EECS 391
Intro to AI

Problem Solving by Search

L3: Thu, Sep 7, 2017

Outline

- Problem solving agents
- Defining problem spaces
- Search algorithms
  - types
  - implementation
  - characterizing
- Uniformed search strategies
  - breath-first search
  - depth-first search
  - iterative deepening
  - bi-directional search

Problem solving by search: problem space and goal

- problem solving by search:
  - **define a problem space** in which we can find the solution

- **goal** of search:
  - find the **state** representing the (or a) **solution**

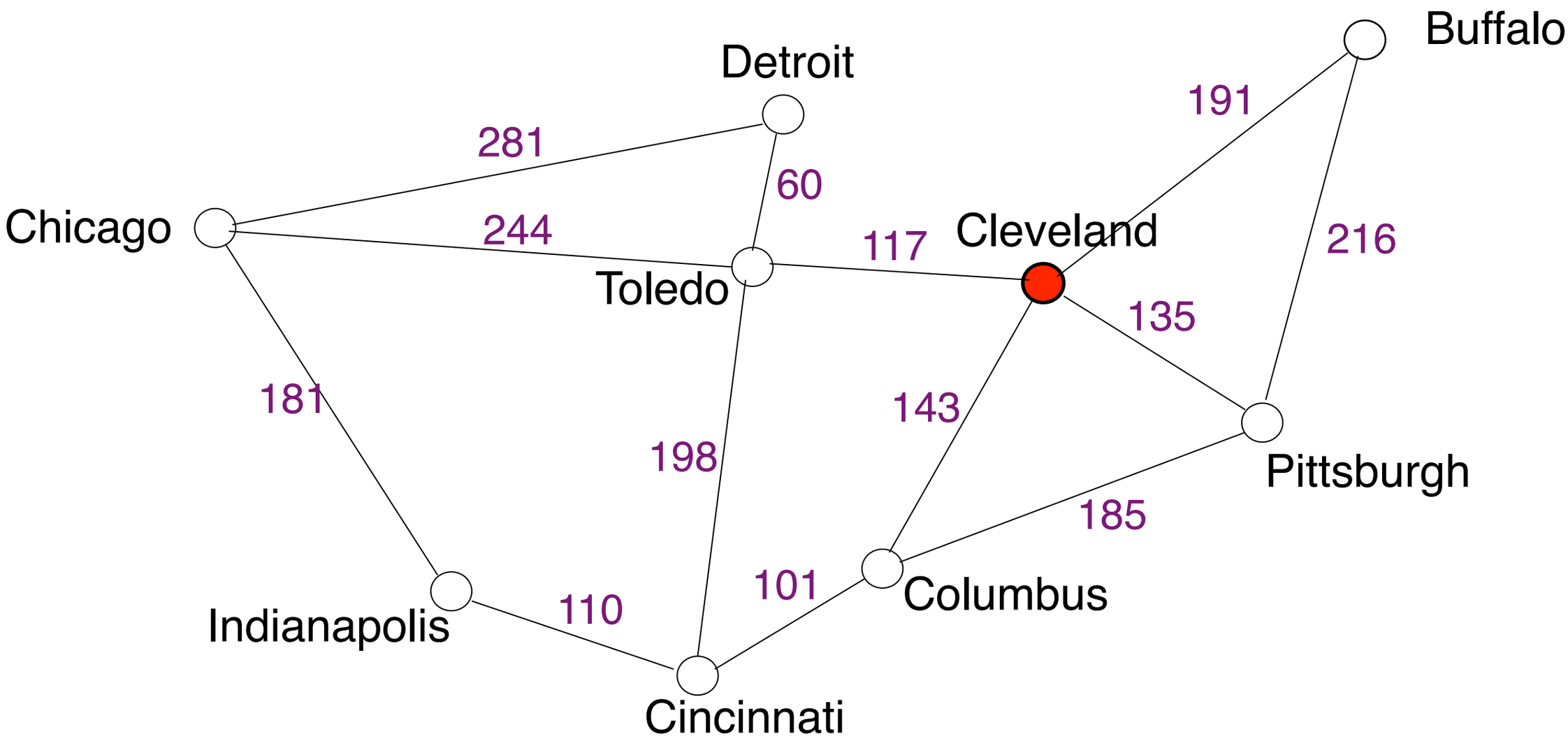
- **uniformed** search:
  - only given the problem definition
  - ***no a priori knowledge*** of where to look for the solution

How do we define the problem space?

- Problem: *How do we get to Cincinnati?*
  - by plane?
  - by car?
  - by bike?

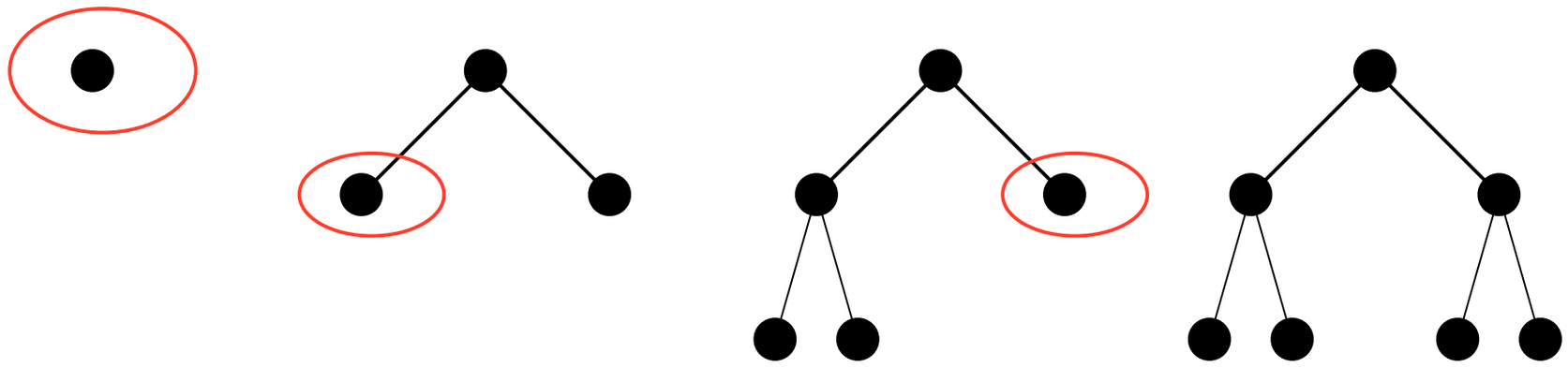
- Need to define:
  1. **initial state**
  2. available **actions**
  3. **goal test**
  4. action **cost** (which we'll address next lecture)

Representing the problem space with a graph



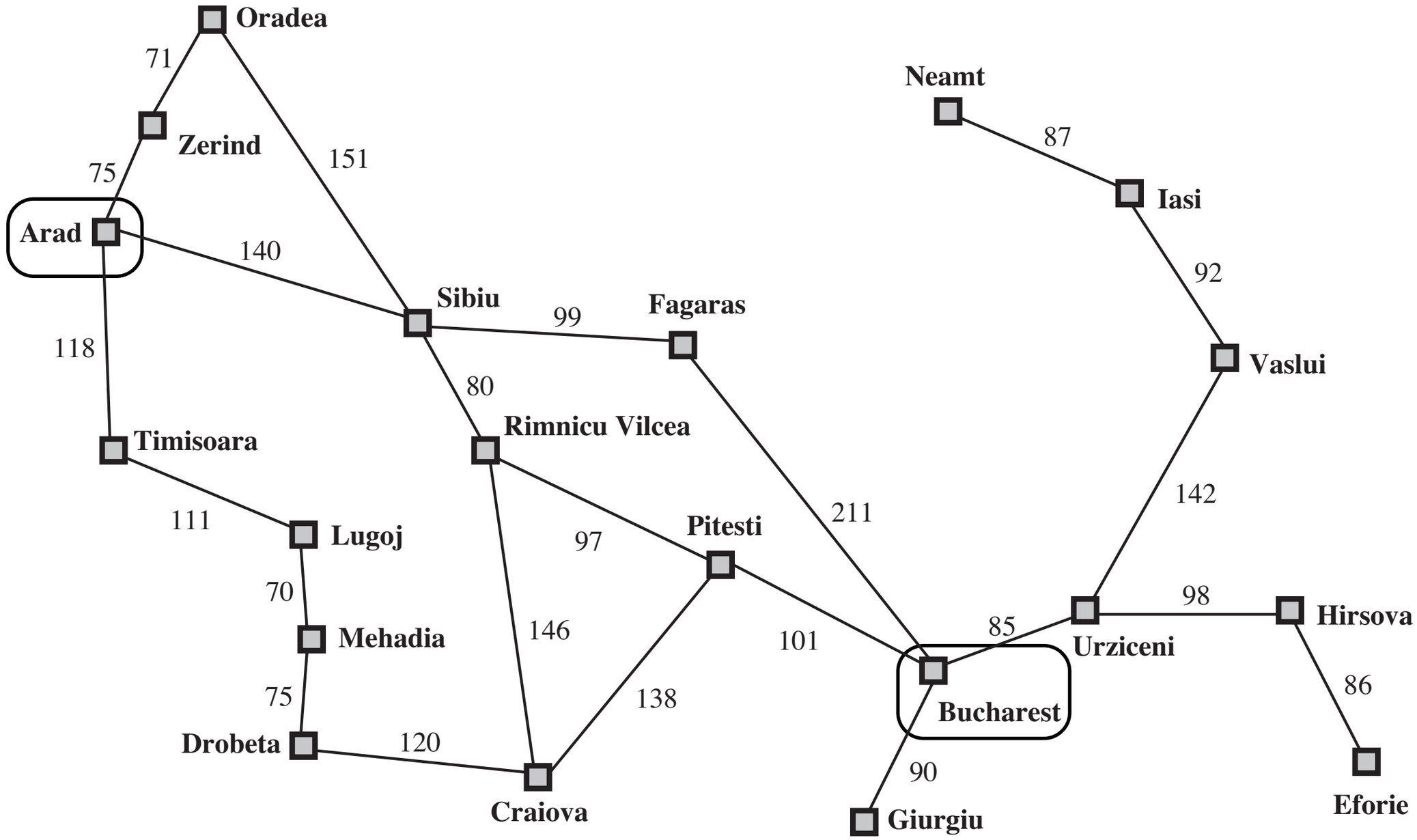
Breadth-first search

function BREADTH-FIRST-SEARCH (*problem*) **returns** a solution or failure
  return GENERAL-SEARCH (*problem*, ENQUEUE-AT-END)



- Type type of search is determined by how the *fringe* is expanded
- breadth-first search is implemented using a first-in first-out (FIFO) *queue*
  - all new nodes go at end of queue
  - shallow nodes are expanded before deeper nodes

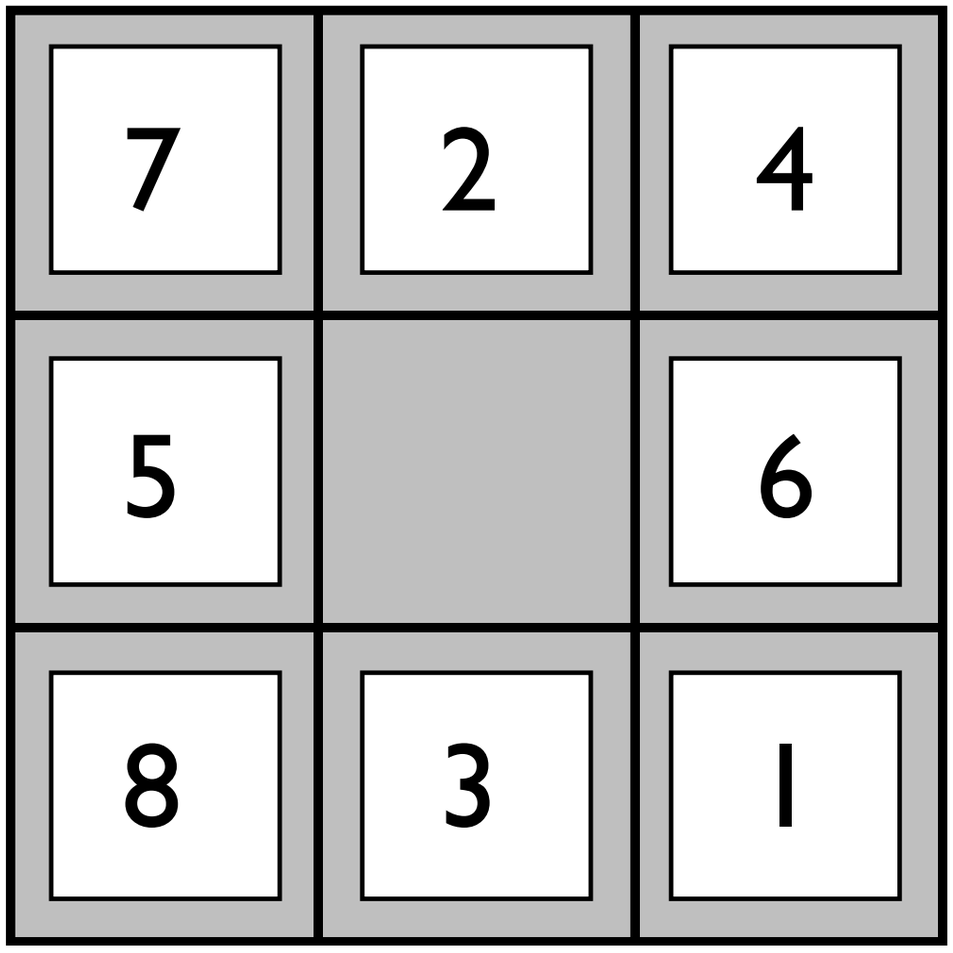
BFS on Romanian road map example



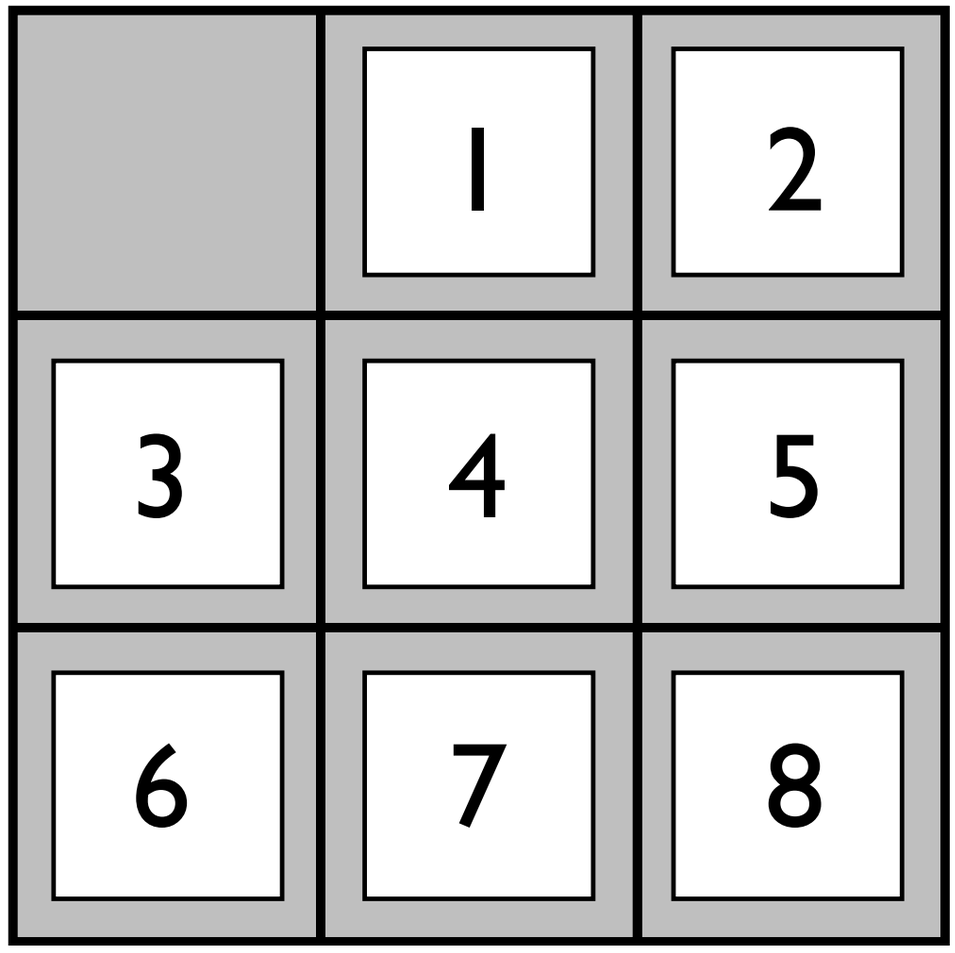
In absence of knowledge, node expansion order is arbitrary, i.e. alphabetical.
(Example on board)

The 8-puzzle (a sliding block puzzle)

start state



goal state



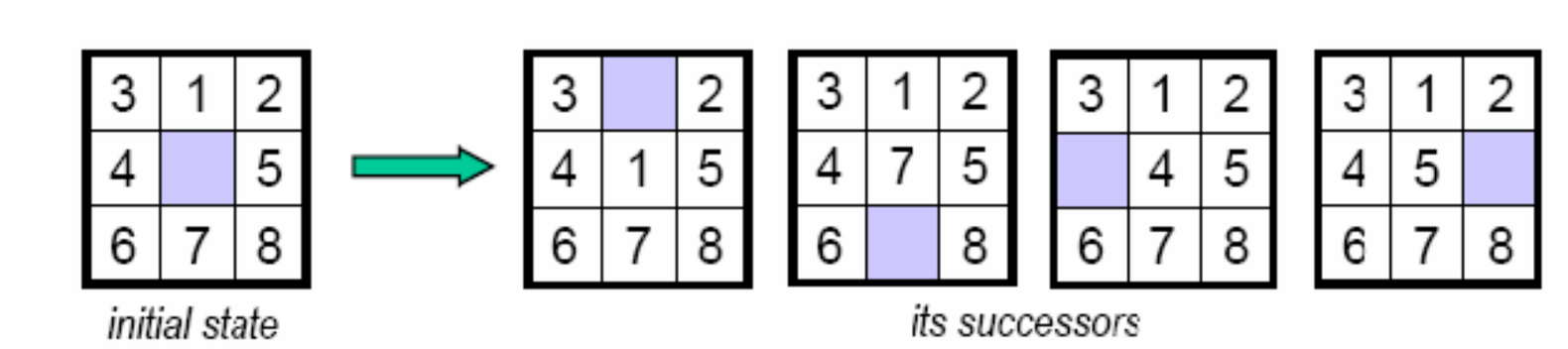
general class is NP-complete

How do we define the problem space?

state space, actions, and goal

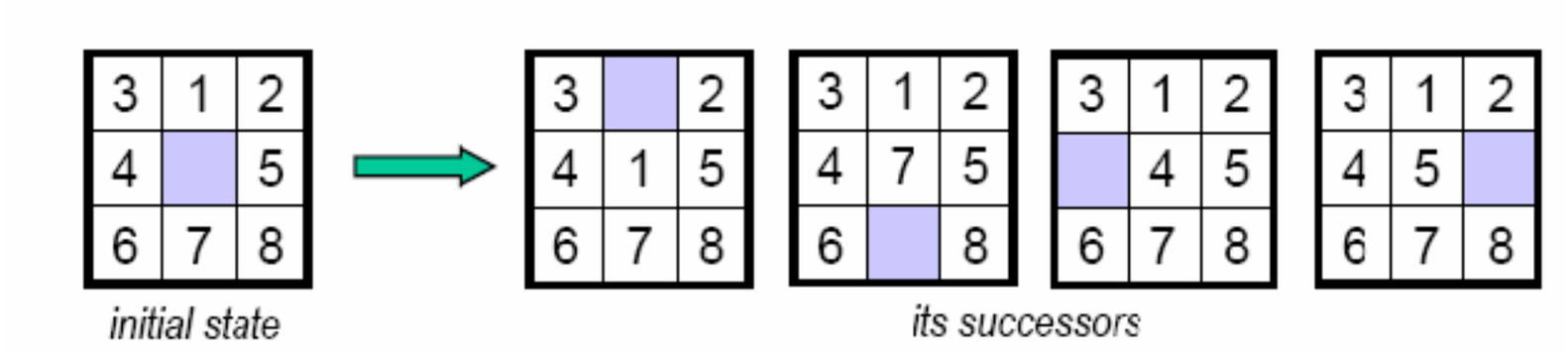
State-space search

- Each successor defines different moves possible from the current puzzle state

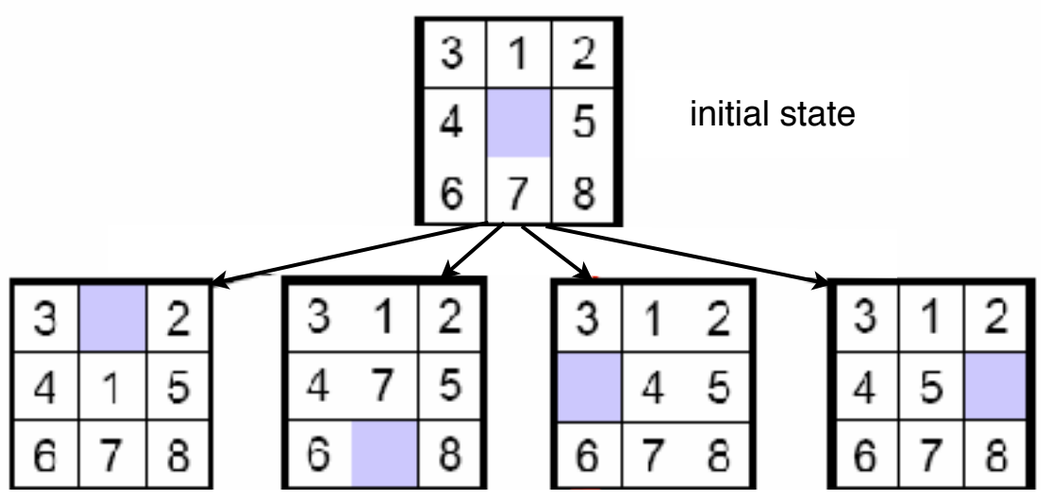


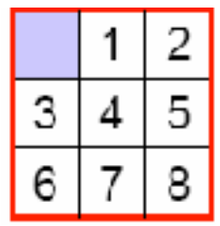
State-space search

- Each successor defines different moves possible from the current puzzle state



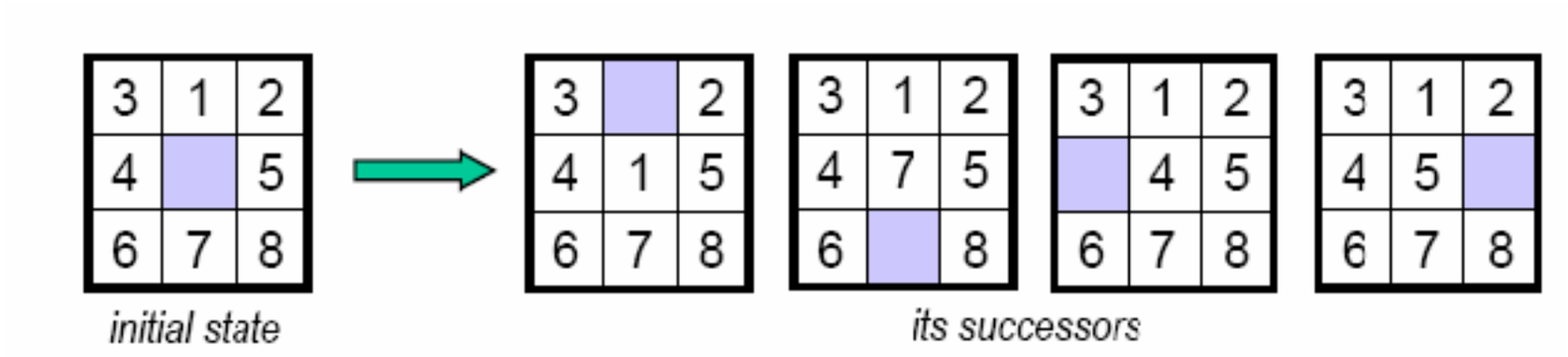
- We can employ different *search strategies* to find the goal state



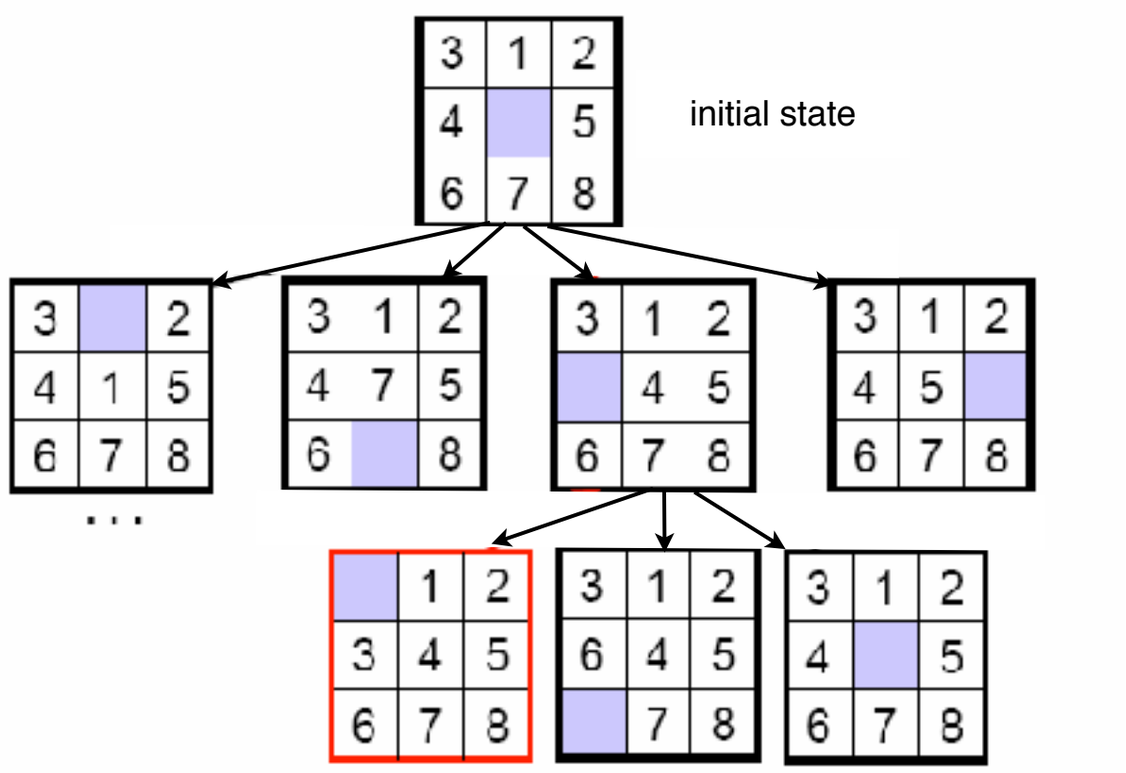


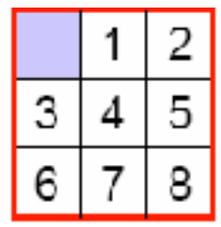
State-space search

- Each successor defines different moves possible from the current puzzle state

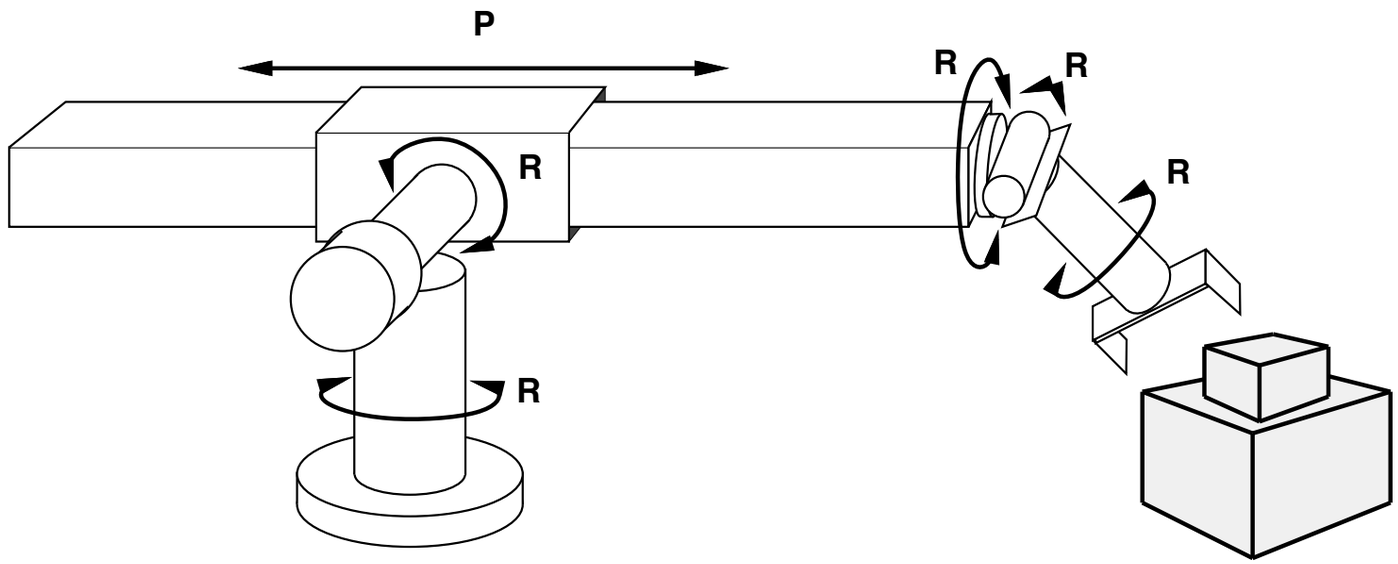


- We can employ different *search strategies* to find the goal state





Robotic arm



1. initial state?	Angles and coordinations of joints. Object shape and position
2. available actions?	motion actions of robot arms and manipulators
3. goal?	grasp/place/sort object
4. action cost?	time, accuracy

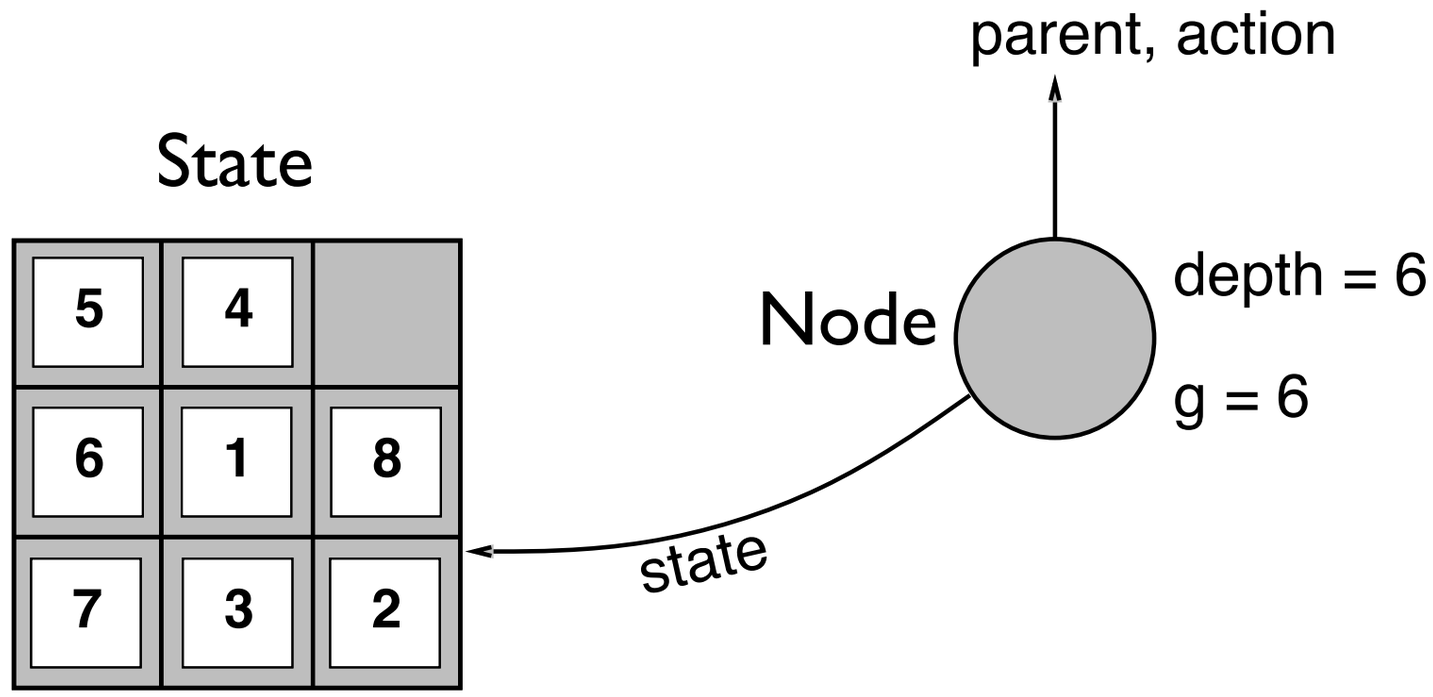
Other types of problems

- puzzles, e.g. Rubik's cube ~ 10^{19} states!
- google maps, GPS navigators
- internet spidering
- decoding
- routing (paths, vehicles, salesman)
- scheduling (meetings, conferences, airlines)
- design (electric grid, VLSI layout)
- protein folding

Implementing search

- state is the representation of the environment (e.g. city, or board position)
- graph nodes are created during search, could be infinite
- need to recognize
  - goal state
  - repeated states

- The **node** data type:
  - state
  - parent node
  - action
  - cost
  - depth



Generic search function outline

function Tree-Search(*problem, strategy*) **returns** solution or failure

initialize tree using initial state of *problem*

loop

if no candidates for expansion **return** failure

choose leaf node for expansion using *strategy*

if node contains goal state **return** solution

else expand node and add resulting nodes to search tree

- How do we specify the *problem*?
- What *strategies* are there? How do we implement them?

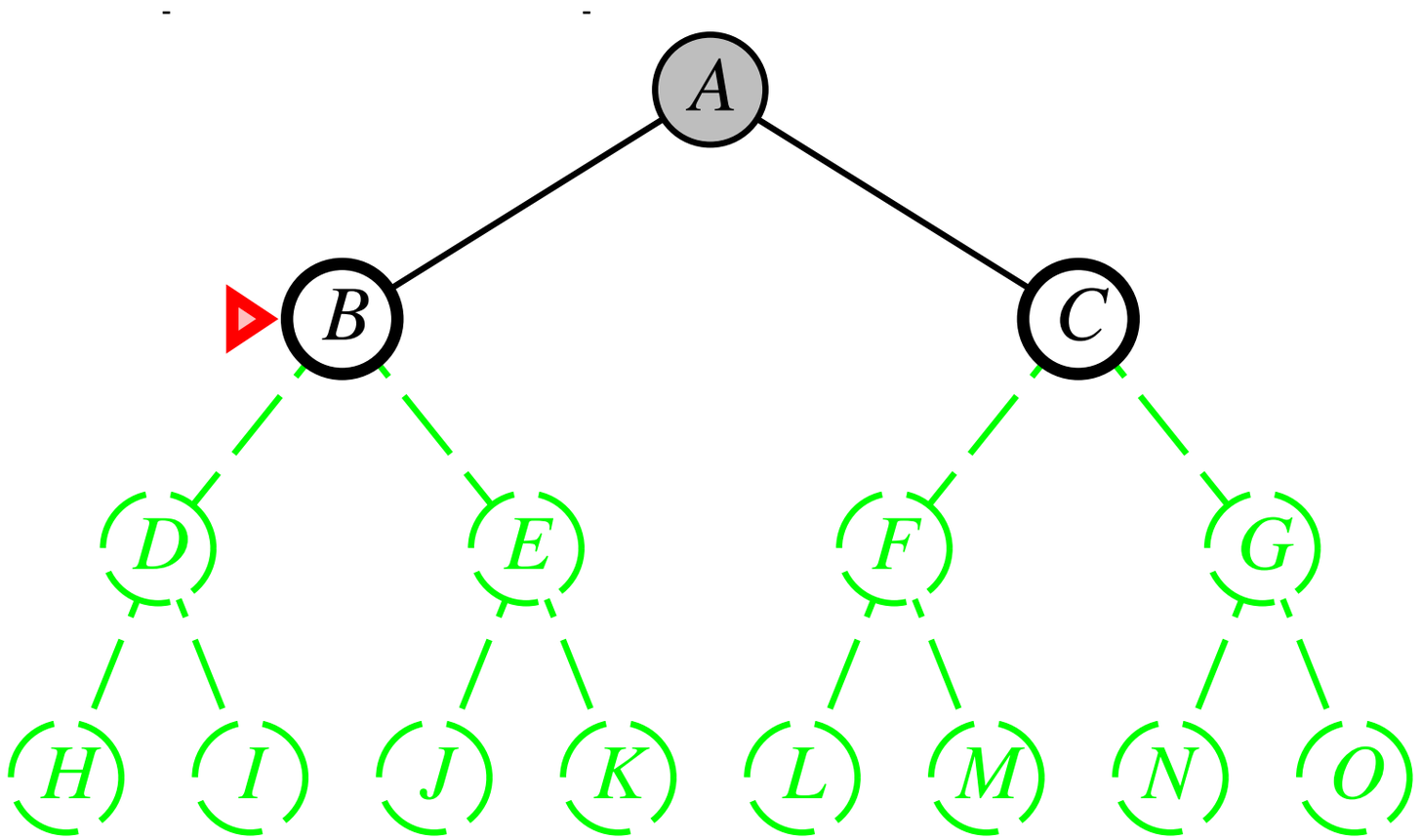
Uninformed search strategies

- Uninformed search strategies can only distinguish goal states from non-goal states, i.e. you can't tell when you're getting closing

- Types of uninformed search strategies:
  - Breadth-first search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bi-directional search

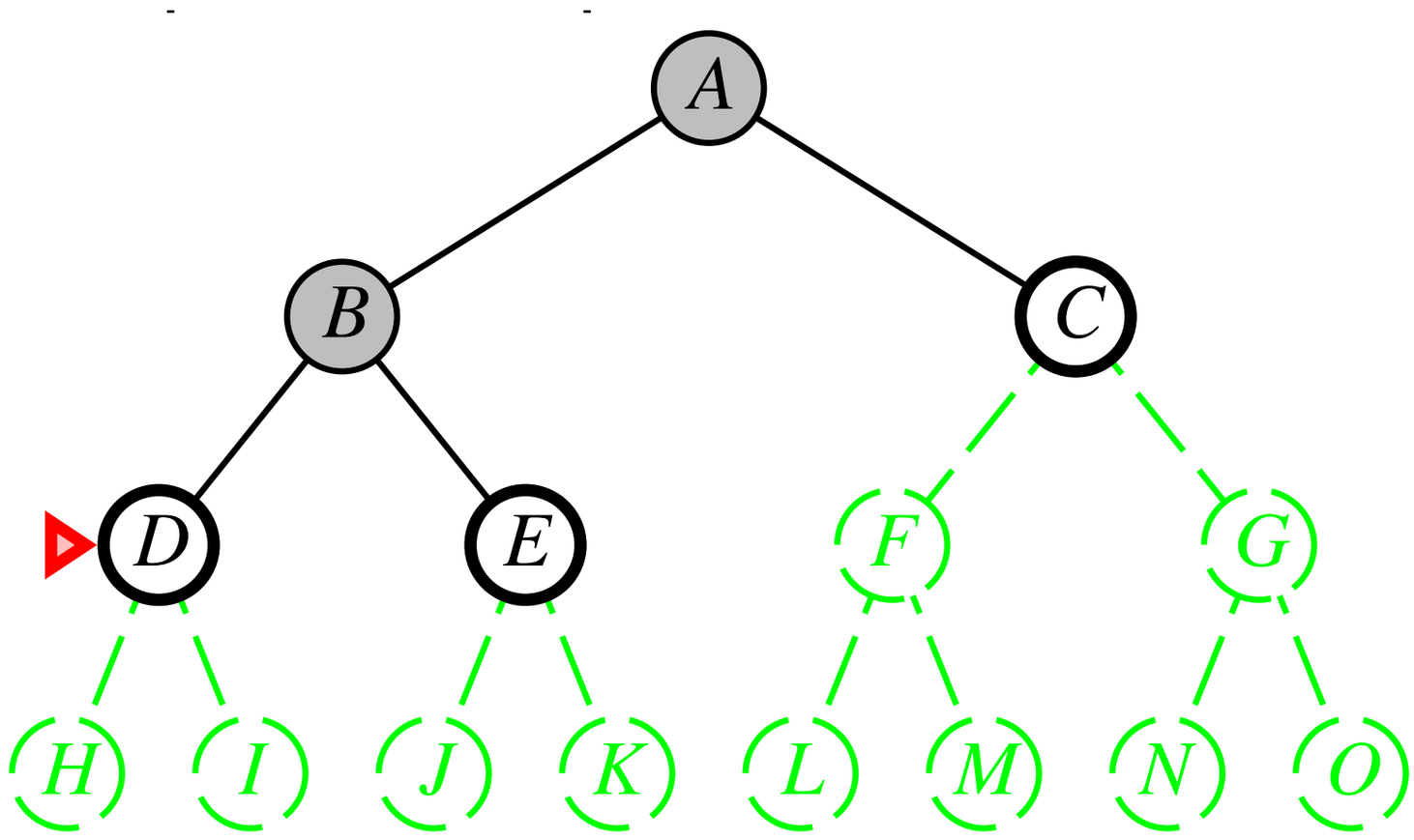
Depth-first search

- expand *deepest* unexpanded node
- when expanding the fringe, the successors are put on the front (LIFO)



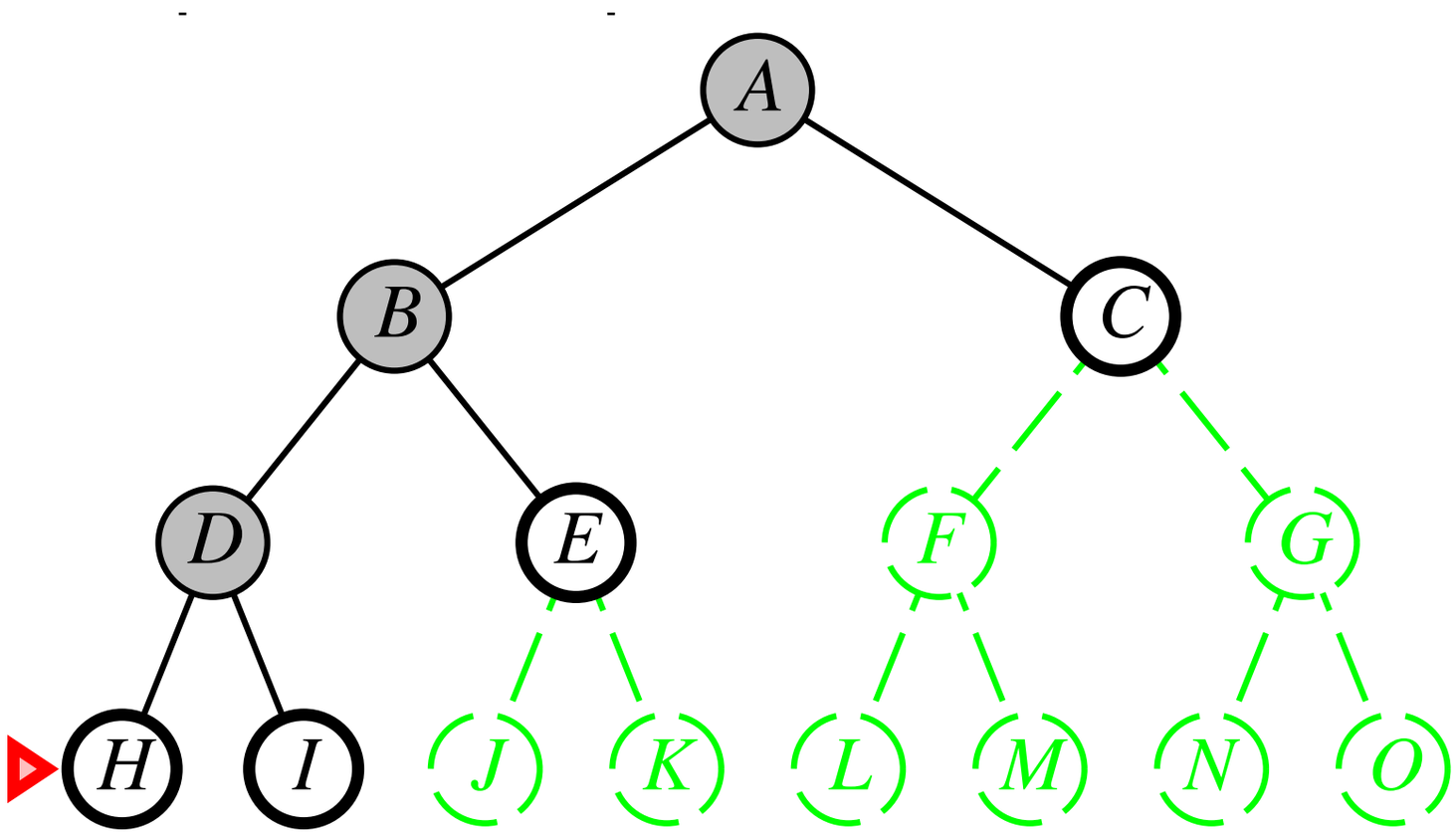
Depth-first search

- expand *deepest* unexpanded node
- when expanding the fringe, the successors are put on the front (LIFO)



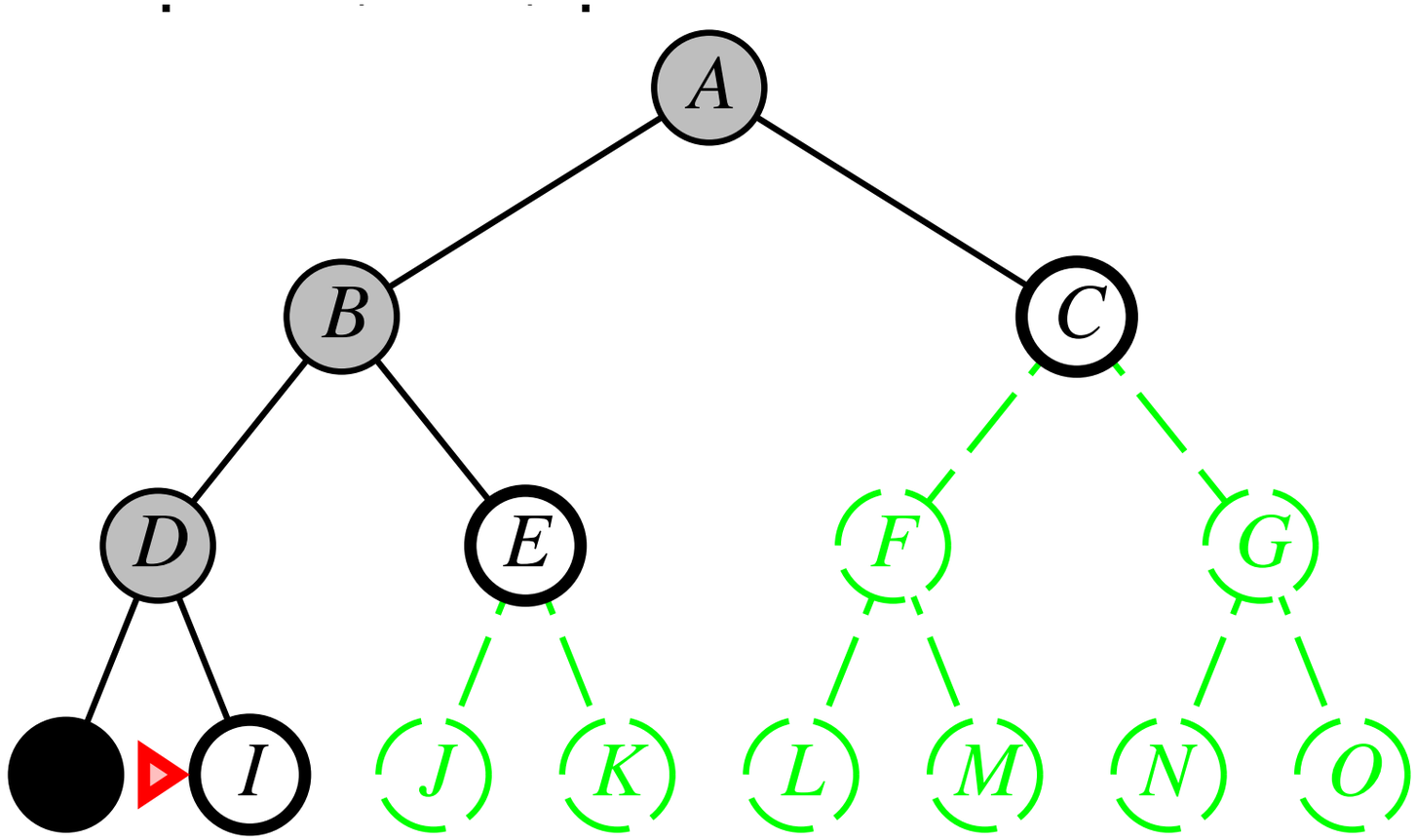
Depth-first search

- expand *deepest* unexpanded node
- when expanding the fringe, the successors are put on the front (LIFO)



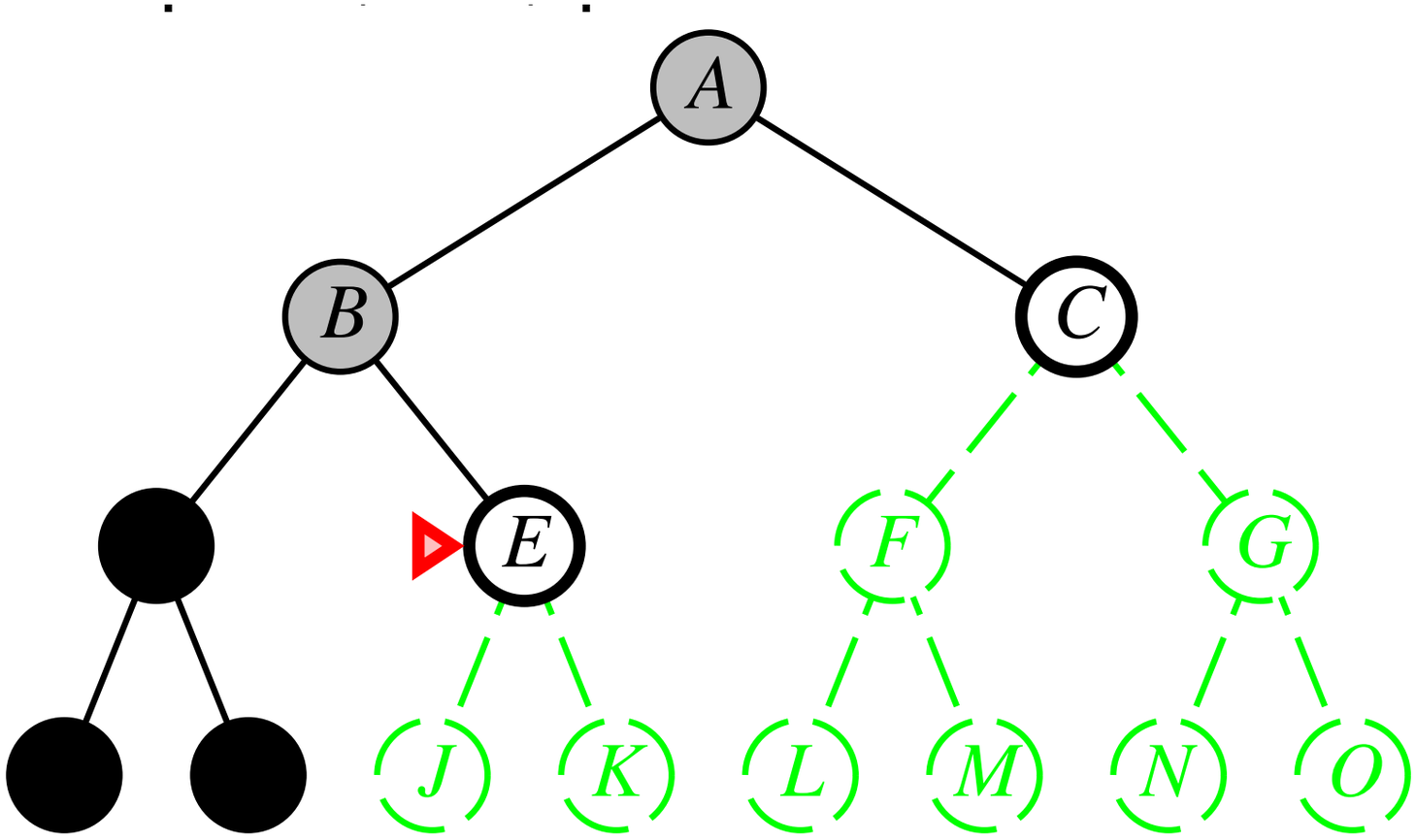
Depth-first search

- expand *deepest* unexpanded node
- when expanding the fringe, the successors are put on the front (LIFO)



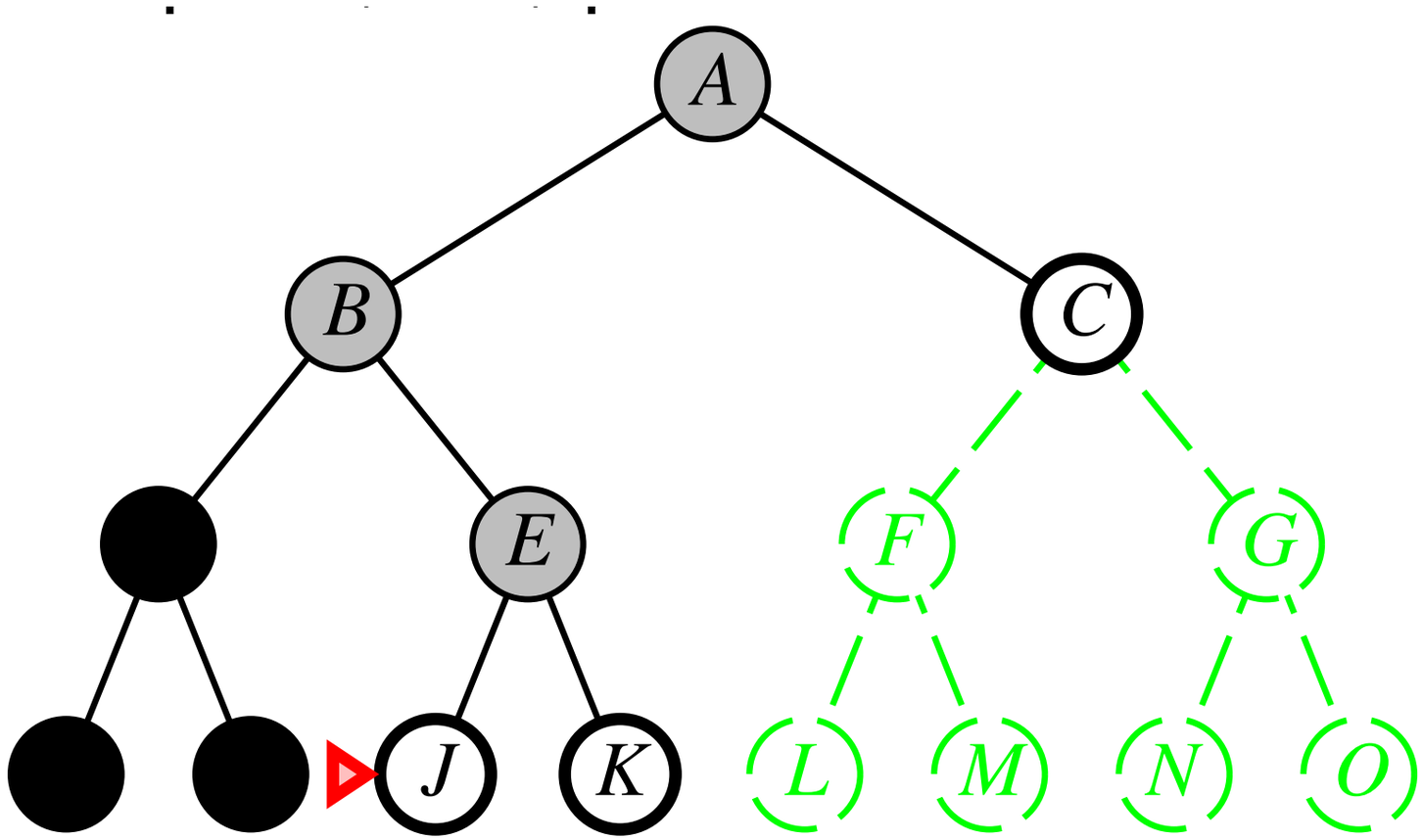
Depth-first search

- expand *deepest* unexpanded node
- when expanding the fringe, the successors are put on the front (LIFO)



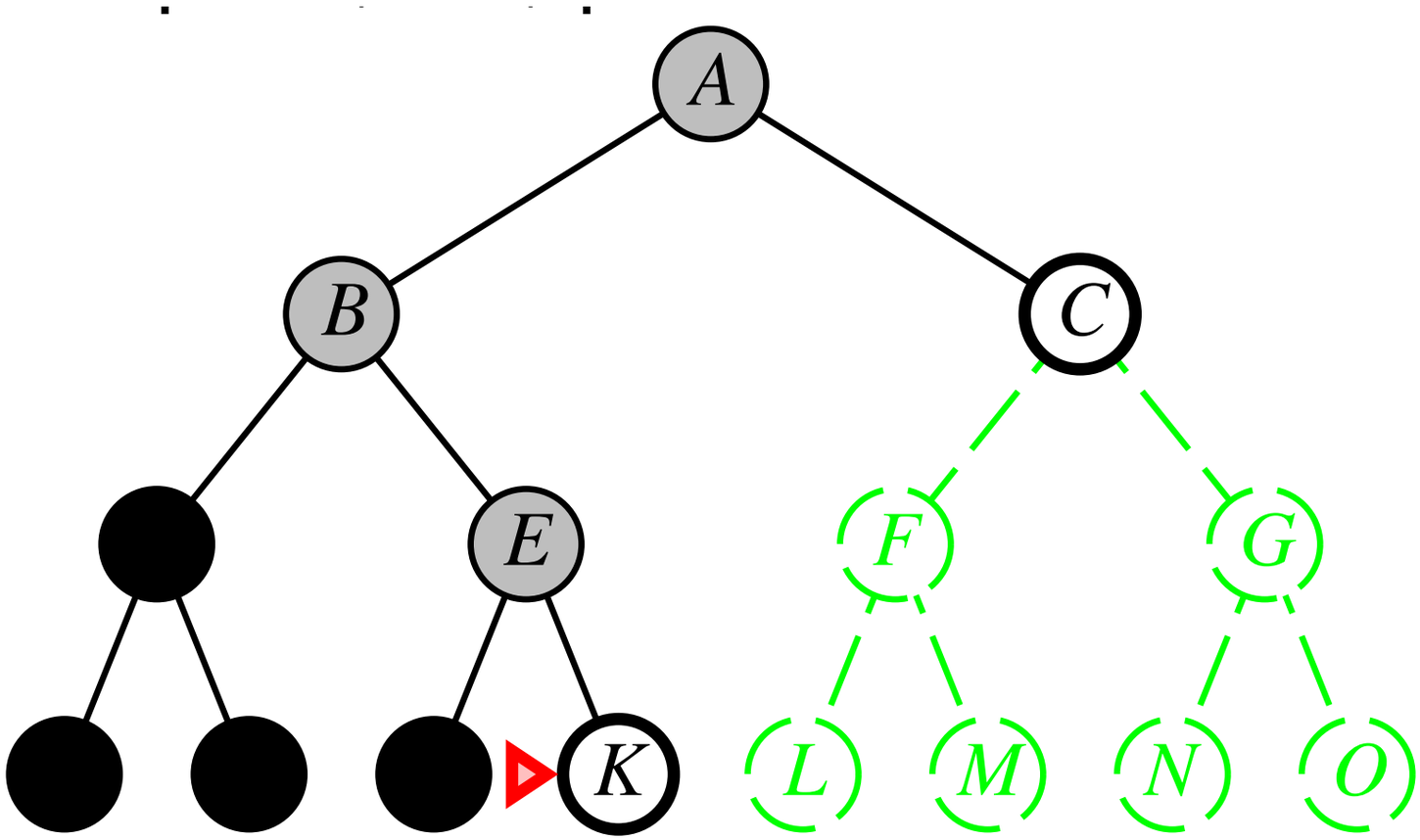
Depth-first search

- expand *deepest* unexpanded node
- when expanding the fringe, the successors are put on the front (LIFO)



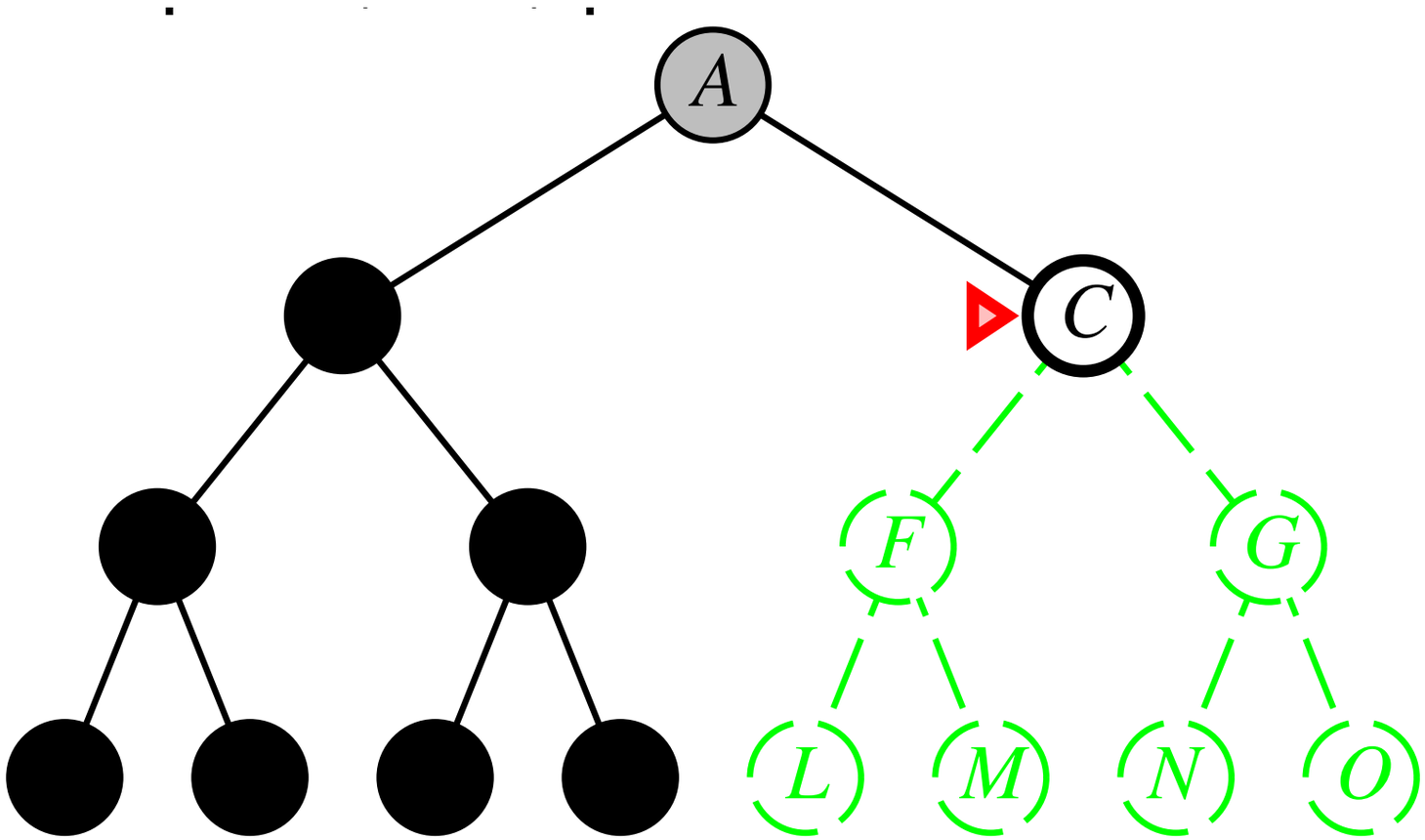
Depth-first search

- expand *deepest* unexpanded node
- when expanding the fringe, the successors are put on the front (LIFO)



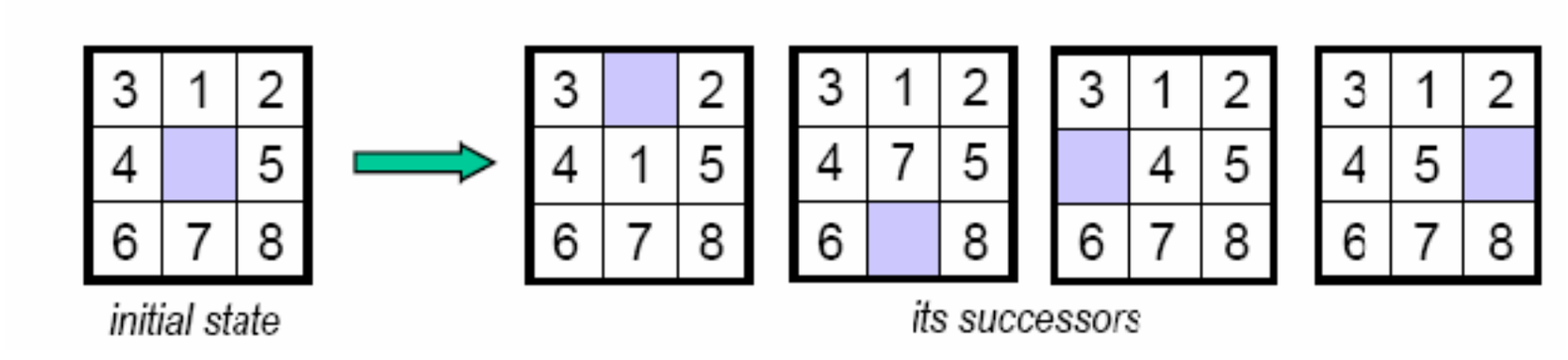
Depth-first search

- expand *deepest* unexpanded node
- when expanding the fringe, the successors are put on the front (LIFO)

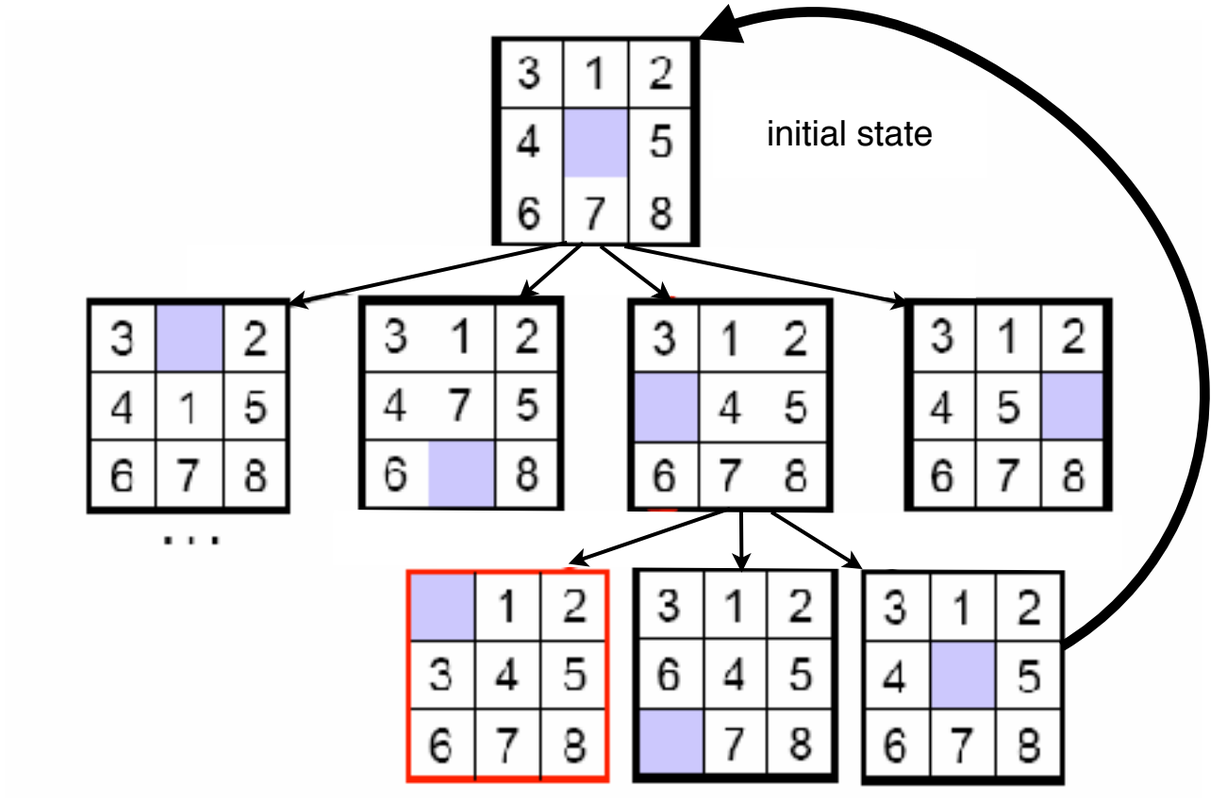


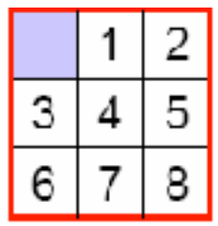
What about loops?

- Each successor defines different moves possible from the current puzzle state



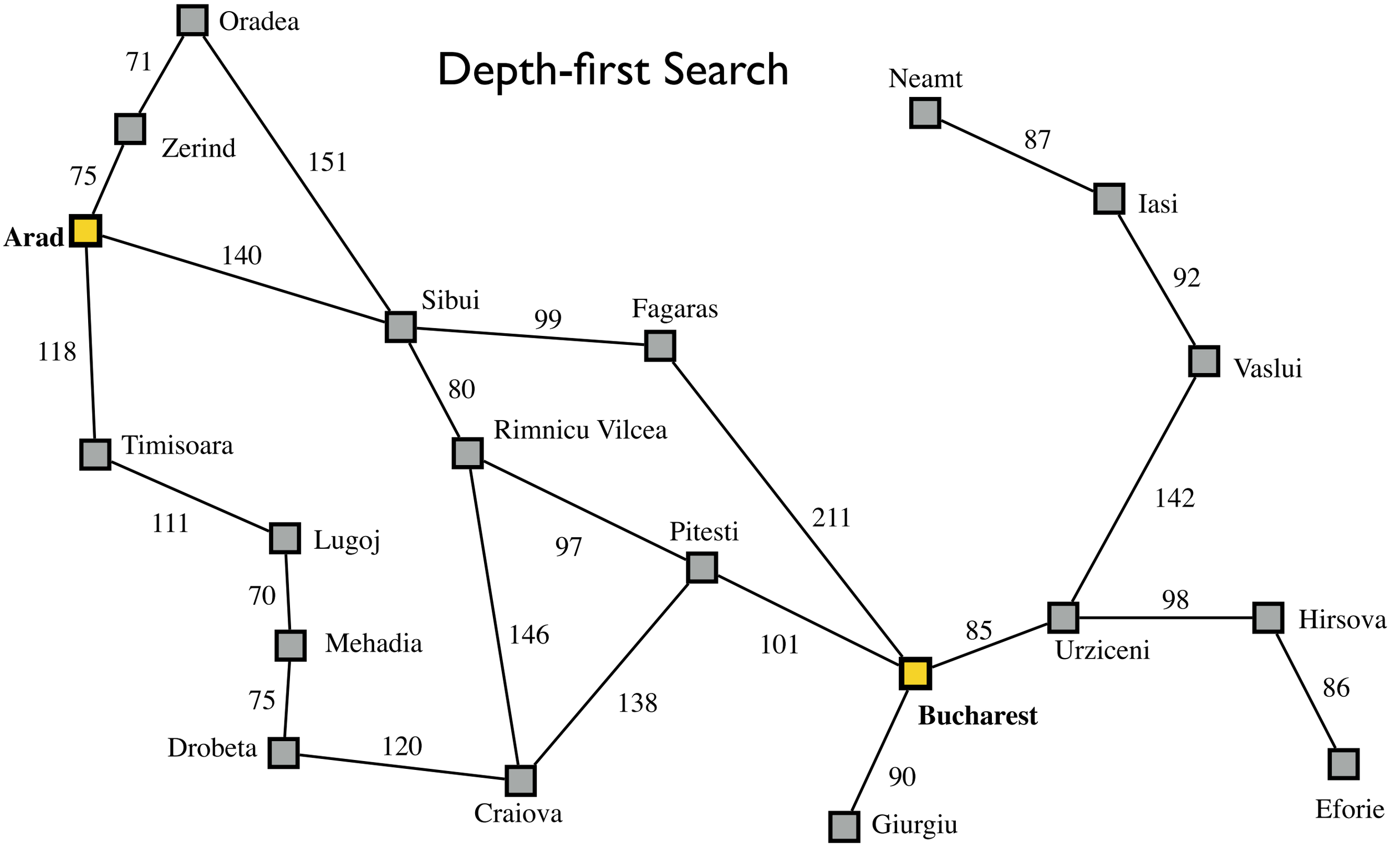
- We can employ different *search strategies* to find the goal state





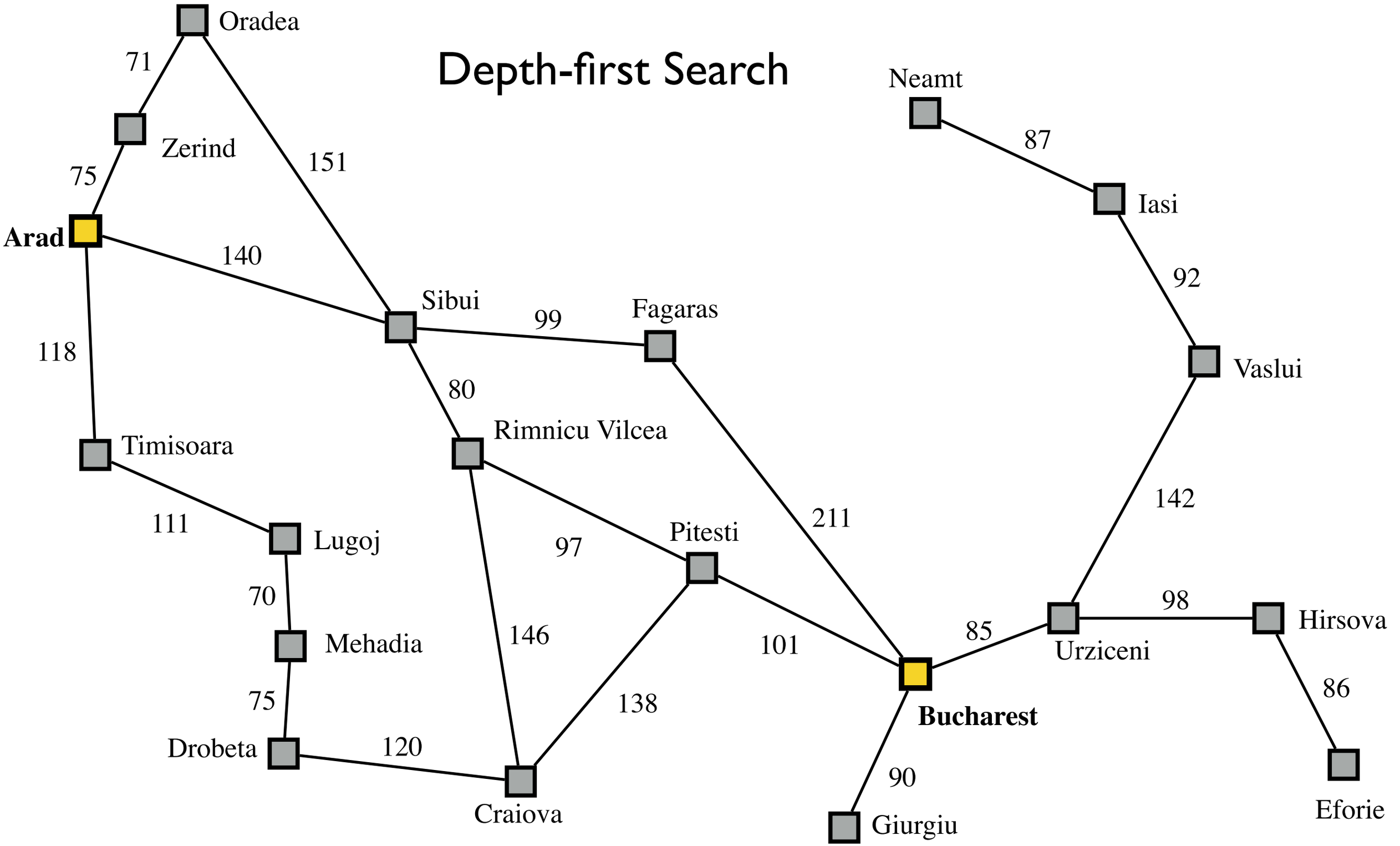
Need to mark nodes that have been visited.

Depth-first Search



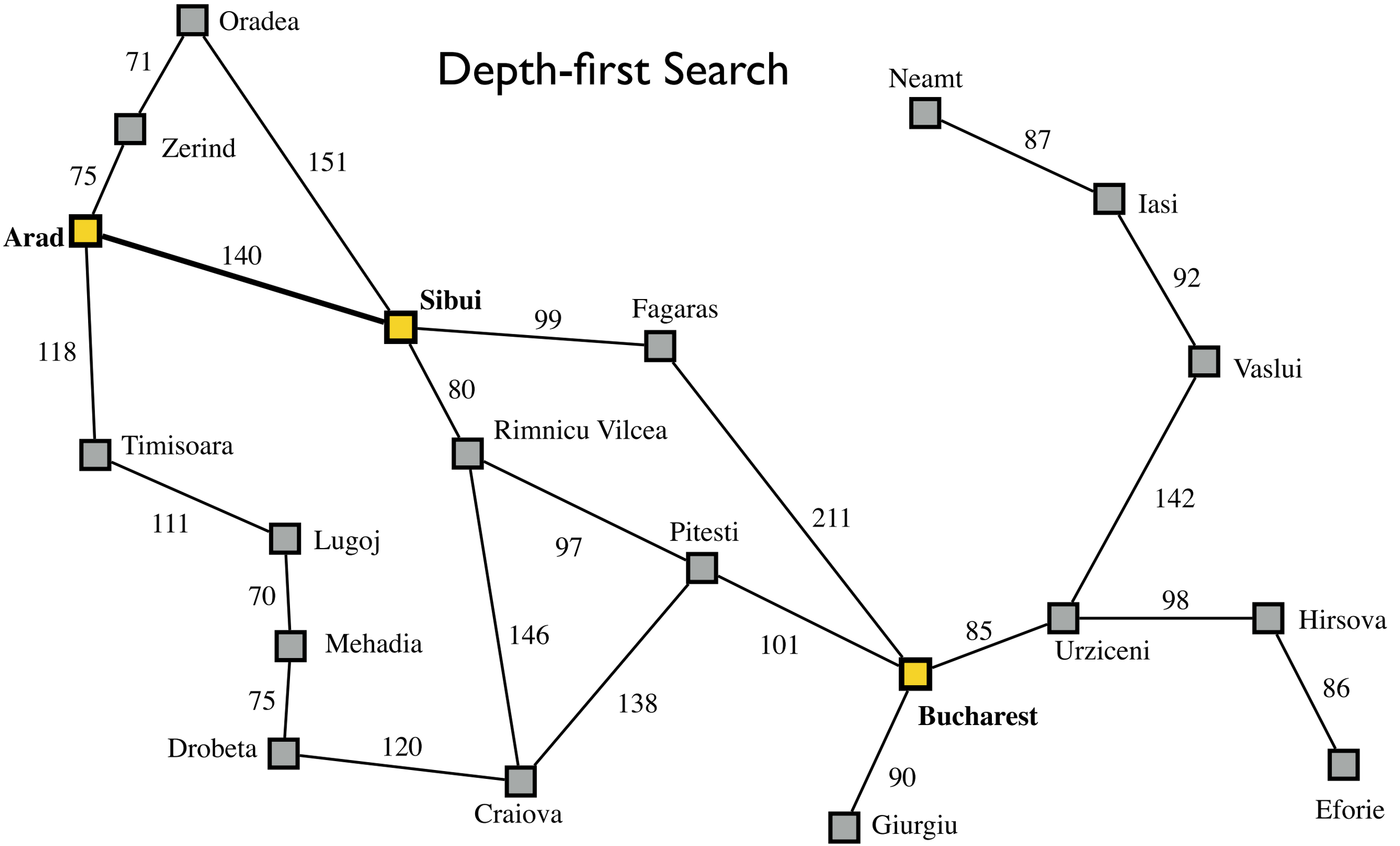
Ignore the edge costs for now: we're only trying to find the goal.

Depth-first Search



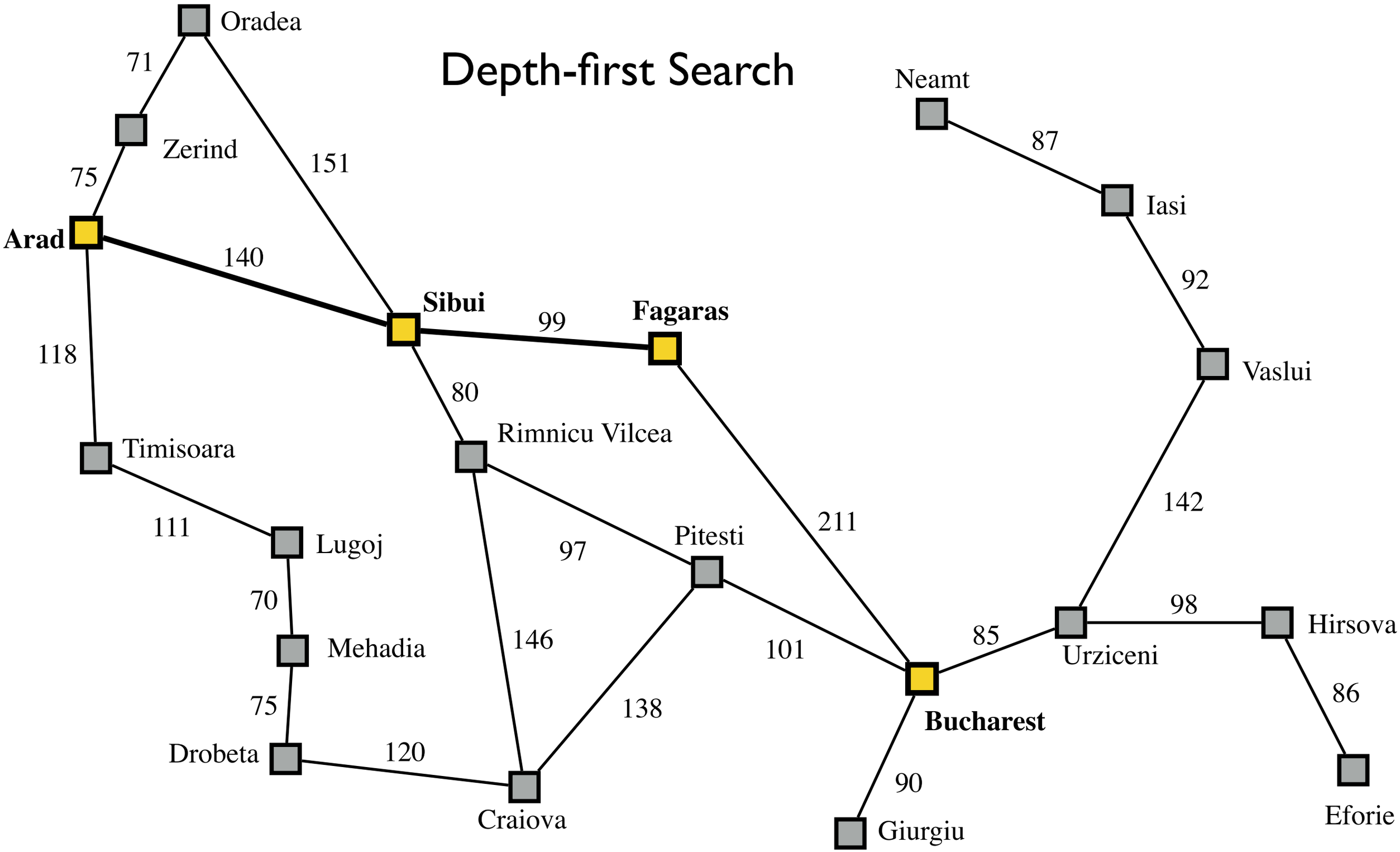
In absence of knowledge, node expansion order is arbitrary, i.e. alphabetical.

Depth-first Search



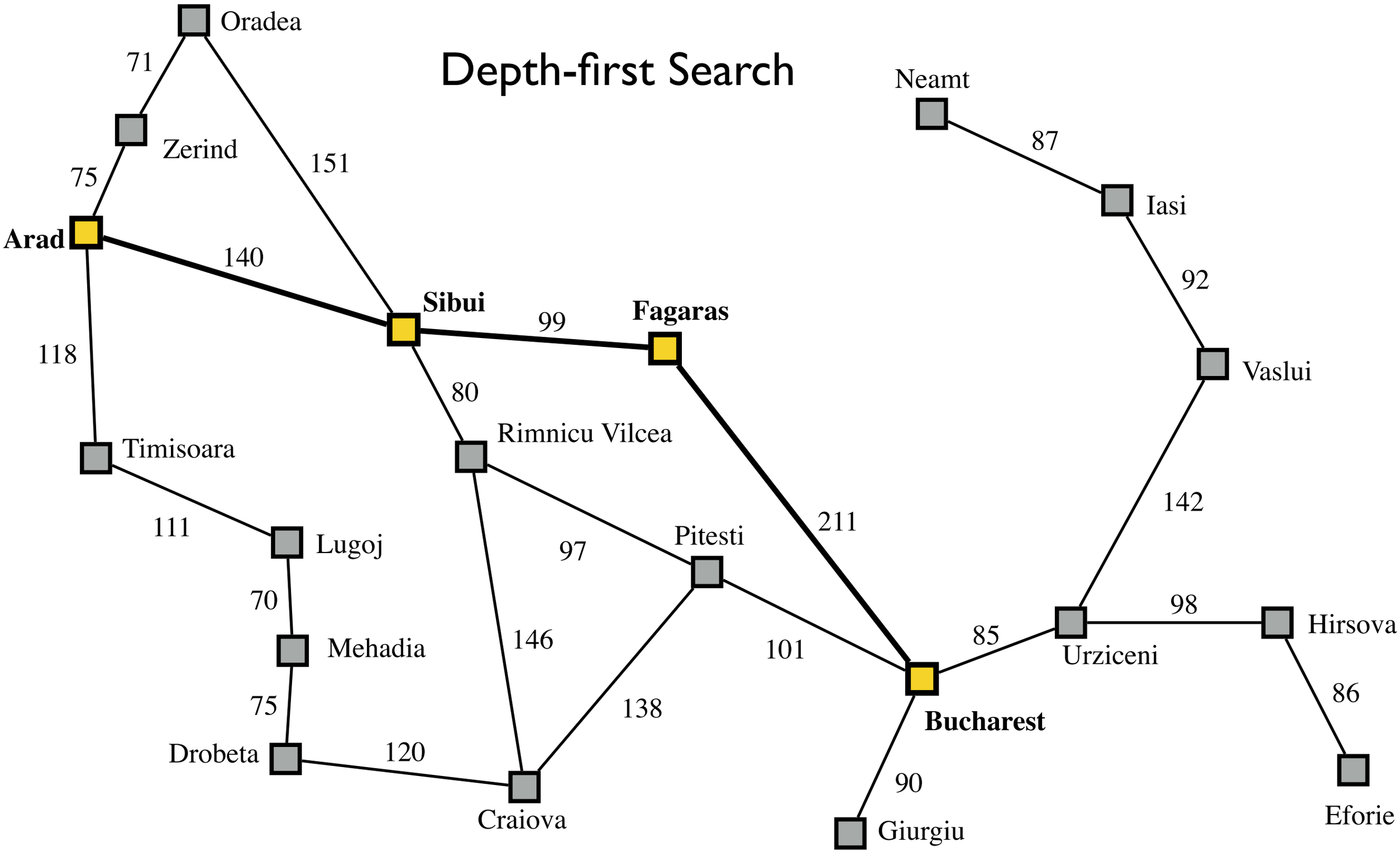
In absence of knowledge, node expansion order is arbitrary, i.e. alphabetical.

Depth-first Search



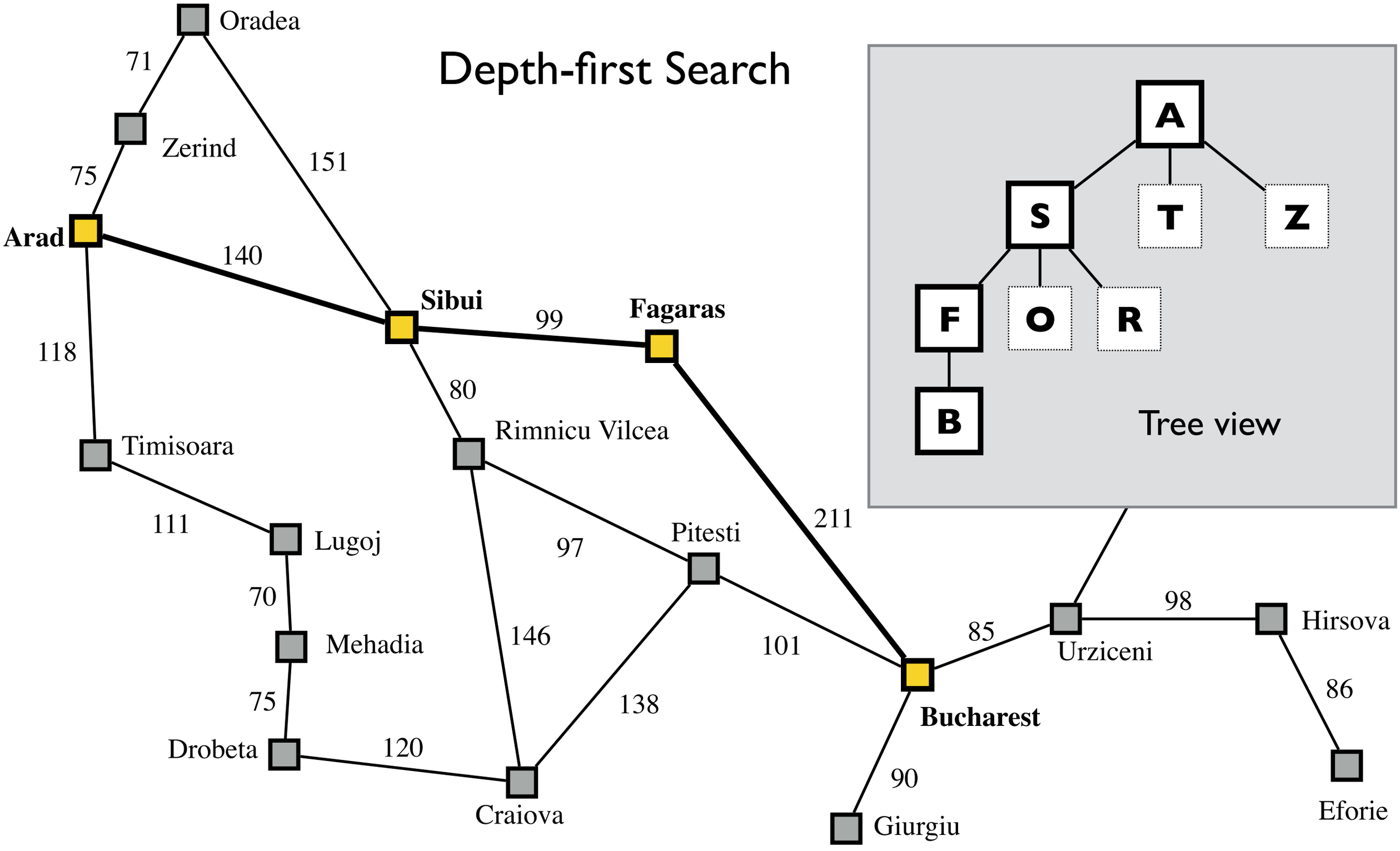
In absence of knowledge, node expansion order is arbitrary, i.e. alphabetical.

Depth-first Search



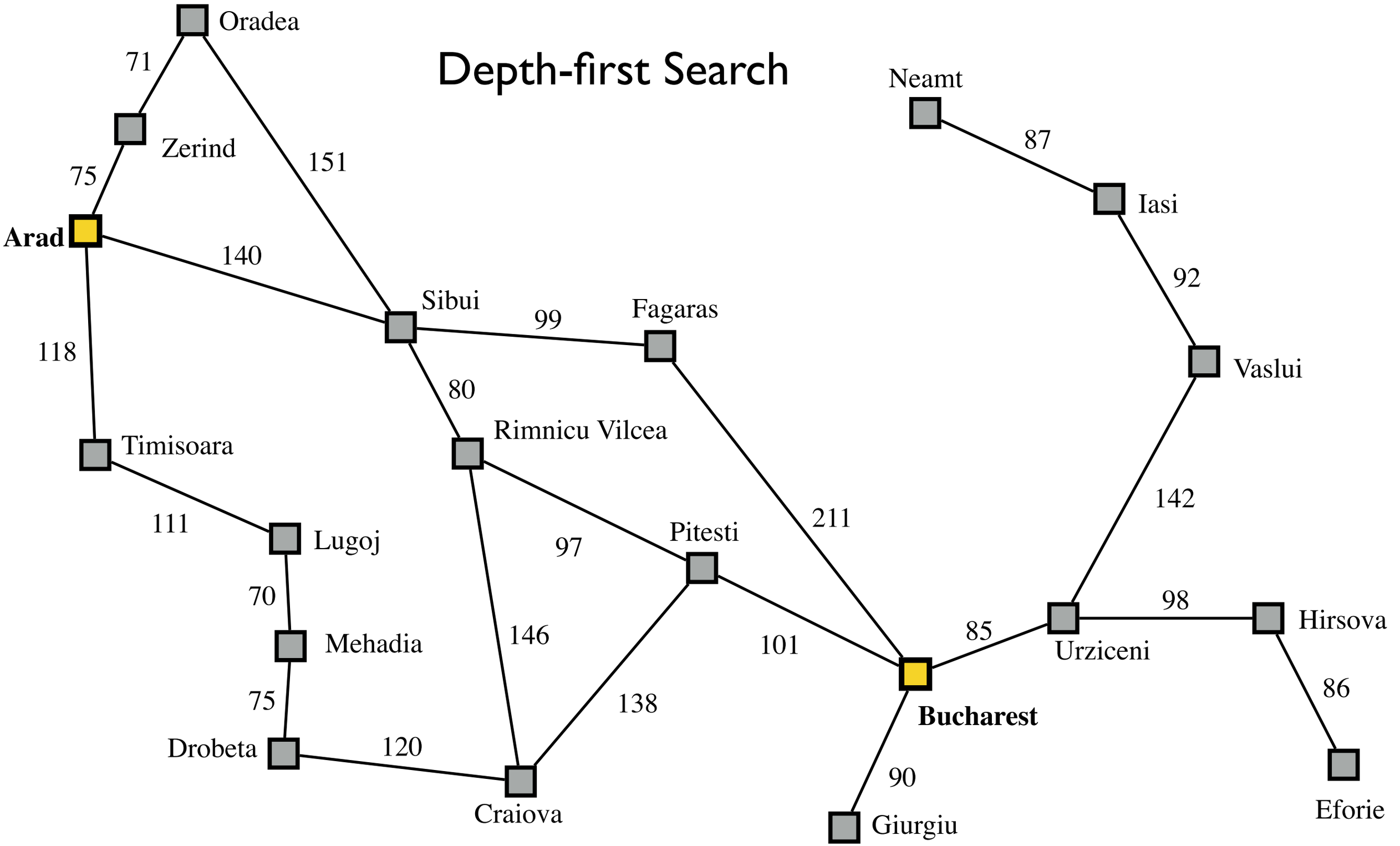
We lucked out!

Depth-first Search



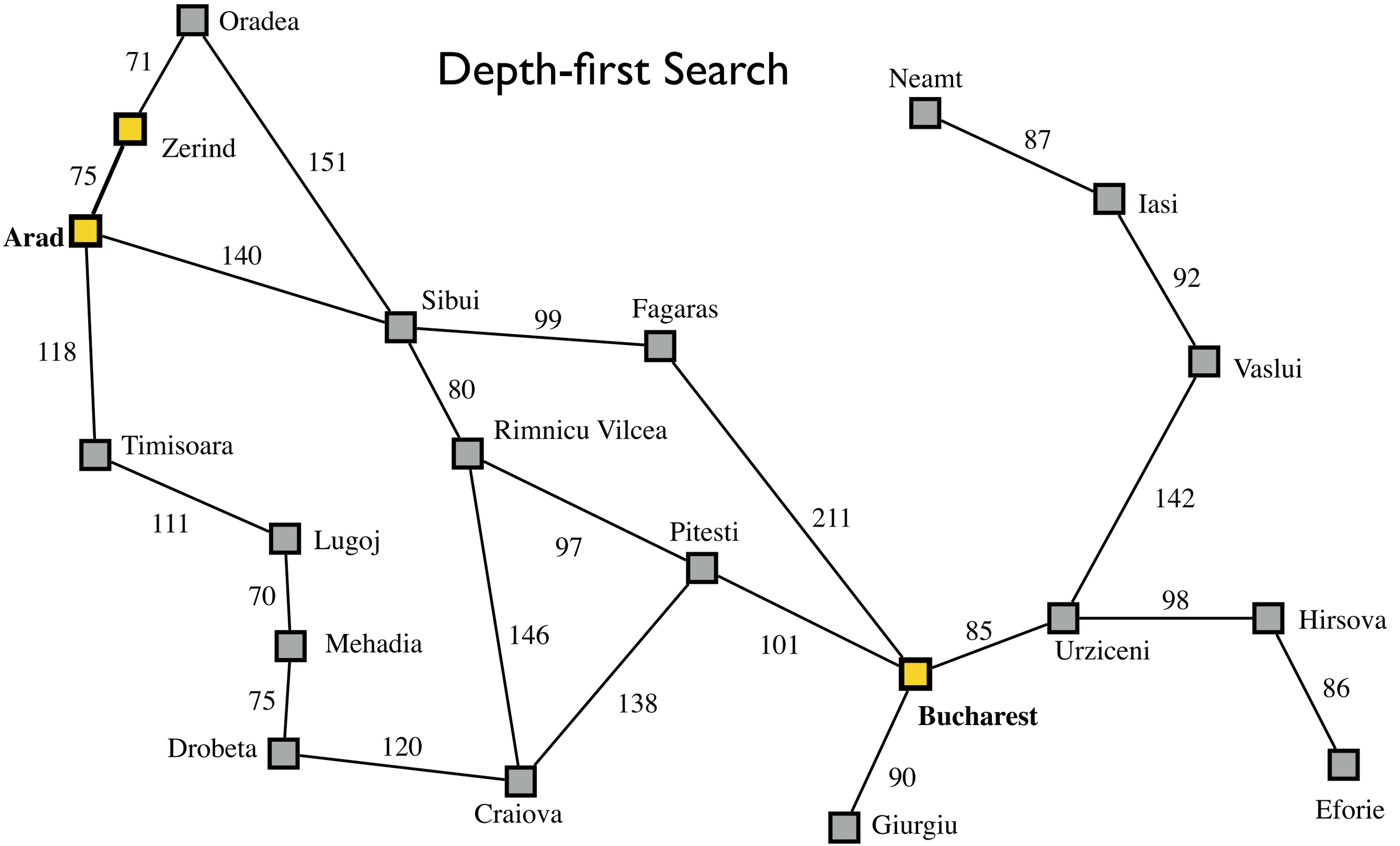
We lucked out!

Depth-first Search



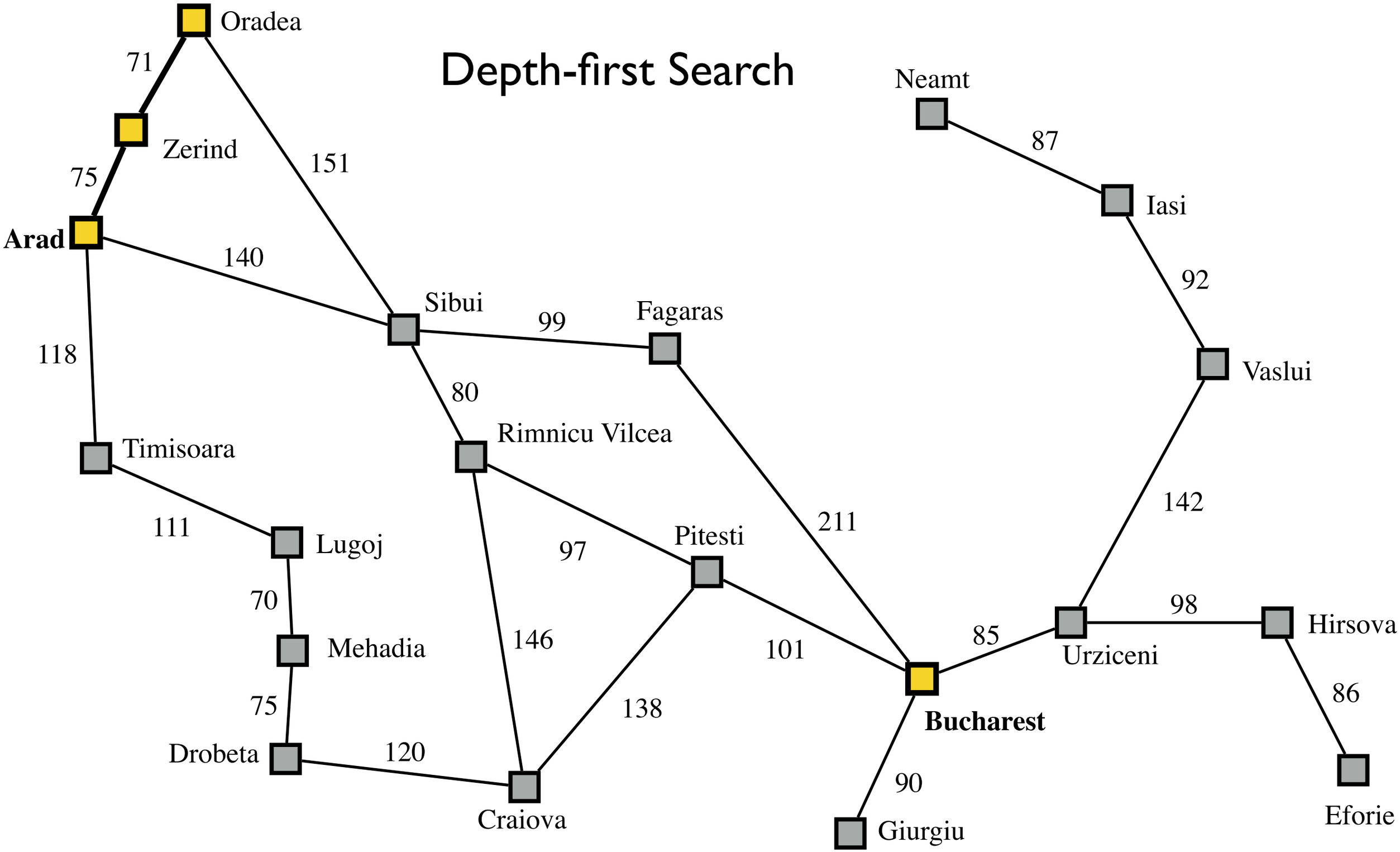
But what if we did reverse-alphabetical expansion?

Depth-first Search



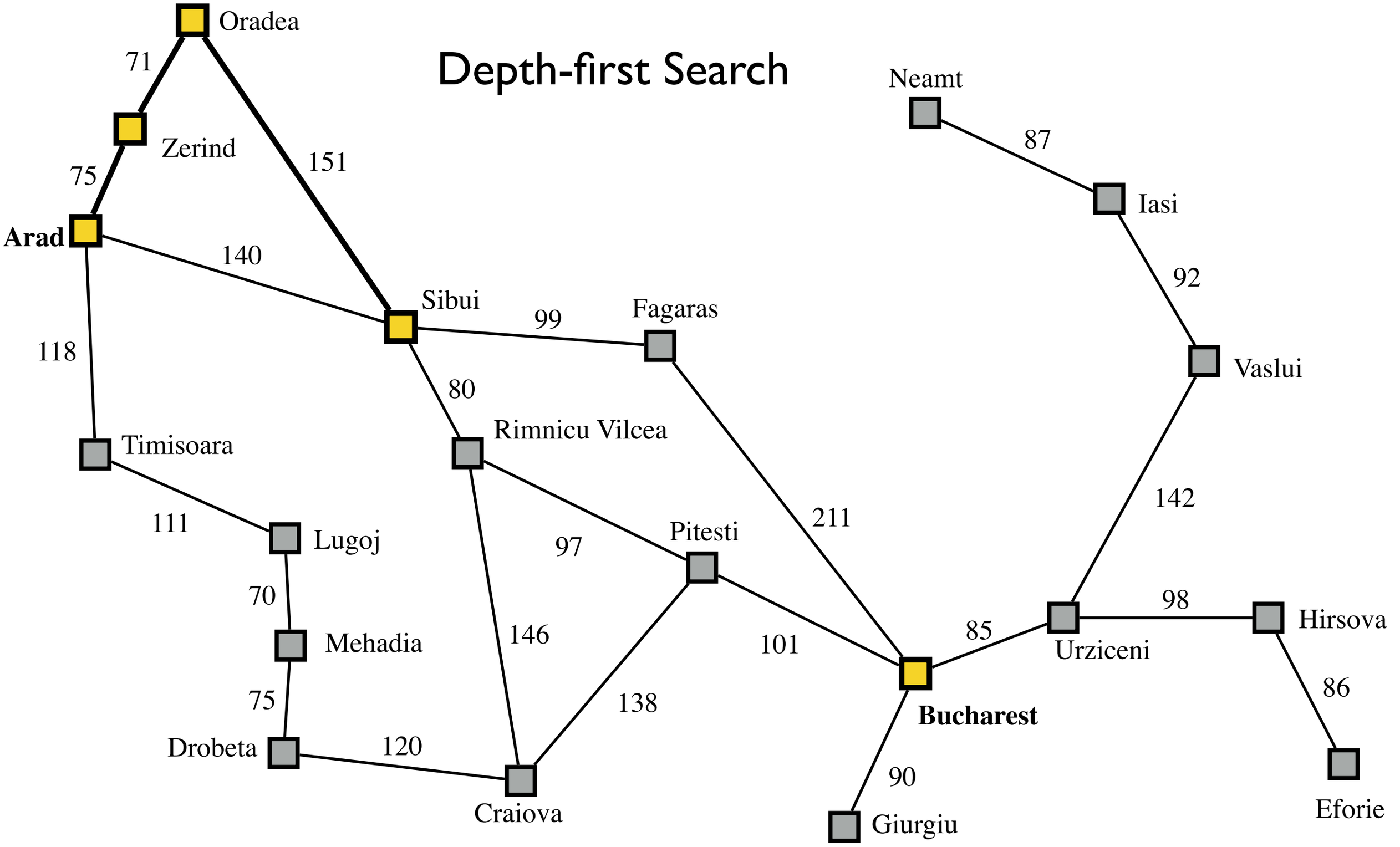
But what if we did reverse-alphabetical expansion?

Depth-first Search



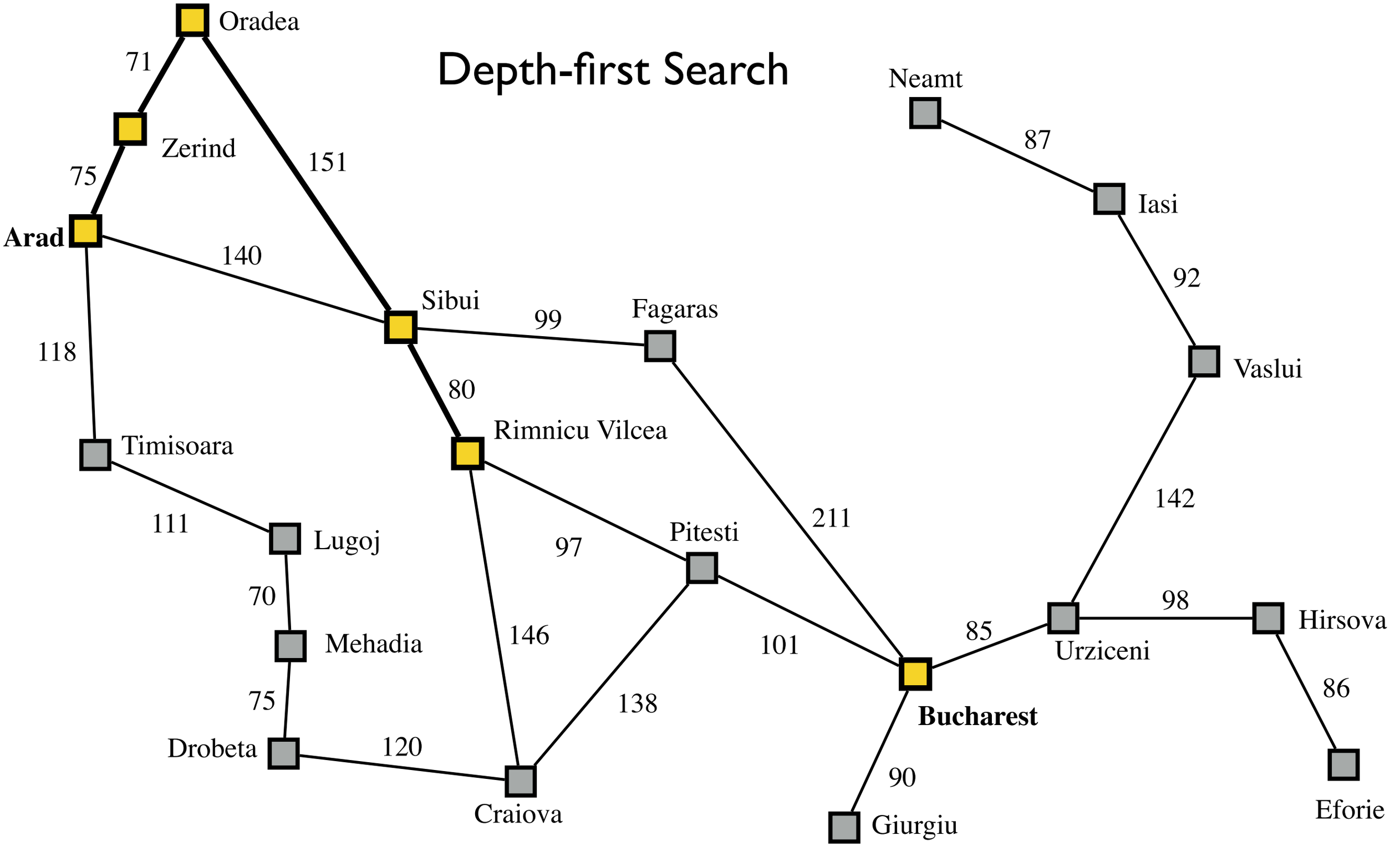
But what if we did reverse-alphabetical expansion?

Depth-first Search



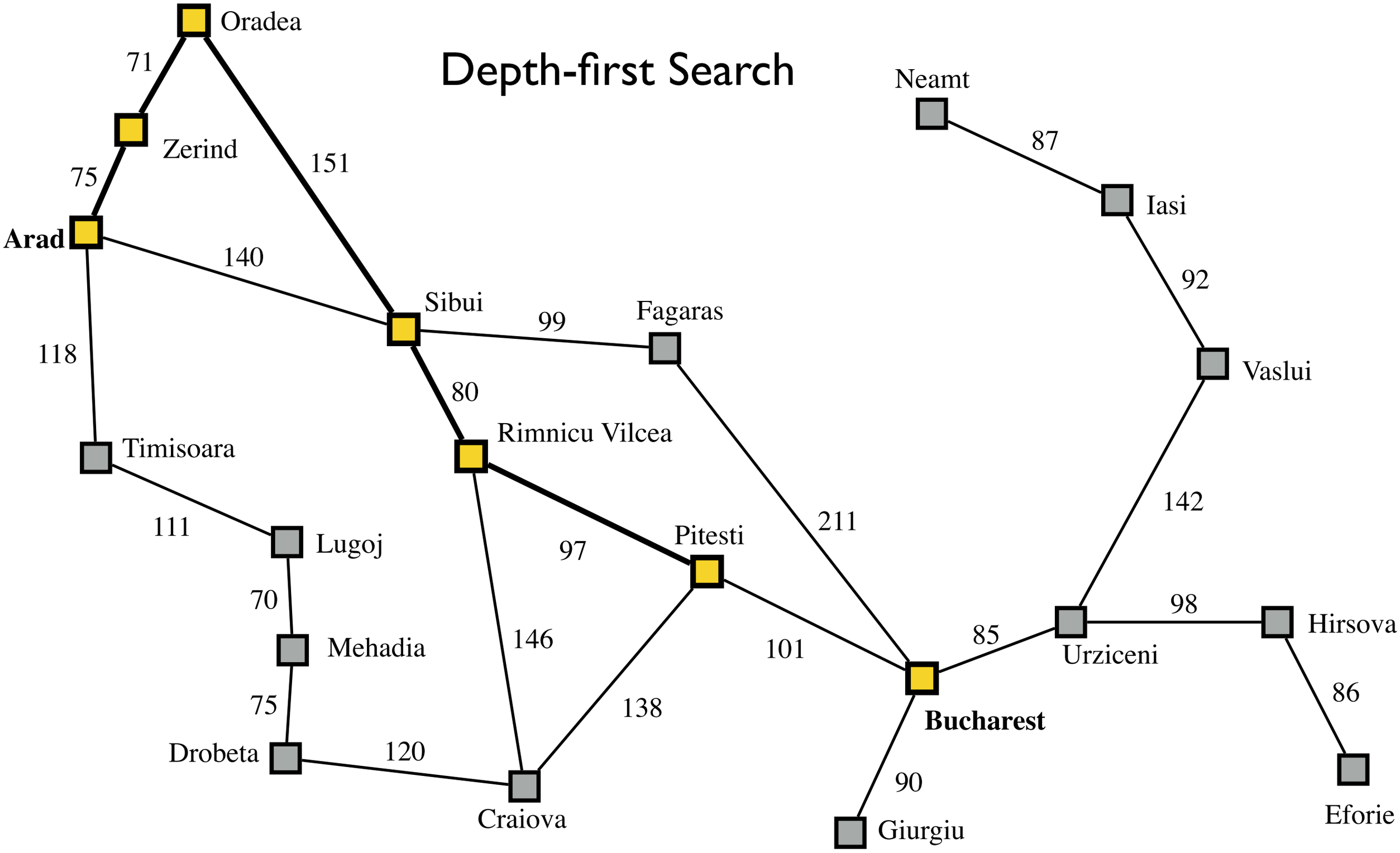
But what if we did reverse-alphabetical expansion?

Depth-first Search



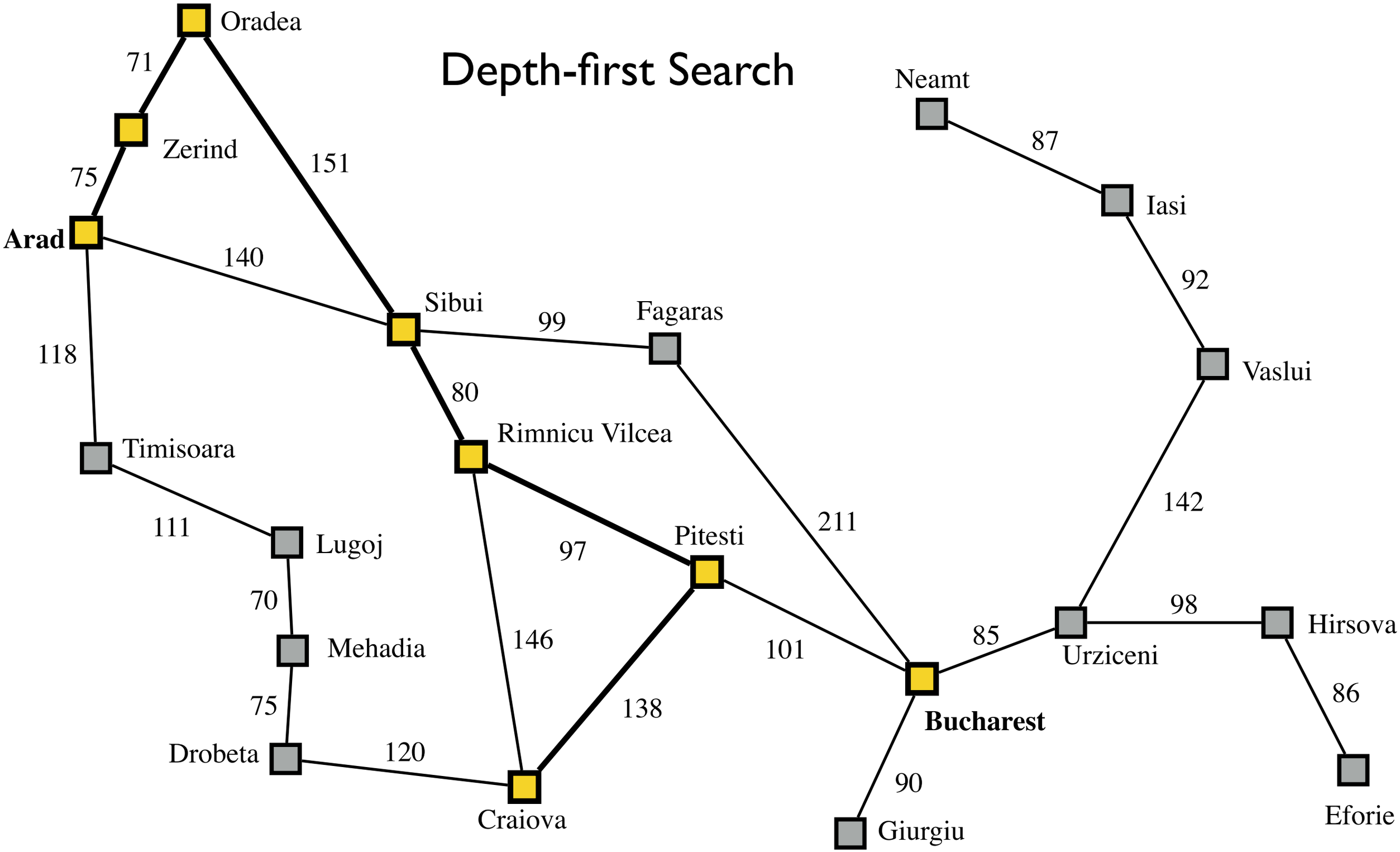
But what if we did reverse-alphabetical expansion?

Depth-first Search



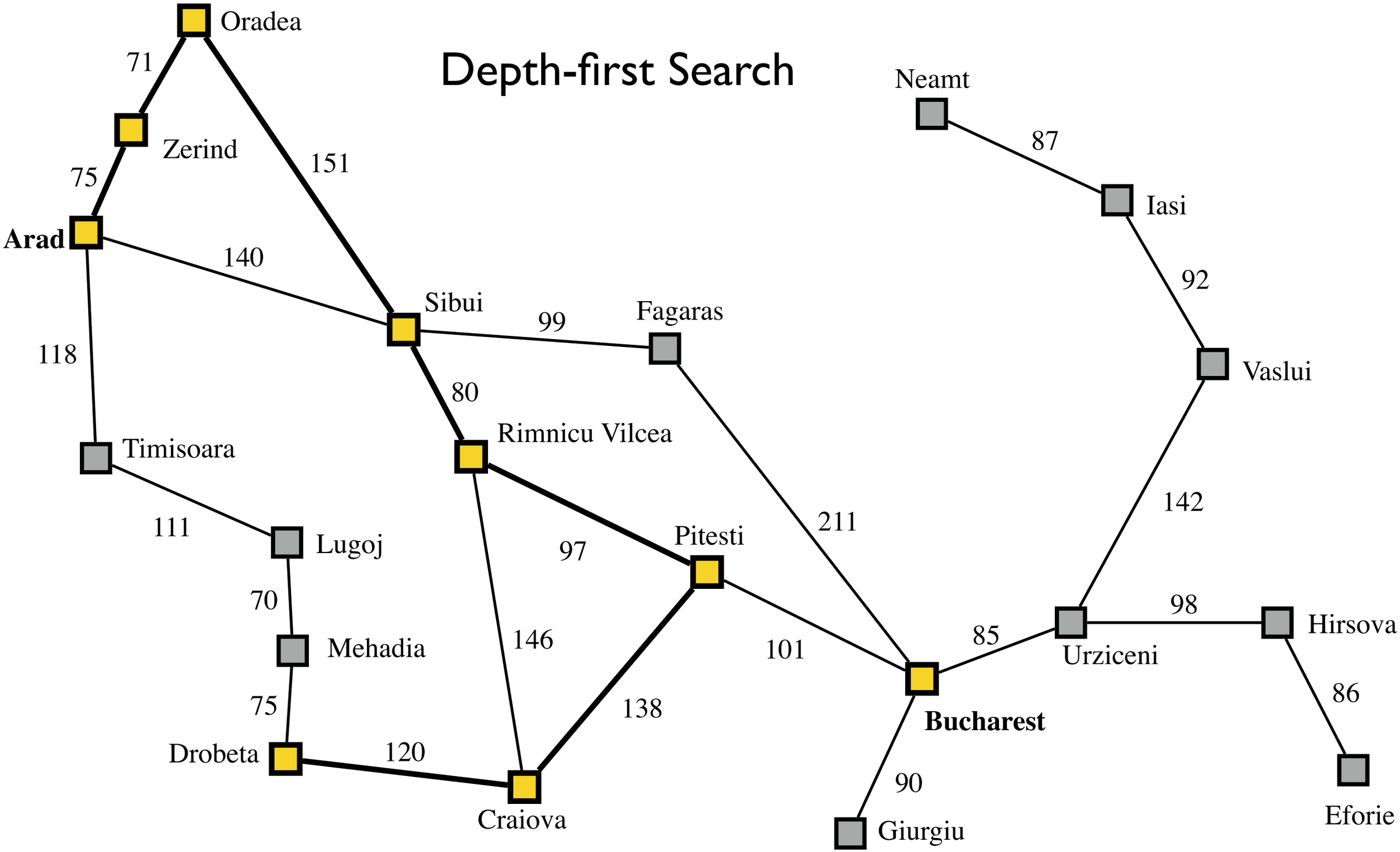
But what if we did reverse-alphabetical expansion?

Depth-first Search



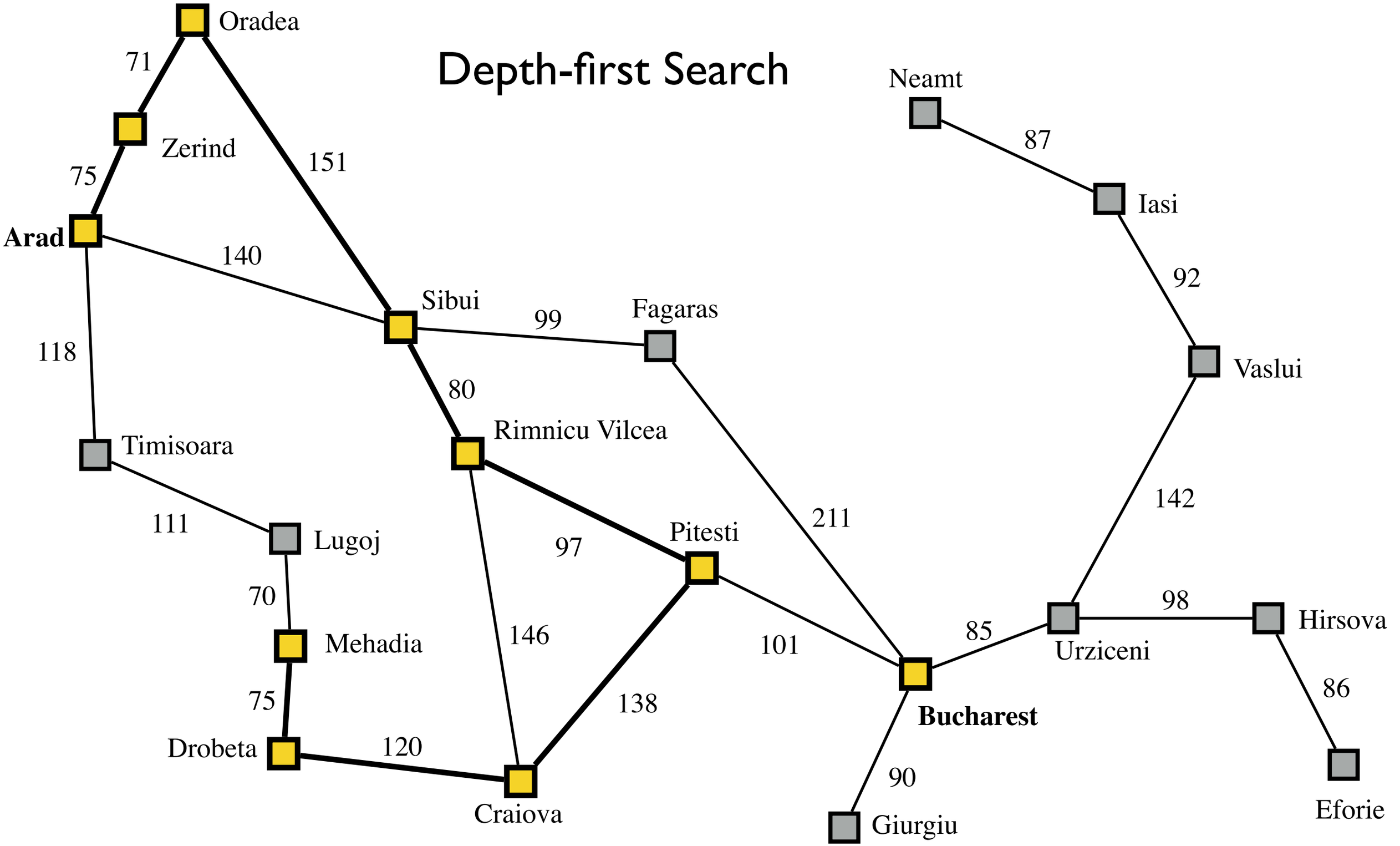
But what if we did reverse-alphabetical expansion?

Depth-first Search



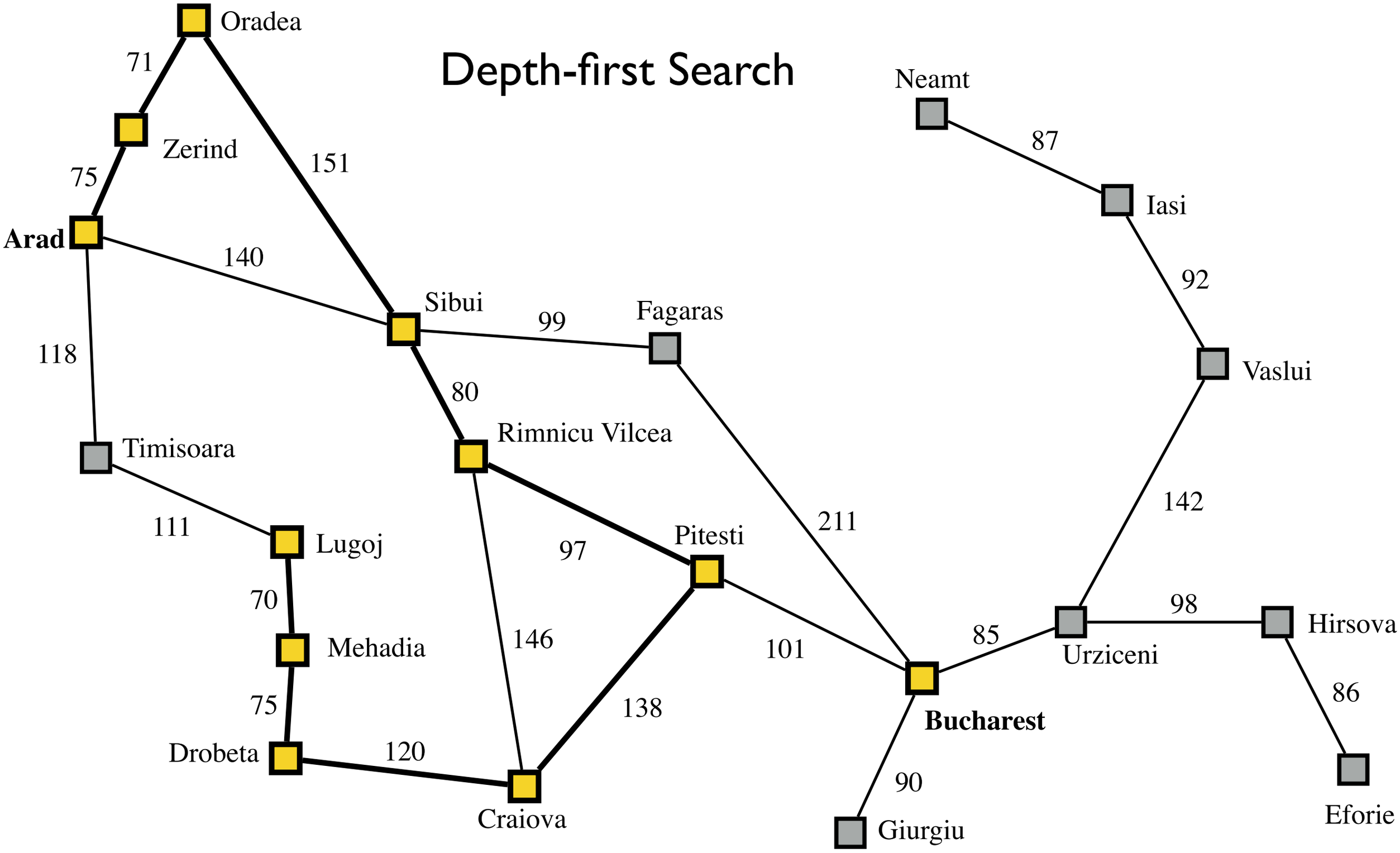
But what if we did reverse-alphabetical expansion?

Depth-first Search



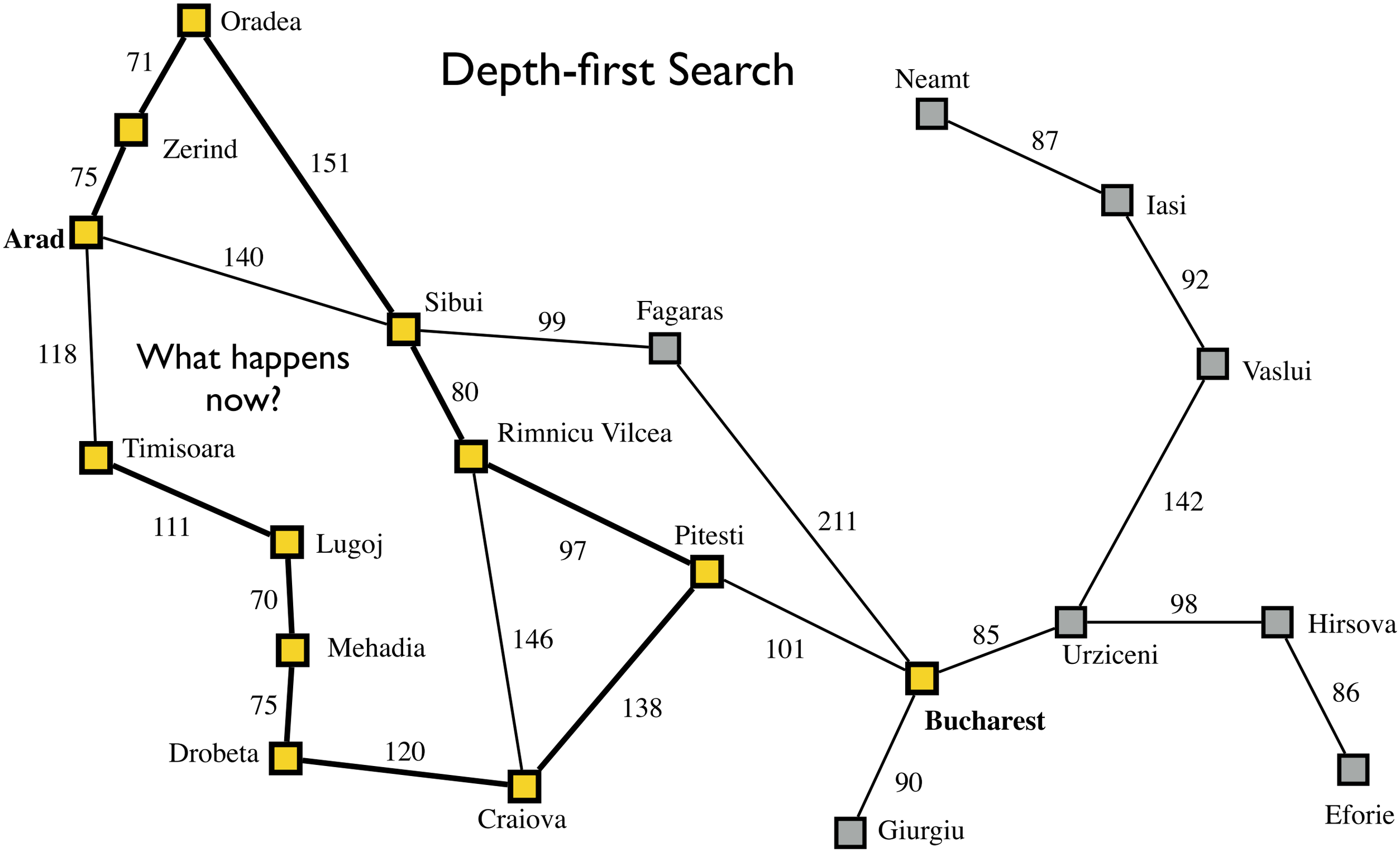
But what if we did reverse-alphabetical expansion?

Depth-first Search



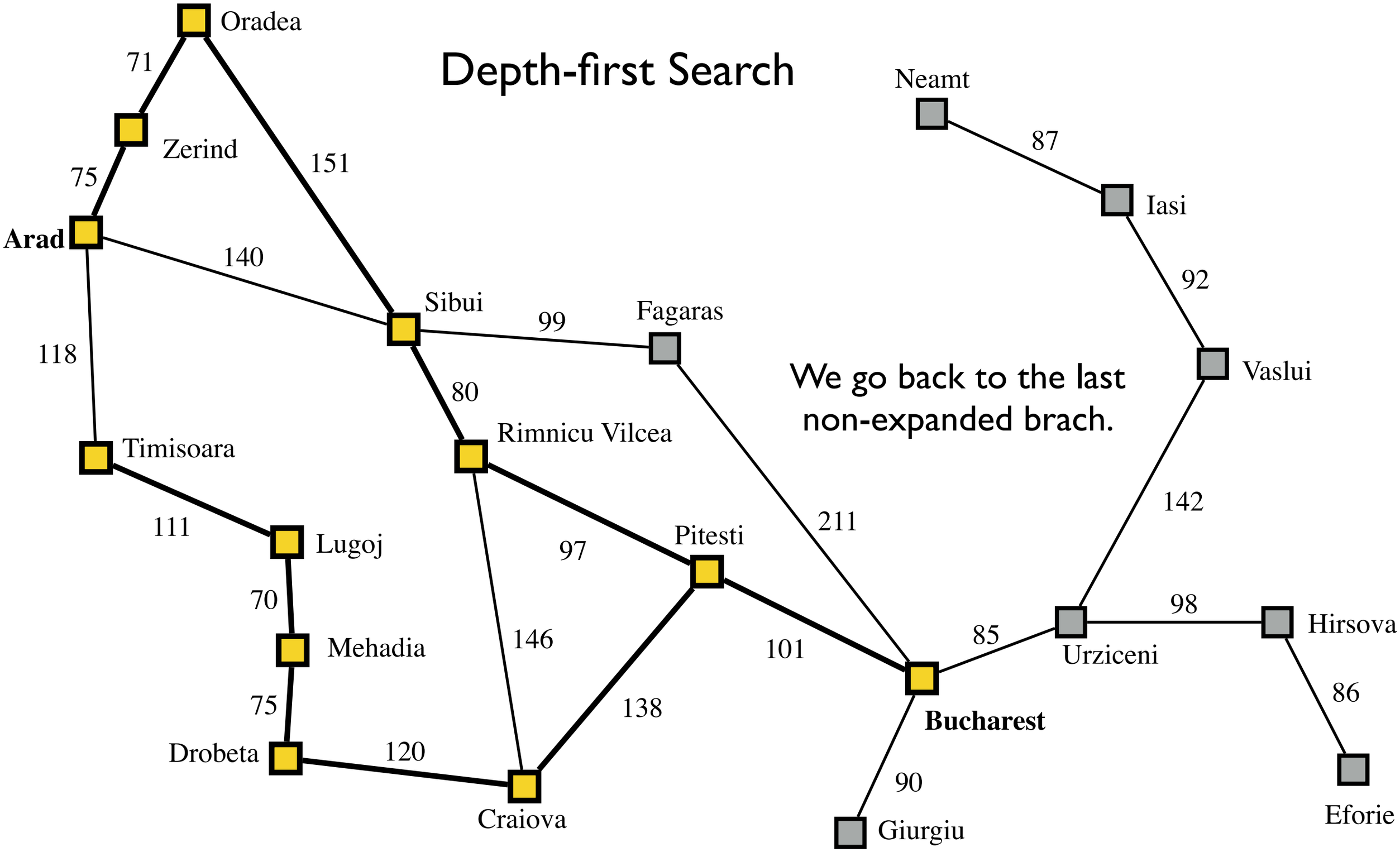
But what if we did reverse-alphabetical expansion?

Depth-first Search



But what if we did reverse-alphabetical expansion?

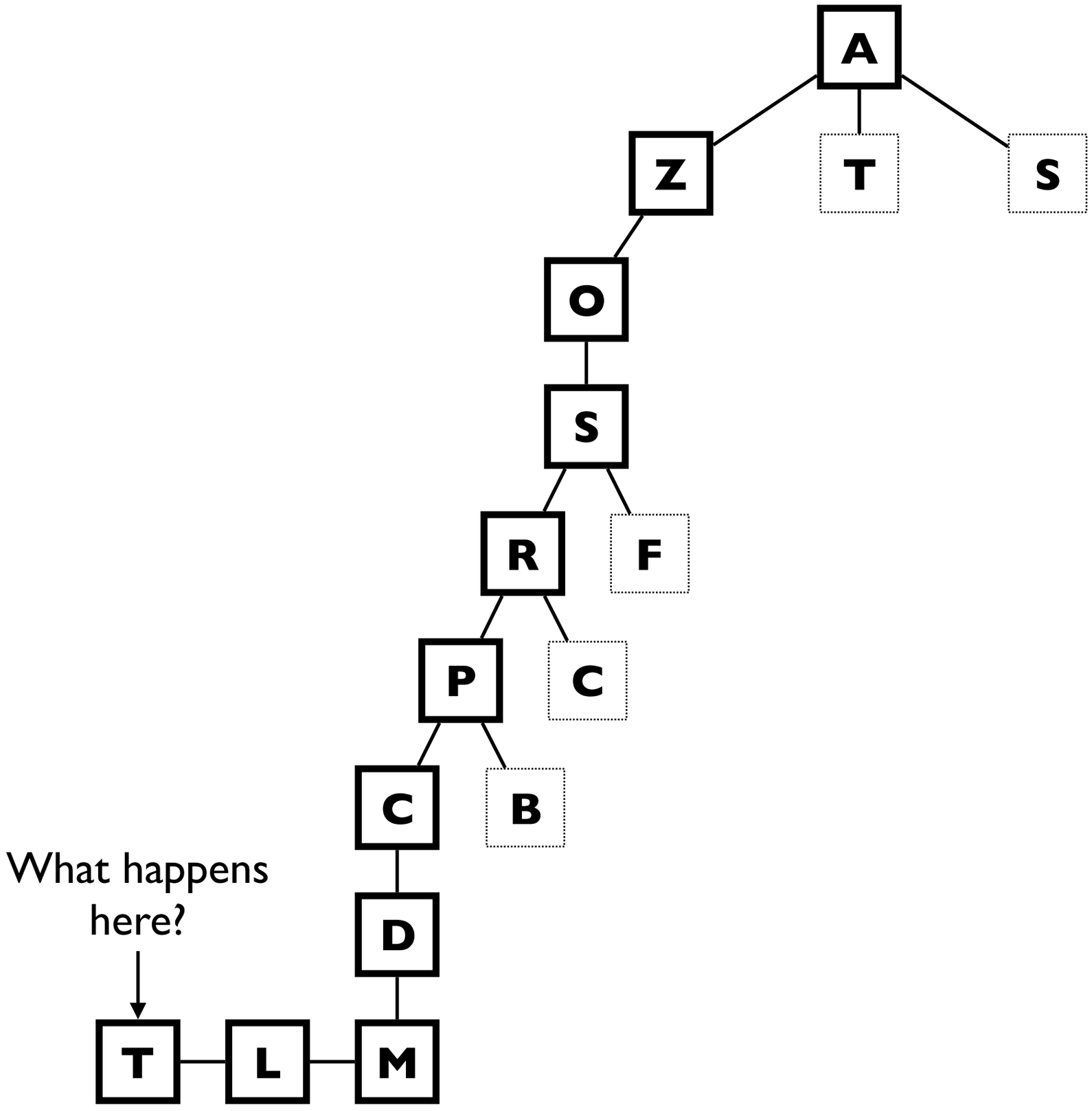
Depth-first Search



We go back to the last non-expanded brach.

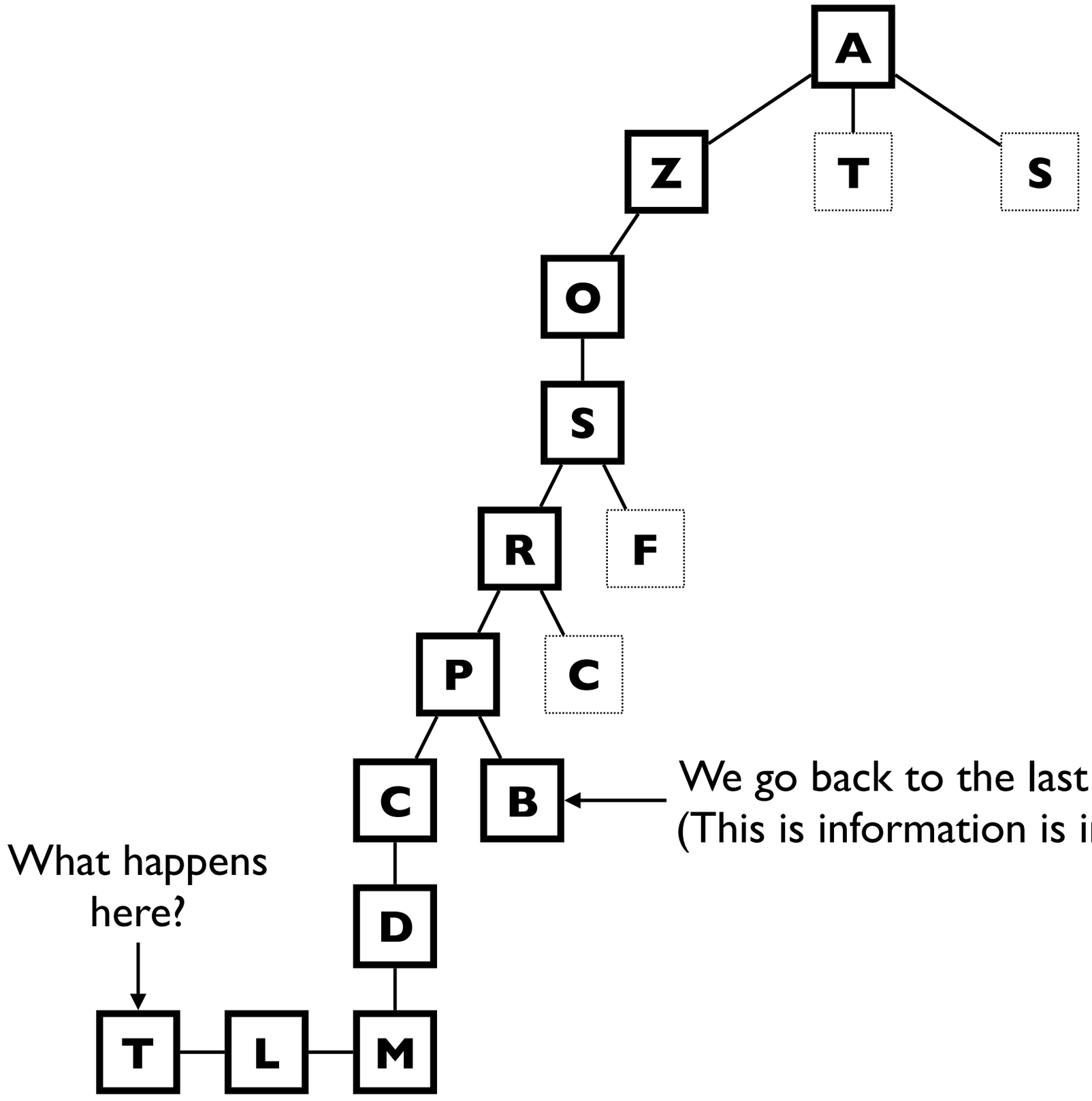
But what if we did reverse-alphabetical expansion?

Tree view of DFS (reverse-alphabetical expansion)



What happens here?

Tree view of DFS (reverse-alphabetical expansion)



We go back to the last non-expanded brach.
(This is information is in the recursion stack)

Generic search function

function Tree-Search(*problem, strategy*) **returns** solution or failure

initialize tree using initial state of *problem*

loop

if no candidates for expansion **return** failure

choose leaf node for expansion using *strategy*

if node contains goal state **return** solution

else expand node and add resulting nodes to search tree

Depth-limited search

- incorporate depth limit in depth-first search
- L = depth limit, ie nodes at depth L are not expanded
- Complete if L >= d
- Still not optimal, because DFS isn't optimal.
- Time is $O(b^L)$
- Space is O(bL)

Recursive implementation of 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
```

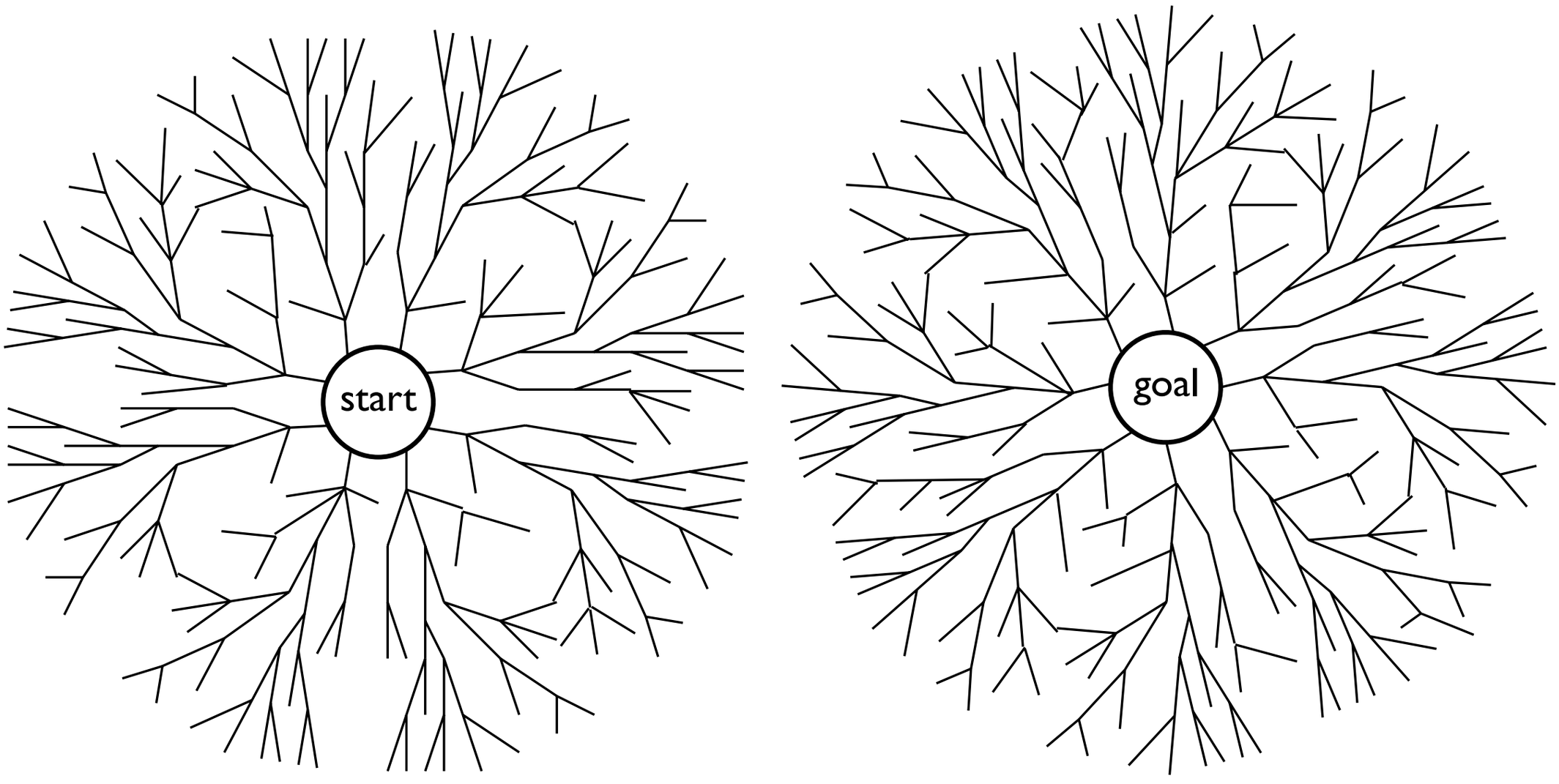
Iterative deepening search

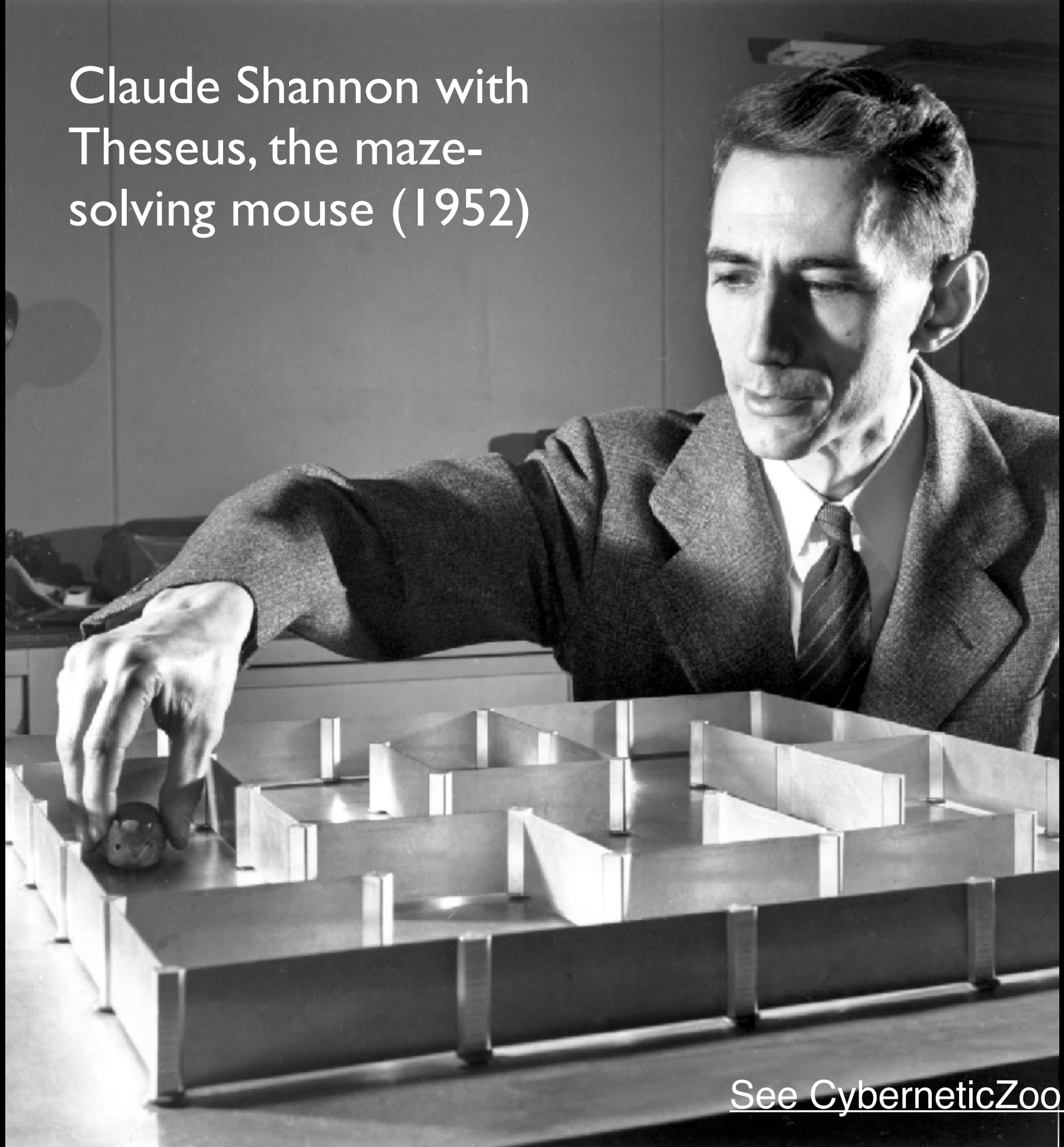
- depth-limited search with gradually increasing depth limit

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

Bi-directional search

- perform two simultaneous searches: one from start, one from goal
- stop when search paths meet
- Complexity $O(b^{d/2}) << O(b^d)$ (checking constant time with hash table)



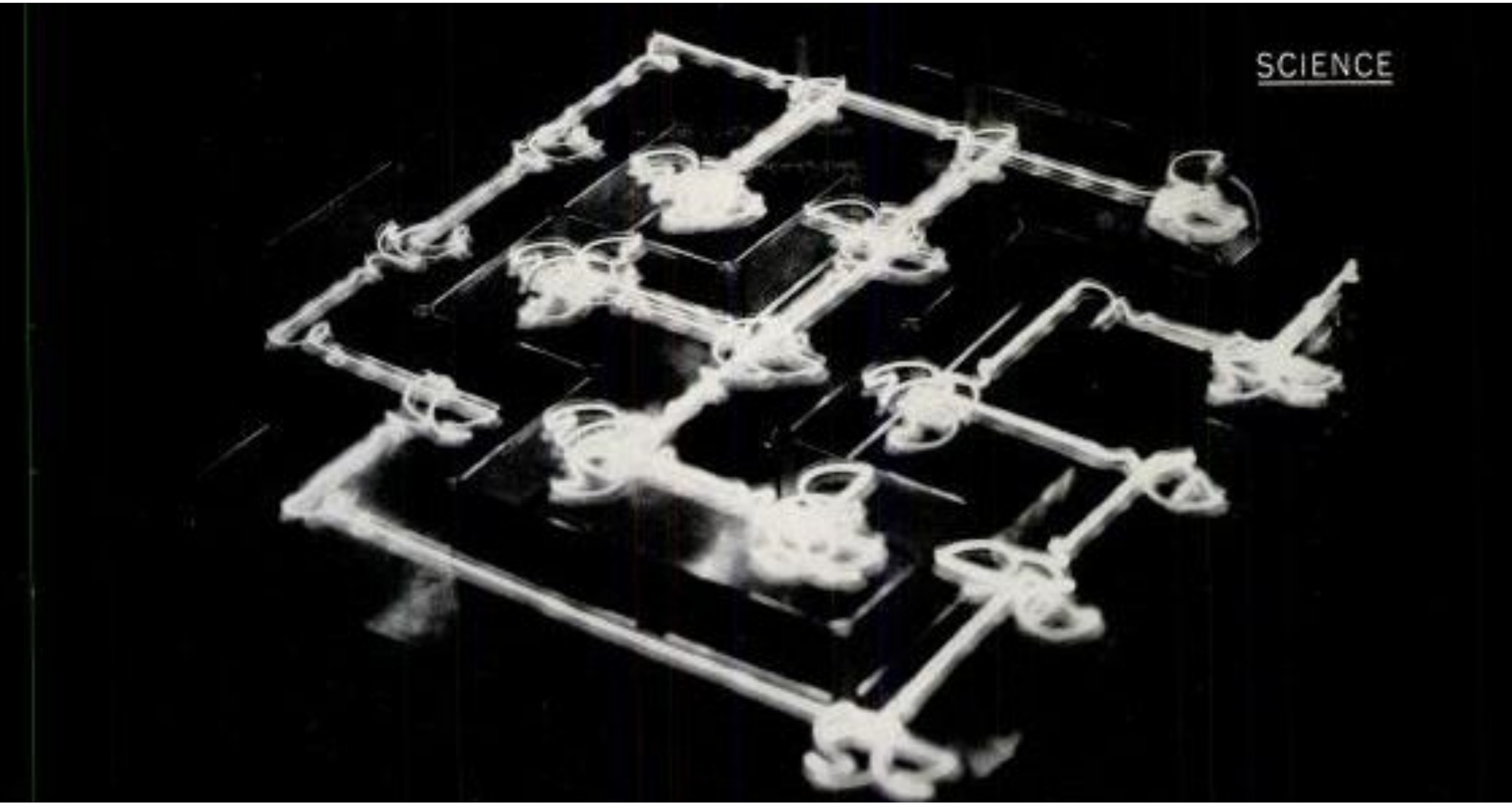


See CyberneticZoo.com article

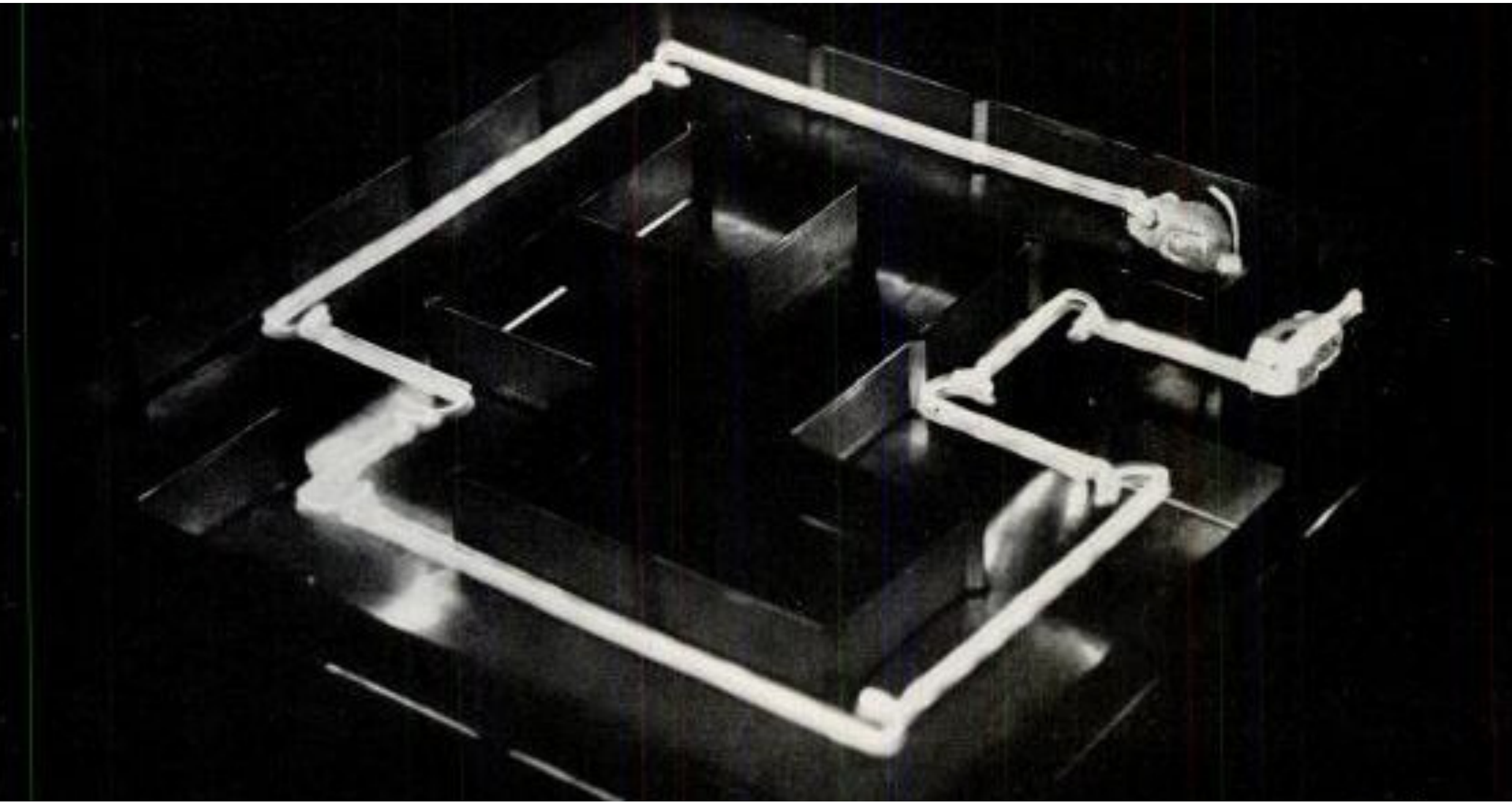
Theseus algorithm:

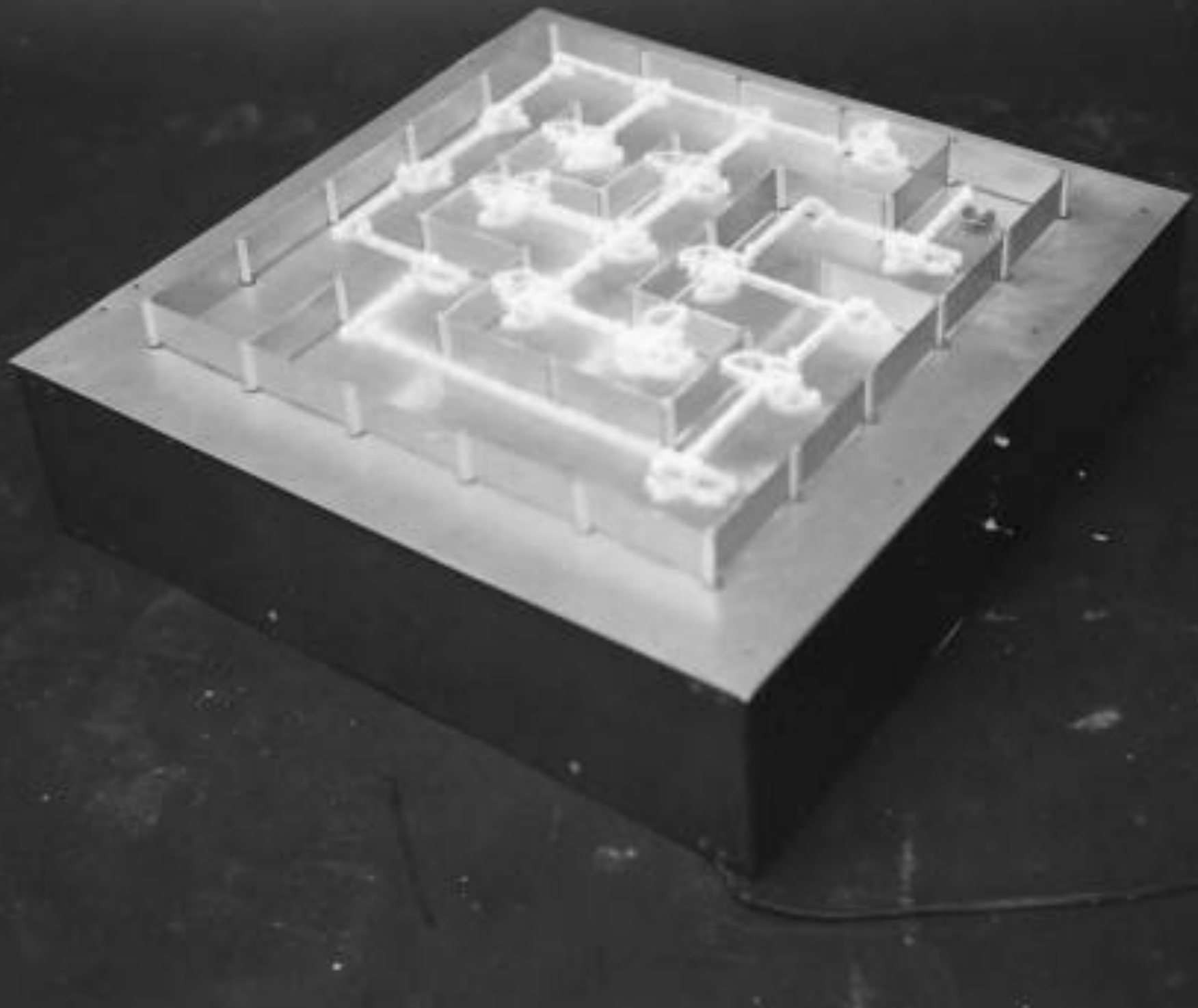
- Trémaux's algorithm:
  - unroll a ball of string to backtrace
  - mark locations visited to avoid looping
  - done when no options remain
- equivalent to depth first search
- an efficient way to find a way out of a maze
- Theseus the mouse was named after the mythical Greek hero Theseus who entered Labyrinth to kill the Minotaur
- Theseus used a ball of string to find his way back out

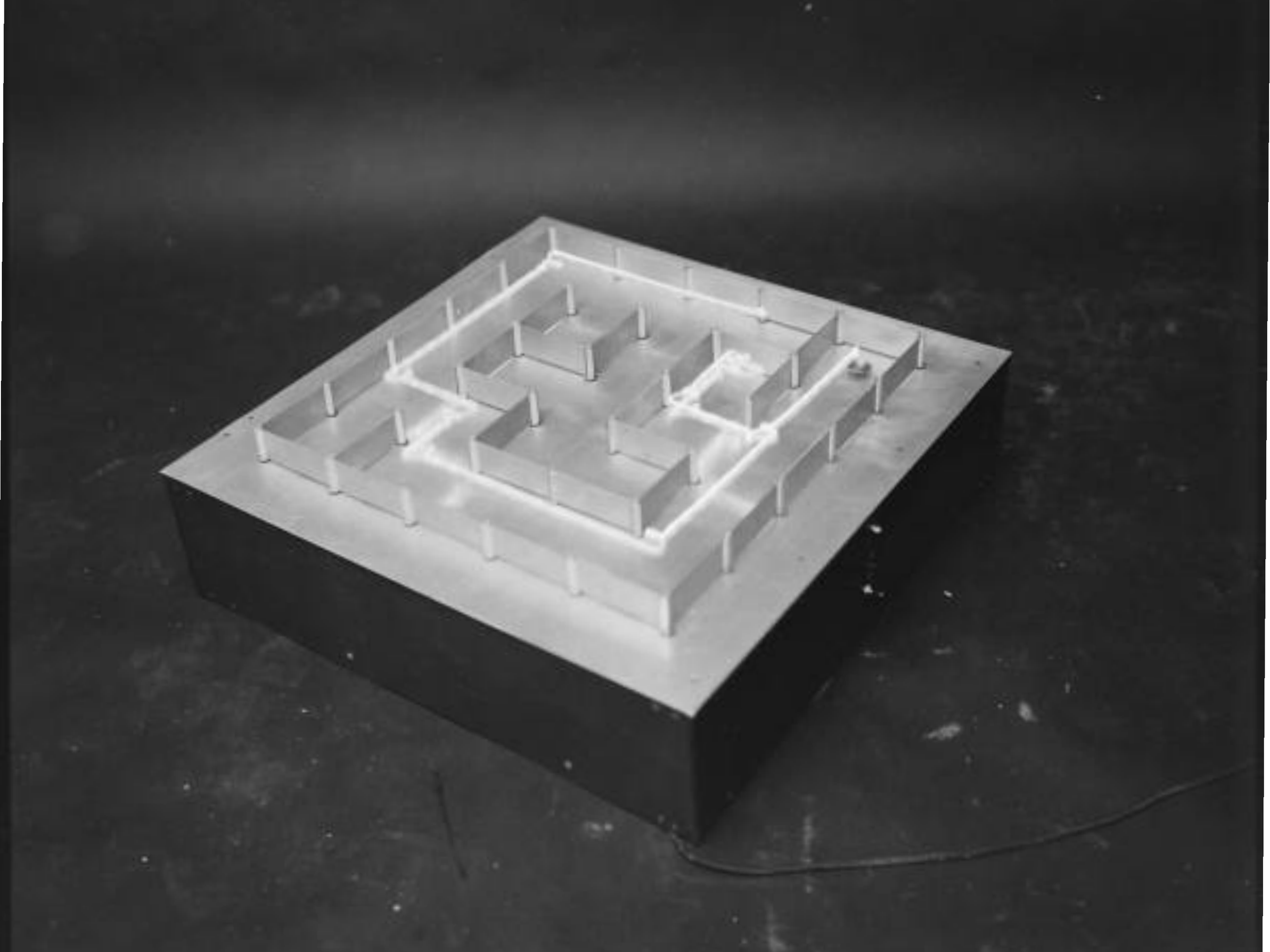
From 1952 *Life* magazine article: trace of Theseus' first pass

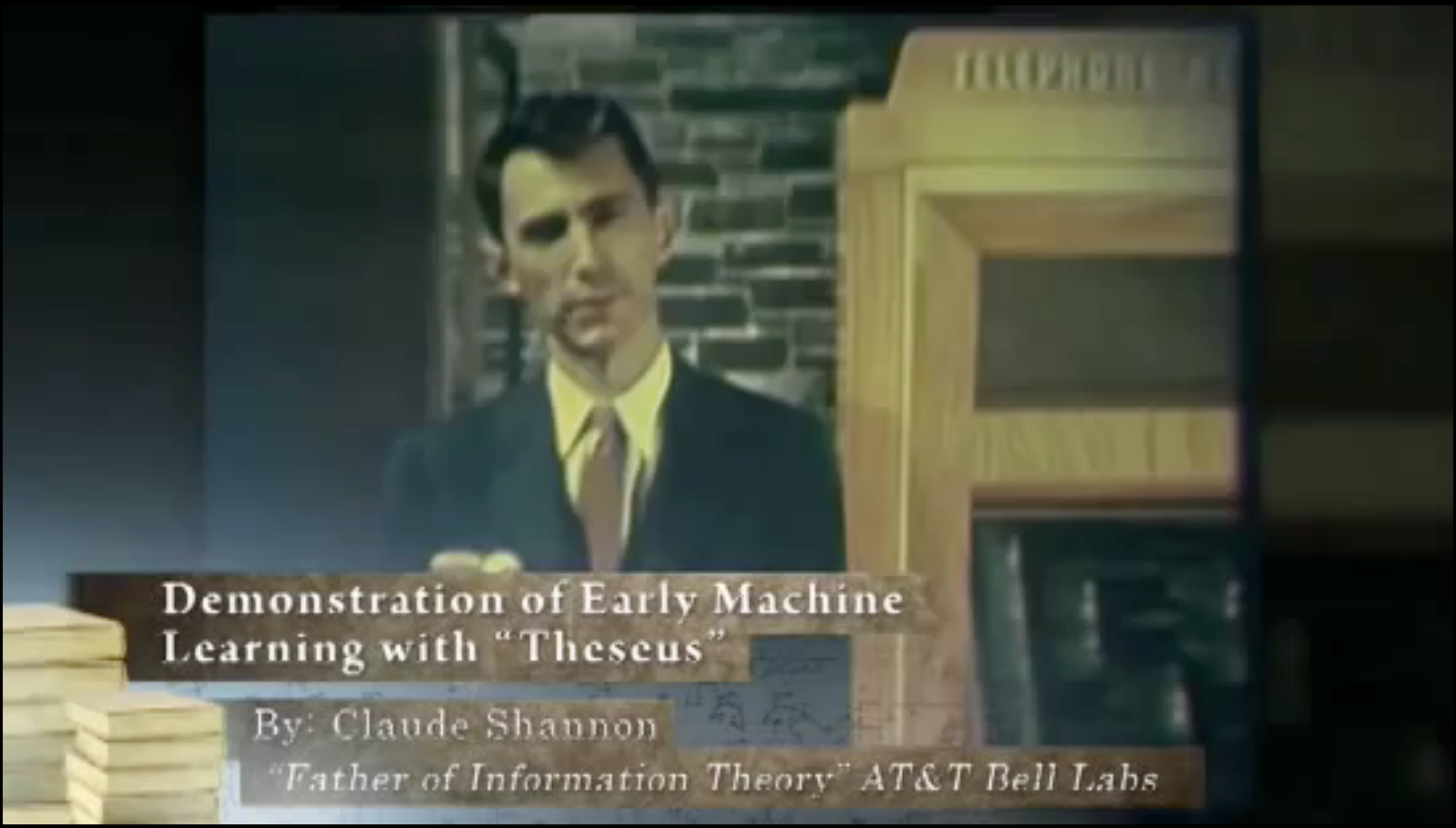


Trace of Theseus' second pass, after learning









https://www.youtube.com/watch?v=vPKkXibQXGA