Elective Course

Course Code: CS4103

Autumn 2025-26

Lecture #08

Artificial Intelligence for Data Science

Week-2: PROBLEM SOLVING BY SEARCH

Uninformed Search Techniques [Part-III] &
Introduction to Informed Search [Part-I]

Course Instructor:

Dr. Monidipa Das

Assistant Professor

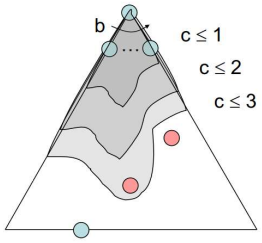
Department of Computational and Data Sciences

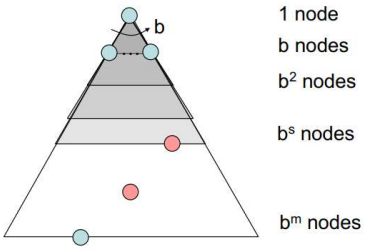
Indian Institute of Science Education and Research Kolkata, India 741246

UCS vs. BFS

2

- Instead of expanding the shallowest node (as in case of BFS), **uniform-cost search expands the node with the lowest path cost**.
- **BFS can stop as soon as it generates a goal**, whereas **UCS examines all the nodes at the goal's depth** to see if one has a lower cost.
- Thus **uniform-cost search does strictly more work** by expanding nodes at depth d unnecessarily.
- The worst-case time and space complexity of UCS is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than $O(b^d)$.
- A **test is added** in case a better path is found to a node currently on the frontier.
- UCS may get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions





BFS [Graph Search Version]


```
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)
```

Note: A BFS graph search version, where a node is tested for goal state immediate after getting generated

UCS [Graph Search version]


```
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
```

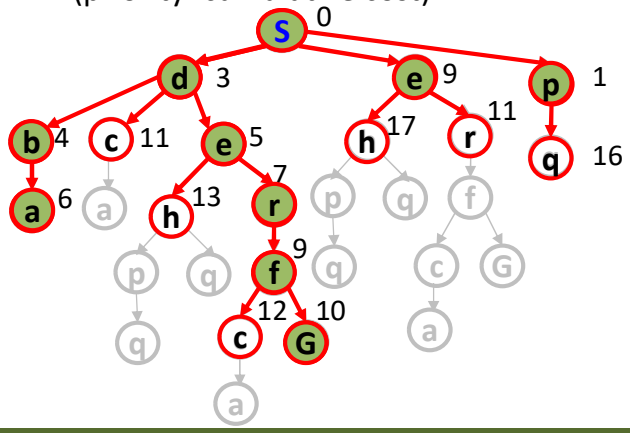
Note: In UCS, a node is tested for goal state before getting expanded. So, all the nodes that are in the same level of the goal state and that are generated before the goal state need to be expanded before founding the goal state.

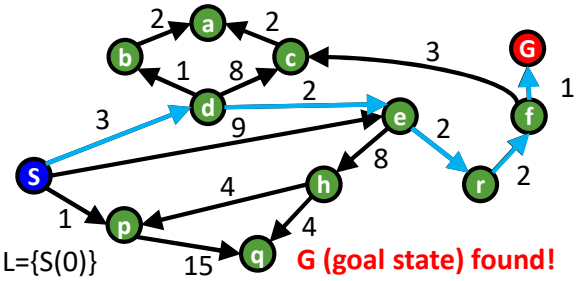
Note: In UCS, a test is added in case a better path is found to a node currently on the frontier.

Uniform-Cost Search (UCS)

[Tree-Search version]

- **Strategy:** Expand least-cost unexpanded node or the cheapest node
- **Implementation:** Fringe = **priority queue** (priority: cumulative cost)





L={S(0)}
L={p(1), d(3), e(9)}
L={d(3), e(9), q(16)}
L={b(4), e(5), e(9), c(11), q(16)}
L={e(5), a(6), e(9), c(11), q(16)}
L={a(6), r(7), e(9), c(11), h(13), q(16)}
L={r(7), e(9), c(11), h(13), q(16)}
L={e(9), f(9), c(11), h(13), q(16)}
L={f(9), c(11), r(11), h(13), q(16), h(17)}
L={G(10), c(11), r(11), c(12), h(13), q(16), h(17)}

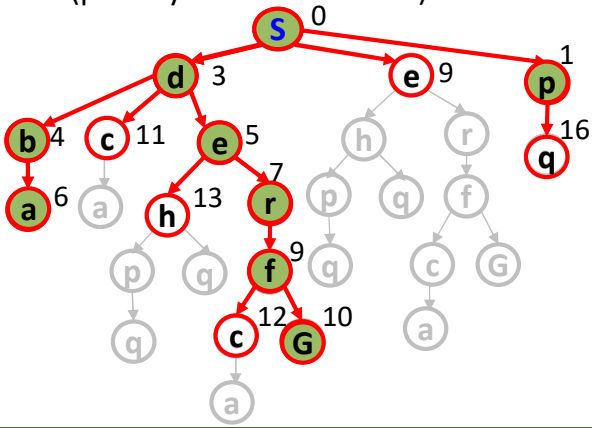
G (goal state) found!

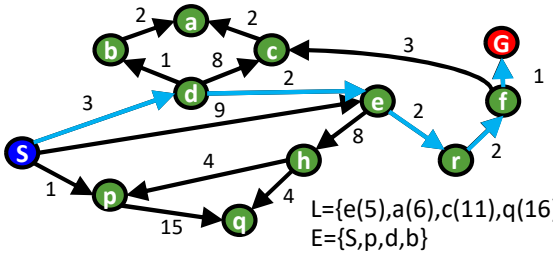
Path:
S→ d→ e→ r→ f→G

Uniform-Cost Search (UCS)

[Graph-Search version]

- **Strategy:** Expand least-cost unexpanded node or the cheapest node
- **Implementation:** Fringe = **priority queue** (priority: cumulative cost)





L={S(0)}
E={}

L={p(1), d(3), e(9)}
E={S}

L={d(3), e(9), q(16)}
E={S,p}

L={b(4), **e(5)**, c(11), q(16)}
E={S,p,d}

L={e(5),a(6),c(11),q(16)}
E={S,p,d,b}

L={a(6),r(7),c(11),h(13),q(16)}
E={S,p,d,b,e}

L={r(7),c(11),h(13),q(16)}
E={S,p,d,b,e,a}

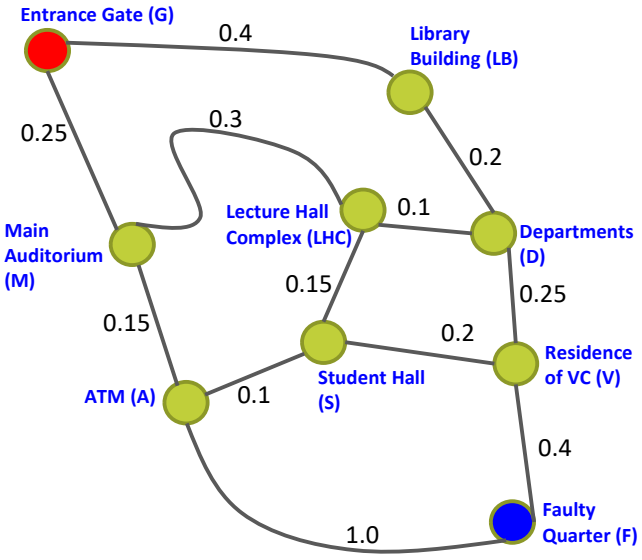
L={f(9),c(11),h(13),q(16)}
E={S,p,d,b,e,r}

L={G(10),c(11),h(13),q(16)}
E={S,p,d,b,e,r,f}

G (goal state) found!

Path: S→ d→ e→ r→ f→G **Path cost: 10**

Example Problem



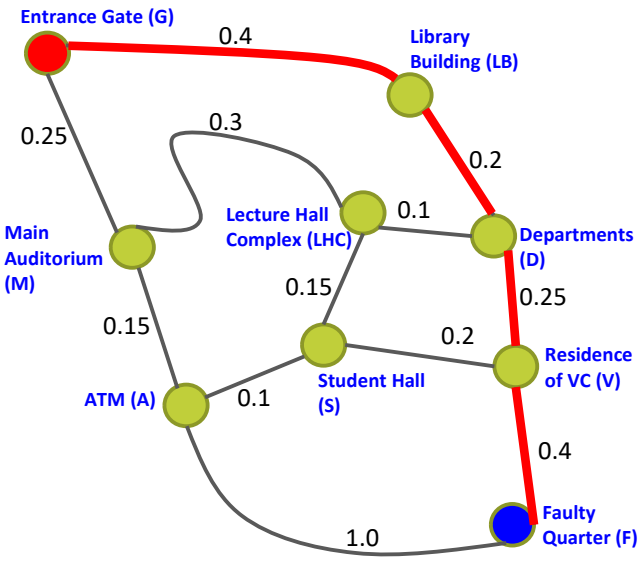
Currently in Faculty Quarter (F). Find a route to drive to Gate (G).

- **Solve using BFS** (graph search version)
- **Solve using DFS** (graph search version)
- **Solve using UCS** (graph search version)

(show the steps)

[Edge values in the given state-space indicate the road distance (in km).]

Example Problem (solution using DFS)

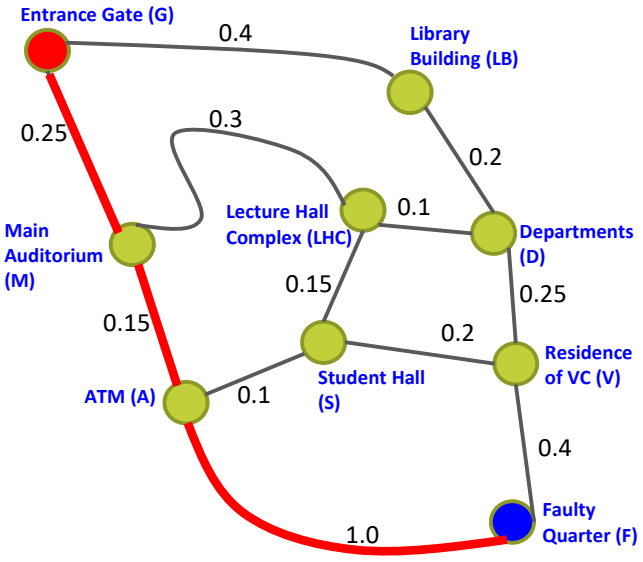


L={F}
E={}
↓
L={V, A}
E={F}
↓
L={D, S, A}
E={F, V}
↓
L={LB, LHC, S, A}
E={F, V, D}
↓

The successor of LB is the goal state (G)!

Path: F→V→D→LB→G
Path Cost: 1.25

Example Problem (solution using BFS)



L={F}
E={}

L={A,V}
E={F}

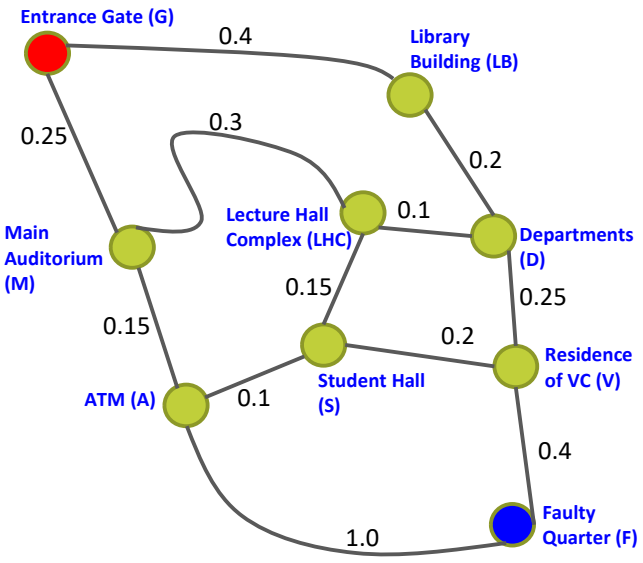
L={V, M, S}
E={F, A}

L={M, S, D}
E={F, A, V}

The successor of M is the goal state (G)!

Path: F→A→M→G
Path Cost: 1.4

Example Problem (solution using UCS)



L={F(0)}
E={}

L={V(0.4), A(1.0)}
E={F}

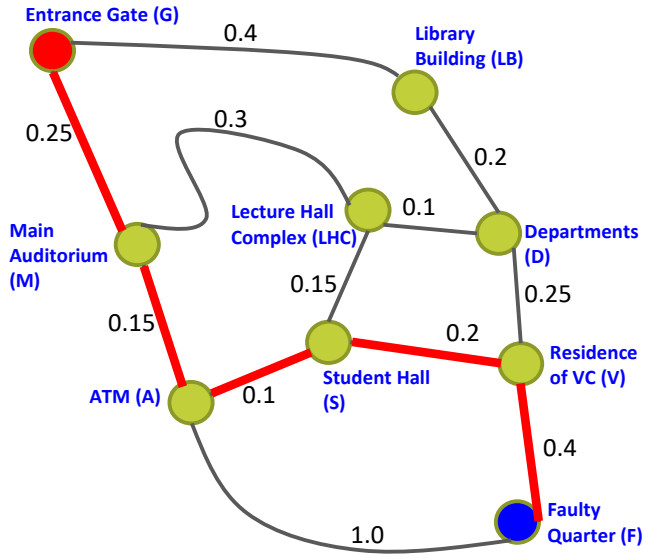
L={S(0.6), D(0.65), A(1.0)}
E={F, V}

L={D(0.65), **A(0.7)**, LHC(0.75)}
E={F, V, S}

L={A(0.7), LHC(0.75), LB(0.85)}
E={F, V, S, D}

L={LHC(0.75), LB(0.85), M(0.85)}
E={F, V, S, D, A}

Example Problem (solution using UCS) [contd.]



L={LB(0.85), M(0.85)}
E={F, V, S, D, A, LHC}

L={M(0.85), **G(1.25)**}
E={F, V, S, D, A, LHC, LB}

L={**G(1.10)**}
E={F, V, S, D, A, LHC, LB, M}

L={}
E={F, V, S, D, A, LHC, LB, M}

The goal state (G) found!

Path: F→V→S→A→M→G
Path Cost: 1.10

Home Assignment

You have a 3-Liter(L) and a 4-Liter(L) water jug. Also, you have a faucet with an unlimited amount of water. You need to get exactly 2L in 4L jug. Write a menu-driven program in Python, giving options to the user to solve the problem using BFS, DFS, and ID-DFS techniques (Graph Search versions). The program output should be the sequence of water jug states in **(x, y)** format starting from **Initial state (0, 0)** to the **Goal state (_, 2)**. The program should also show the visual representations of the steps as suggested by the user-selected search algorithm. (You may please utilize the "jugplot" function as given in the assignment dated 11-AUG-2025).

Operators

1. Fill 3L jug from faucet
2. Fill 4L jug from faucet
3. Fill 3L jug using water from 4L jug
4. Fill 4L jug using water from 3L jug
5. Empty 3L jug into 4L jug
6. Empty 4L jug into 3L jug
7. Dump 3L jug down drain
8. Dump 4L jug down drain

State representation:

(x, y)

x: Contents of 3L jug

y: Contents of 4L jug

To be done by a group of ***maximum*** four (04) students;

Submission: Submit by **18-AUG-2025** (through email with subject "CS4103: Assignment on Water Jug Problem" and cc to your group-mates, if there is any)

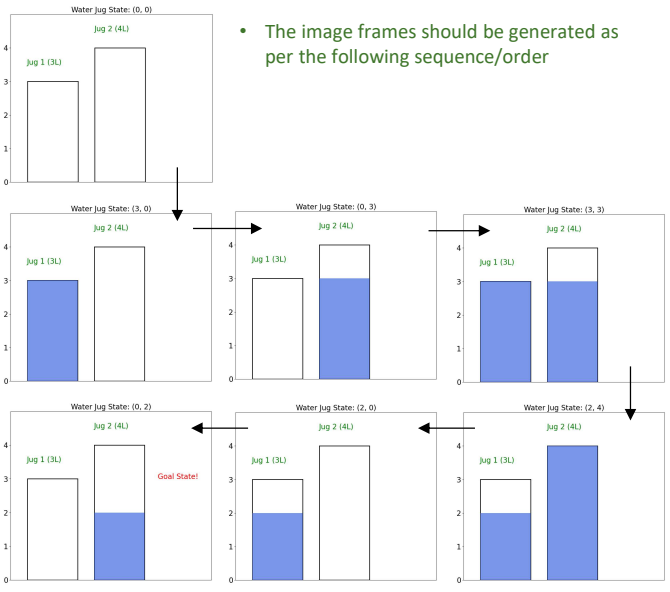
Please write the code yourself. DO NOT use AI tools to develop the code.

Note:

- You may use *list* in Python to implement the concepts of fringe/frontier and explored/closed-set and utilize *insert, append, pop or del* appropriately.
- It would be helpful to consider each node N comprising of following details:
  (state in terms of x and y values, parent node, action that generated the node, depth of the node)

Desired Output Format


```
==================
OPTIONS AVAILABLE:
==================
1. Depth First Search (DFS)
2. Breadth First Search (BFS)
3. Iterative Deepening Depth First Search (ID-DFS)
4. Exit

Please select an option from above: 2
You have selected BFS. Following are the steps to follow.

Initial State: (0,0)

Fill 3L jug from faucet : (3,0)

Empty 3L jug into 4L jug : (0,3)

Fill 3L jug from faucet : (3,3)

Fill 4L jug using water from 3L jug : (2,4)

Dump 4L jug down drain : (2,0)

Empty 3L jug into 4L jug : (0,2)

Goal Found!
```



- The image frames should be generated as per the following sequence/order

Informed/Heuristic Search

- Introduction
- Greedy Best First Search
- A* Search

Informed Search Strategies

- **Main Idea**
  - Use the knowledge of the problem domain to build an evaluation function f
  - For, every node n in the search space, $f(n)$ quantifies the desirability of expanding n in order to reach the goal
  - Then use the desirability value of the nodes in the fringe to decide which node to expand next
  - The evaluation function f is typically an imperfect measure of the goodness of the node
  - i.e., the right choice of nodes is not always the one suggested by f

Standard Assumptions on Search Spaces

- The cost of a node increases with the node's depth
- Transitions costs are non-negative and bounded below, i.e., there is a $\epsilon > 0$ such that the cost of each transition is $\geq \epsilon$
- Each node has only finitely-many successors

Best-First Search

- **Idea:** use an evaluation function estimating the desirability of each node
- **Strategy:** Always expand the most desirable unexpanded node
- **Implementation:** the fringe is a priority queue sorted in decreasing order of desirability

- **Special cases:**
  - greedy best-first search
  - A* search

- Best-first is a family of search strategies, each with a different evaluation function
- Typically, use estimates of the cost of reaching the goal and try to minimize it

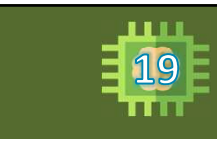
Greedy Best-First Search

- Evaluation function $h(n)$ (heuristics)
  = estimate cost of cheapest path from node n to closest goal

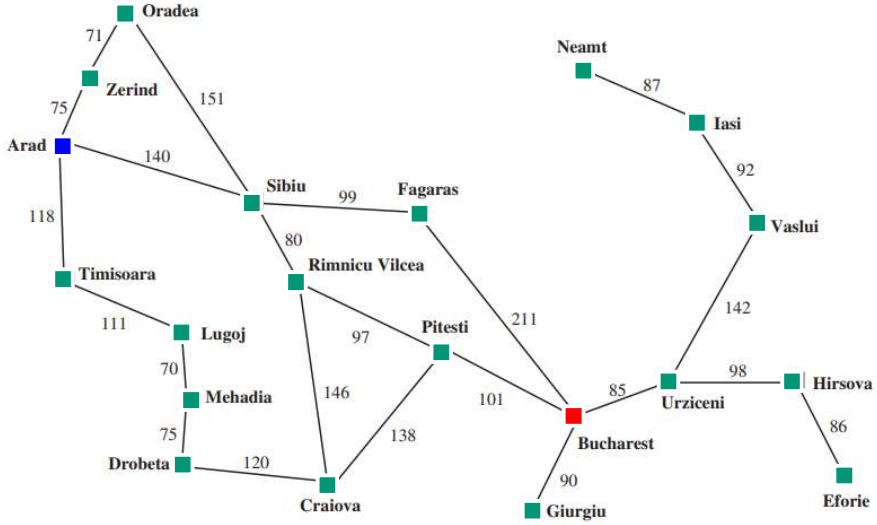
- E.g., $h_{SLD}\ (n)$ = straight-line distance from a city n to the destination city.

- **Greedy search expands the node that appears to be closest to goal**

Romania with Step Costs in Km

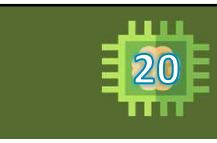


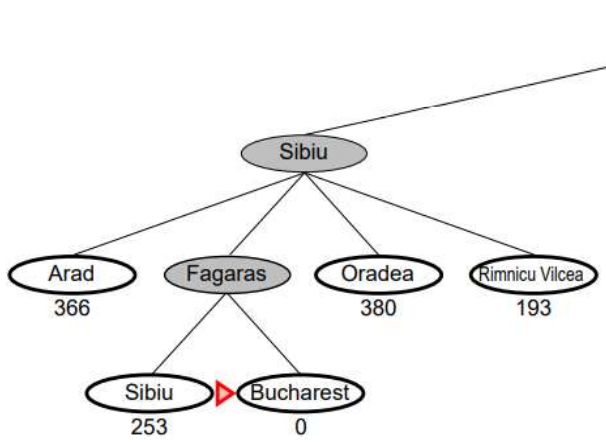
Currently in Arad. Find a route to drive to Bucharest.

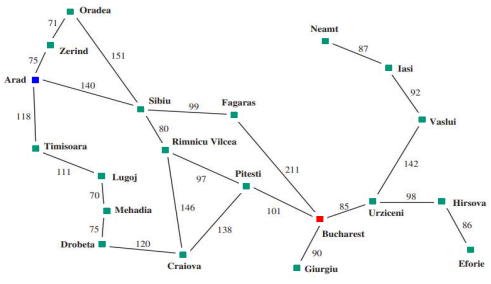


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	**176**
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Search Example







Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	**176**
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

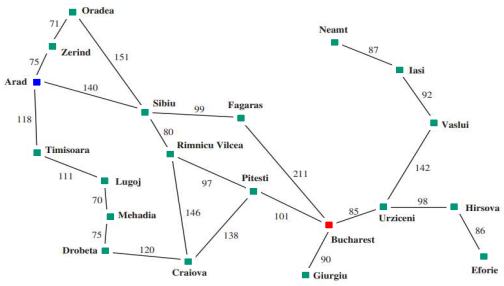
Properties of Greedy Best-First Search

Complete? Only in finite spaces with repeated-state checking
Otherwise, can get stuck in loops:
Iasi → Neamt → Iasi → Neamt →

Time complexity? $O(b^m)$ — may have to expand all nodes

Space complexity? $O(b^m)$ — may have to keep most nodes in memory

Optimal? No



A* — A Better Best-First Strategy

Greedy Best-first search

- minimizes estimated cost $h(n)$ from current node n to goal
- is informed but almost always suboptimal and incomplete

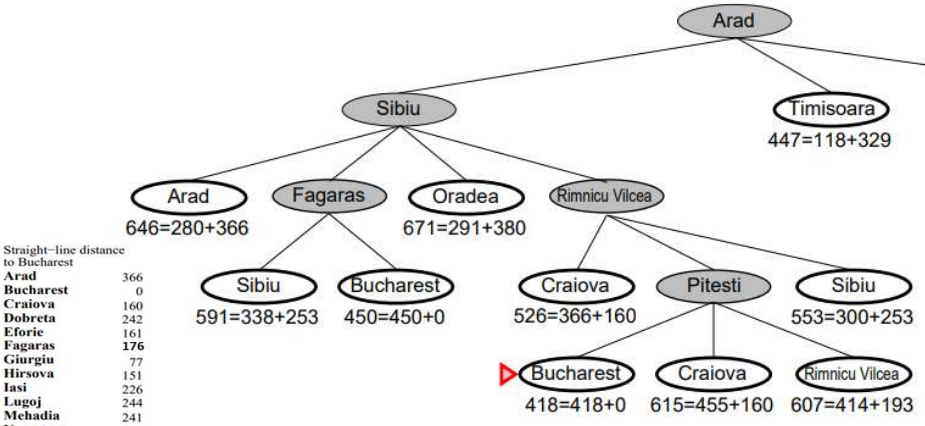
Uniform cost search

- minimizes actual cost $g(n)$ to current node n
- is, in most cases, optimal and complete but uninformed

A* search

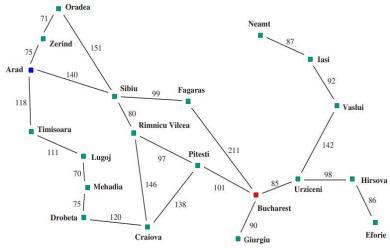
- combines the two by minimizing $f(n) = g(n) + h(n)$
- is, *under reasonable assumptions*, optimal and complete, and also informed

A* Search Example



Straight-line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
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



Questions?