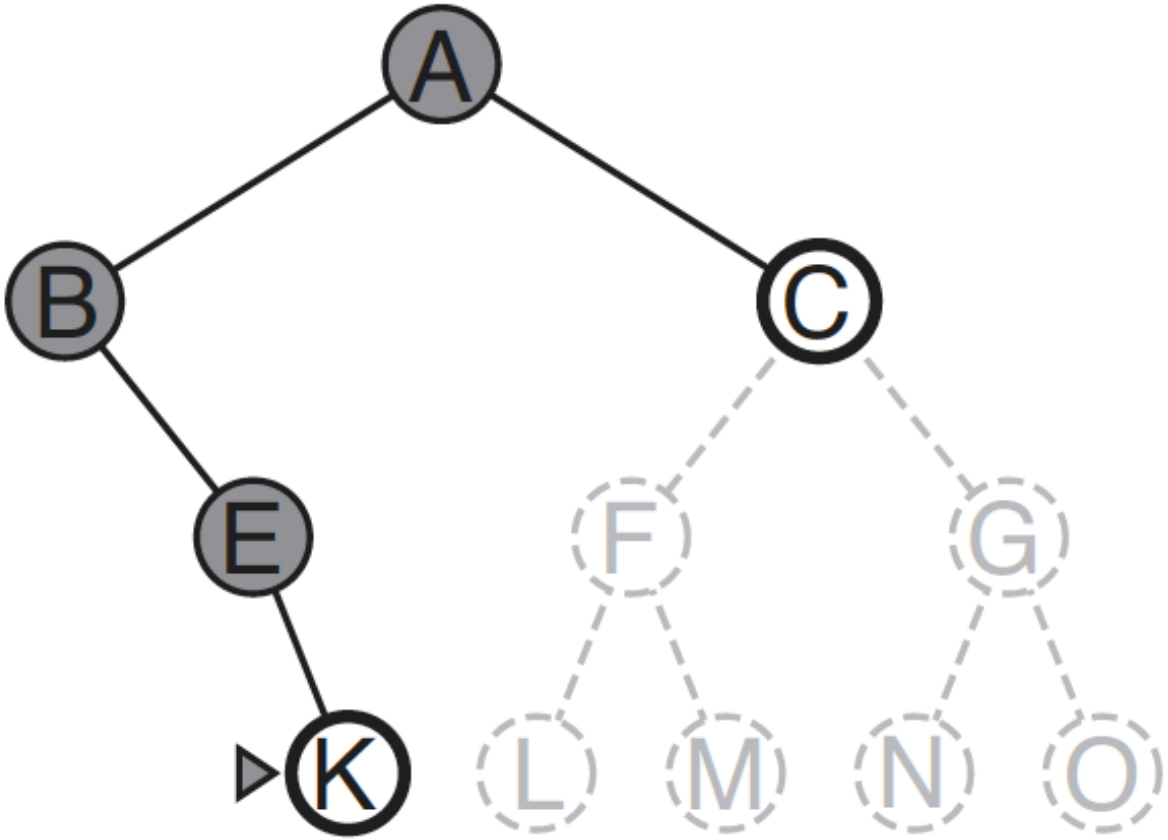
DFS on a binary tree.



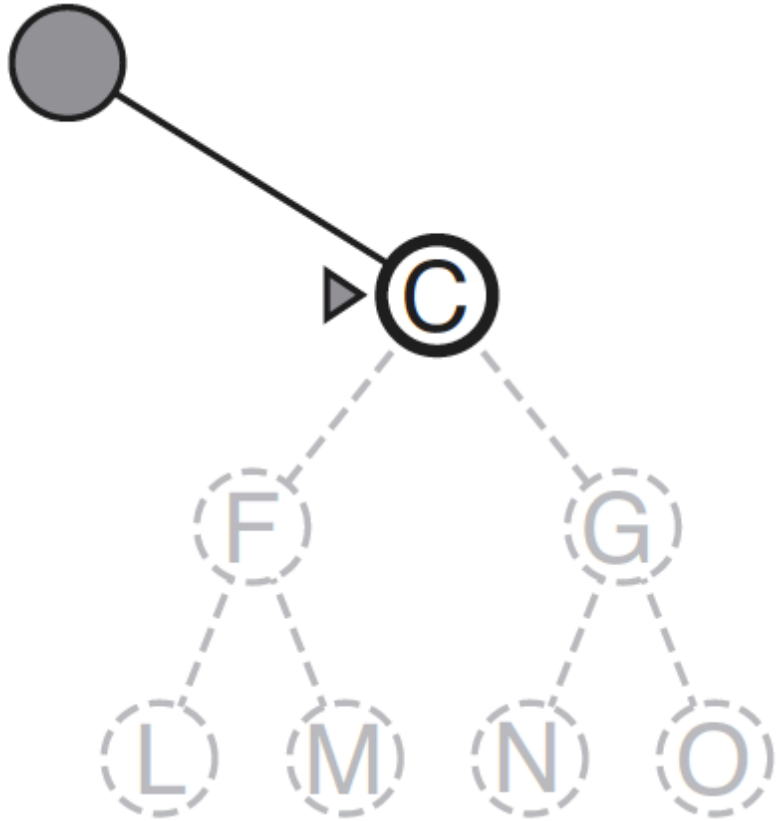
DFS on a binary tree.



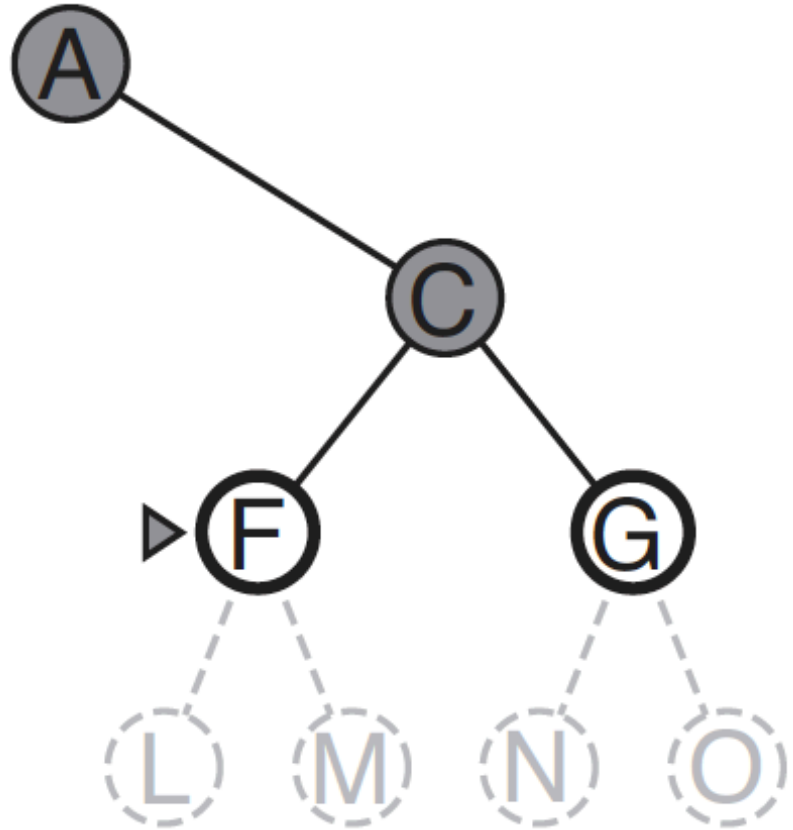
DFS on a binary tree.



DFS on a binary tree.



DFS on a binary tree.

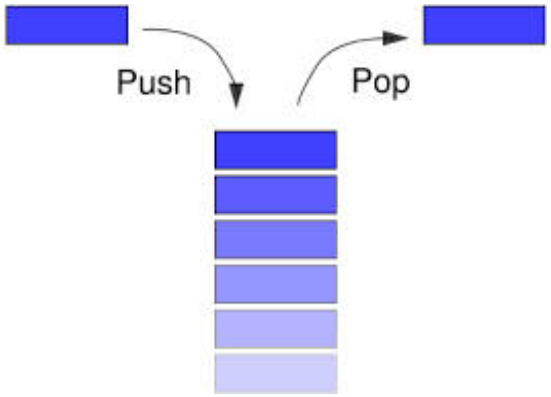


Depth-first search (DFS) Փնտրում դեպի խորությունն

Depth-first search always expands the **deepest** node in the current frontier

Question: which data structure should be used as the frontier?

Answer: LIFO (Last In First Out) Queue (also called Stack)



It is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

Completeness

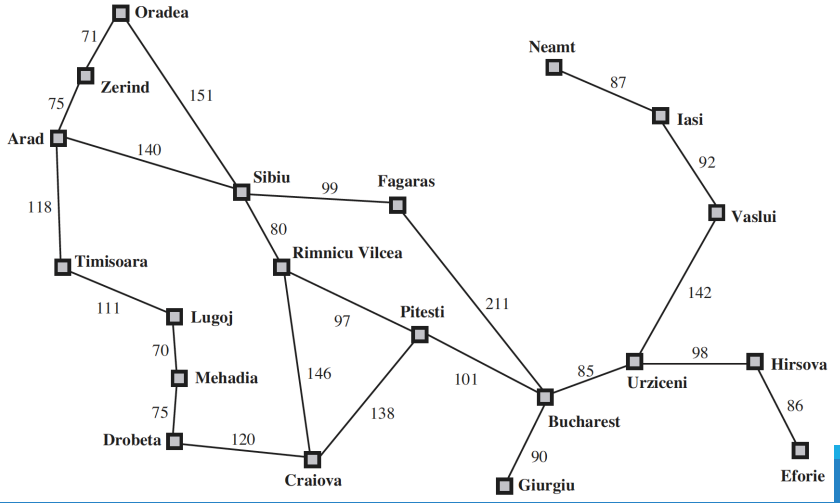
GRAPH SEARCH

The graph-search version, which avoids repeated states and redundant paths, **is complete** in finite state spaces because it will eventually expand every node.

In infinite state spaces, both versions fail if an infinite non-goal path is encountered.

TREE SEARCH

The tree-search version, on the other hand, is not complete—for example, in the algorithm will follow the Arad–Sibiu–Arad–Sibiu loop forever.



Optimality

Question: Is DFS optimal? (Tree search and graph search)

Answer: No, both versions are nonoptimal.

Time Complexity

GRAPH SEARCH

bounded by the size of the state space

(which may be infinite, of course).

TREE SEARCH

May generate $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node

(can be much greater than the size of the state space)

m can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.

Space Complexity

GRAPH SEARCH

No advantage over BFS

TREE SEARCH

Stores only a single path from the root to a leaf node

For branching factor b and maximum depth m, DFS requires storage of only $O(b \times m)$ nodes.

Iterative deepening search Իտերատիվ խորացող որոնում

Iterative deepening search finds the best depth limit.

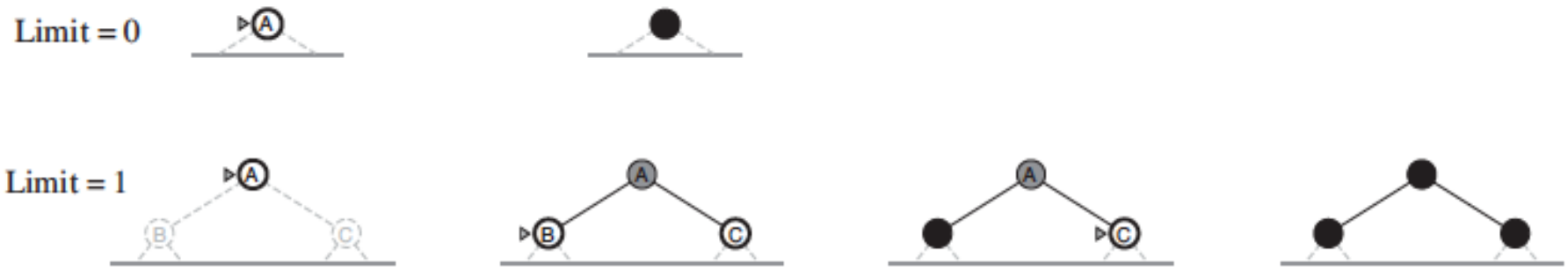
It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.

Combines the benefits of DFS and BFS

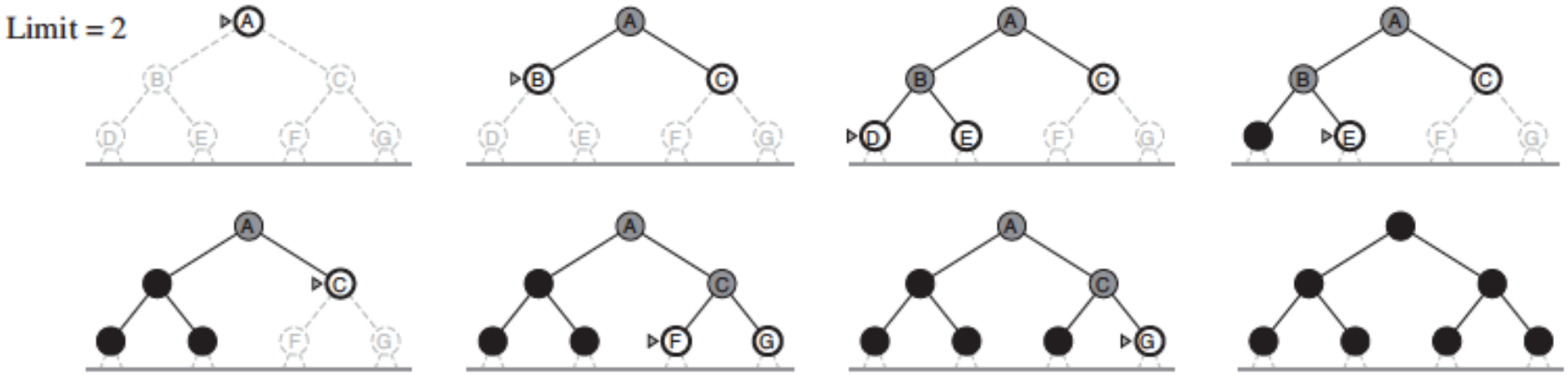
- Like DFS, its memory requirements are : $O(b \times m)$
- Like BDS, complete when b is finite and cost is nondecreasing of the depth of the node (or identical for all steps)

Iterative deepening search

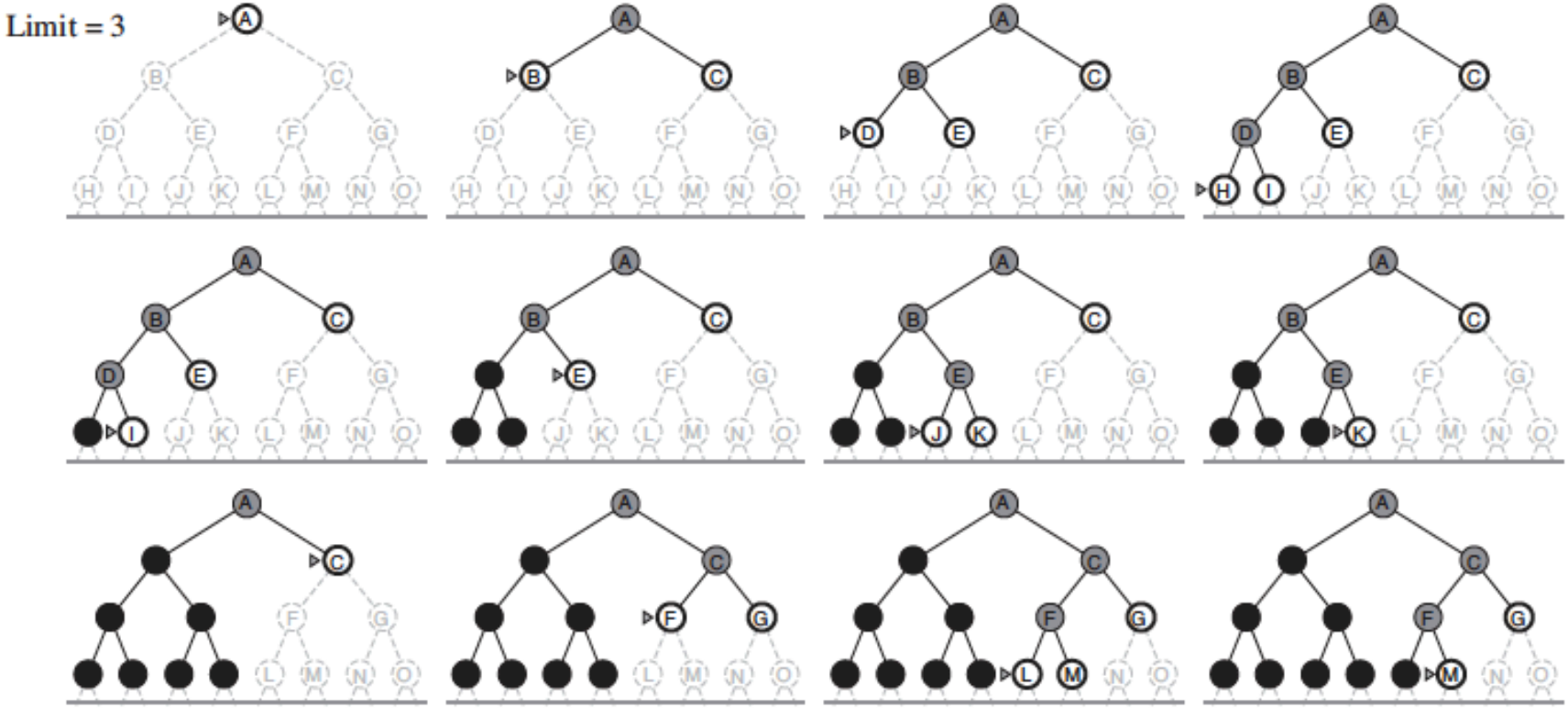
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ cutoff **then return** *result*



Iterative deepening search



Iterative deepening search



Time complexity

Overall number of nodes generated in the worst case is

d level – $1 \times b^d$

d-1 level – $2 \times b^{d-1}$

…

1 level – d times b

$$d(b) + (d-1)\, b^2 + \cdots + (1)b^d = O(b^d)$$

Same as BFS

In general, iterative deepening is the *preferred uninformed search method when the search space is large and the depth of the solution is not known*.

Comparing uninformed search strategies

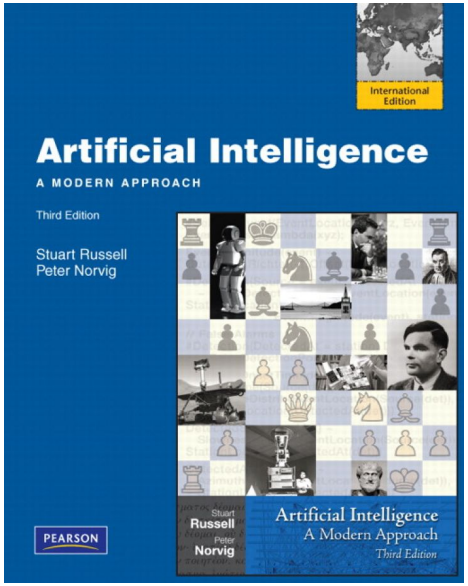
Evaluation of tree-search strategies.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Iterative Deepening
Complete?	Yes^a	Yes^{a,b}	No	Yes^a
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bd)$
Optimal?	Yes^c	Yes	No	Yes^c

b – branching factor
d – depth of the shallowest solution
m – maximum depth

superscripts
^{a} – complete if b < ∞
^{b} – complete if step cost > ϵ
^{c} – optimal if step costs are identical

Reading List
Կարդալ



Chapter 3.4 (both versions)

https://github.com/samvelyan/Intelligent-Systems

Next Lecture

1. Informed search strategies
2. The code and instructions for Practical Assignment 1 can be downloaded from **https://github.com/samvelyan/Intelligent-Systems**

Questions?
Հարցե՞ր