Problem Solving by Searching

Asmaa Elbadrawy
PhD, Lecturer
IFT Program, ASU

Arizona State University

A **goal-based agent** acts to reach a specific goal.

If reaching the goal requires
a sequence of actions,
the agent needs to find
a path to the goal. This is
called a problem-solving agent.

Example: Path Finding

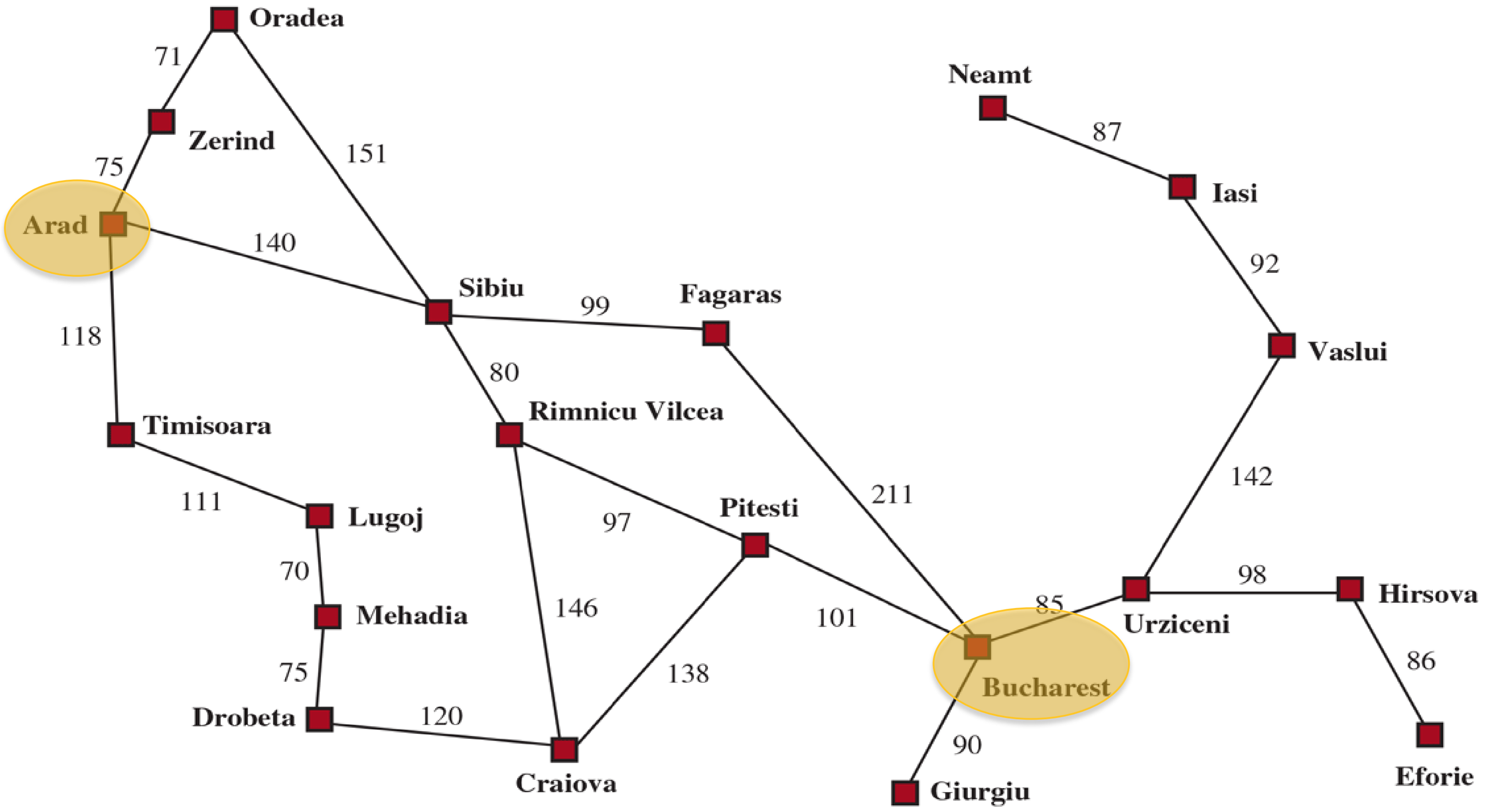

Fig 3.1, Russell & Norvig's Textbook

Find a path from Arad to Bucharest.

Similar to finding flights with multiple legs, or path finding by GPS Apps.

Example: Path Finding

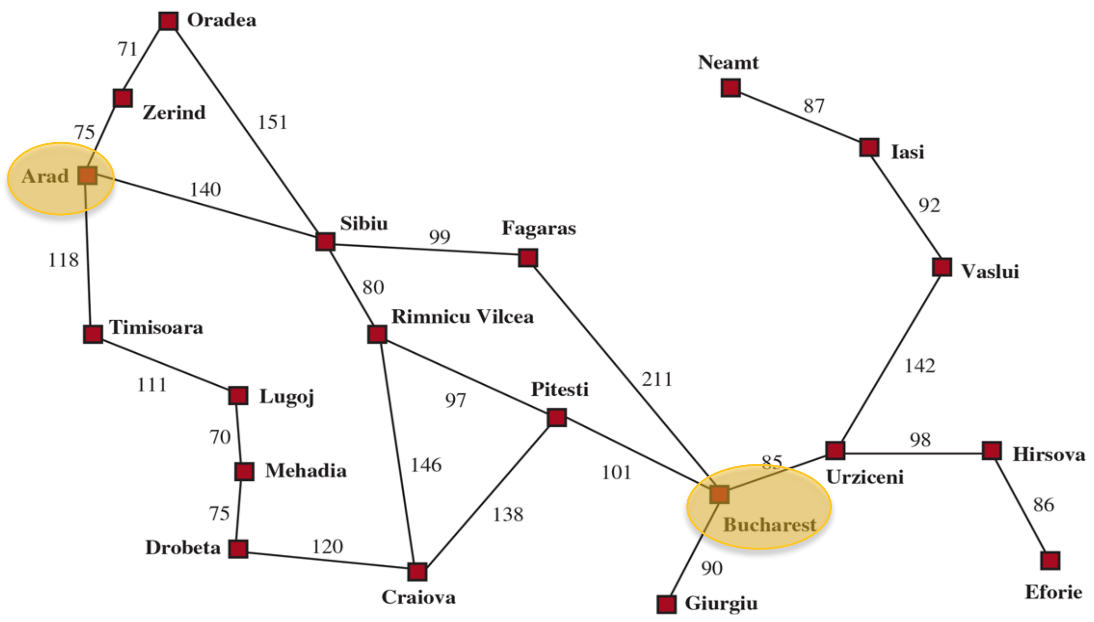

Fig 3.1, Russell & Norvig's Textbook

Since the environment is fully observable, deterministic, and known, the solution is a fixed sequence of actions.

A problem in which we try to find a sequence of actions is called a search problem.

A State Space

Initial State

Goal State(s)

Actions

Transition Model

Action-Cost Function

Defining the Search Problem

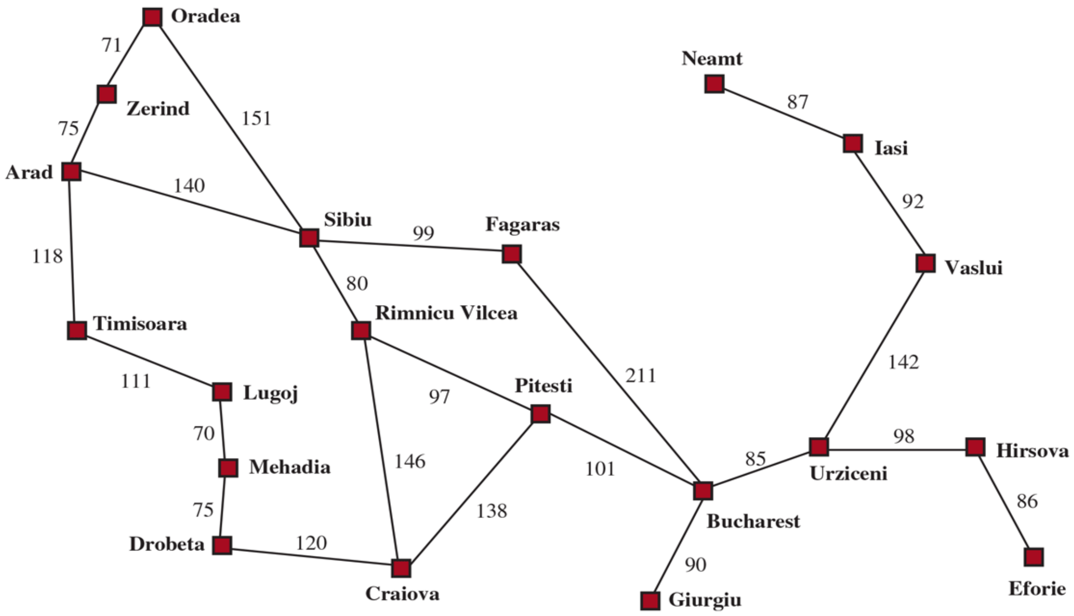

Fig 3.1, Russell & Norvig's Textbook

A State Space

Initial State

Goal State(s)

Actions

Transition Model

Action-Cost Function

Defining the Search Problem

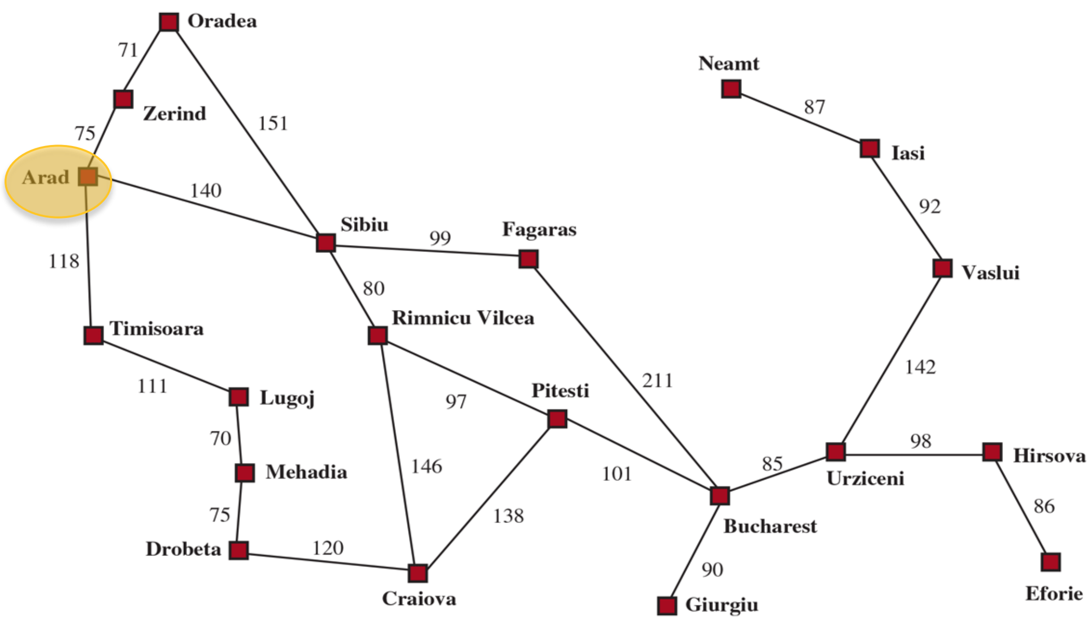

Fig 3.1, Russell & Norvig's Textbook

A State Space

Initial State

Goal State(s)

Actions

Transition Model

Action-Cost Function

Defining the Search Problem

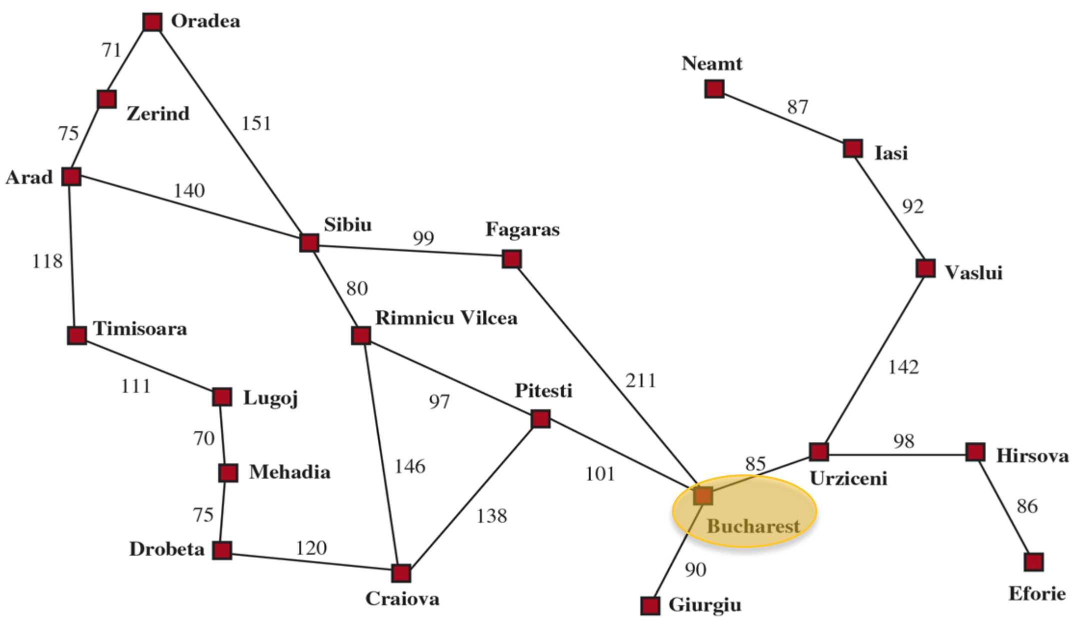

Fig 3.1, Russell & Norvig's Textbook

A State Space

Initial State

Goal State(s)

Actions

Transition Model

Action-Cost Function

Defining the Search Problem

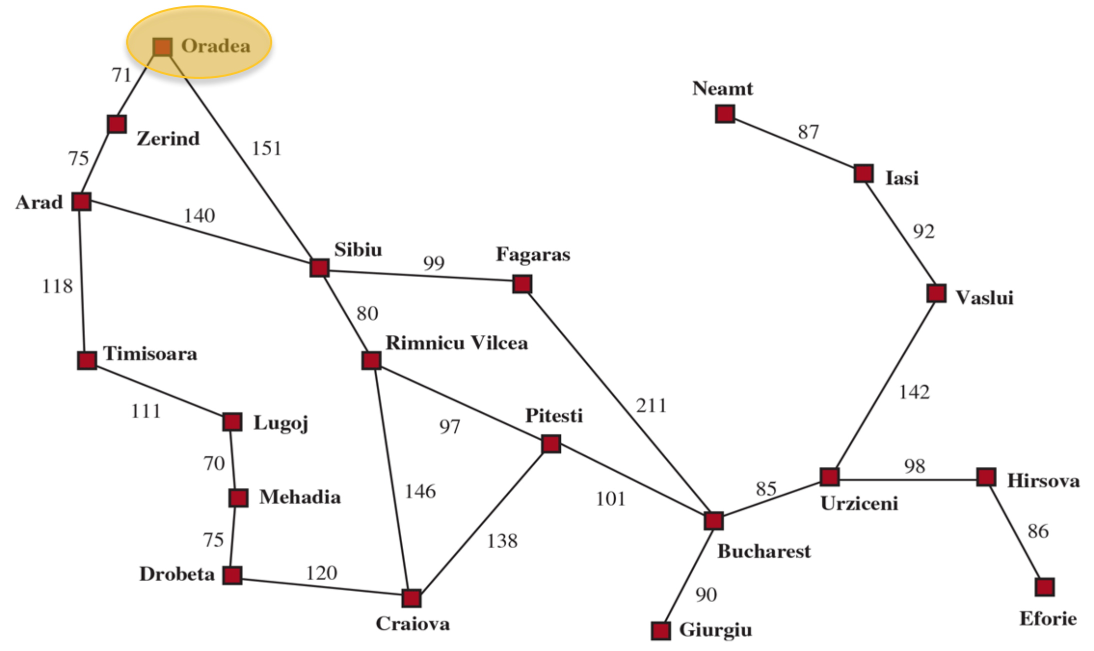

Fig 3.1, Russell & Norvig's Textbook

```
s = Oradea
ACTIONS(s) = {Zerind,Sibiu}
```

A State Space

Initial State

Goal State(s)

Actions

Transition Model

Action-Cost Function

Defining the Search Problem

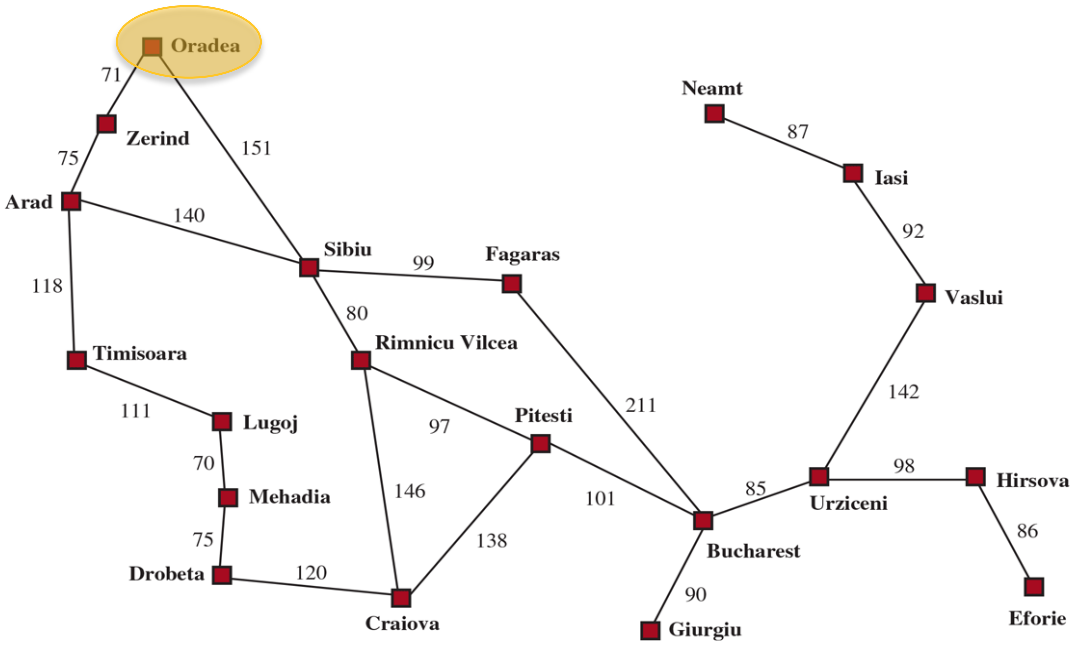

Fig 3.1, Russell & Norvig's Textbook

```
s = Oradea
ACTIONS(s) = {ToZerind,ToSibiu}
```

A State Space

Initial State

Goal State(s)

Actions

Transition Model

Action-Cost Function

Defining the Search Problem

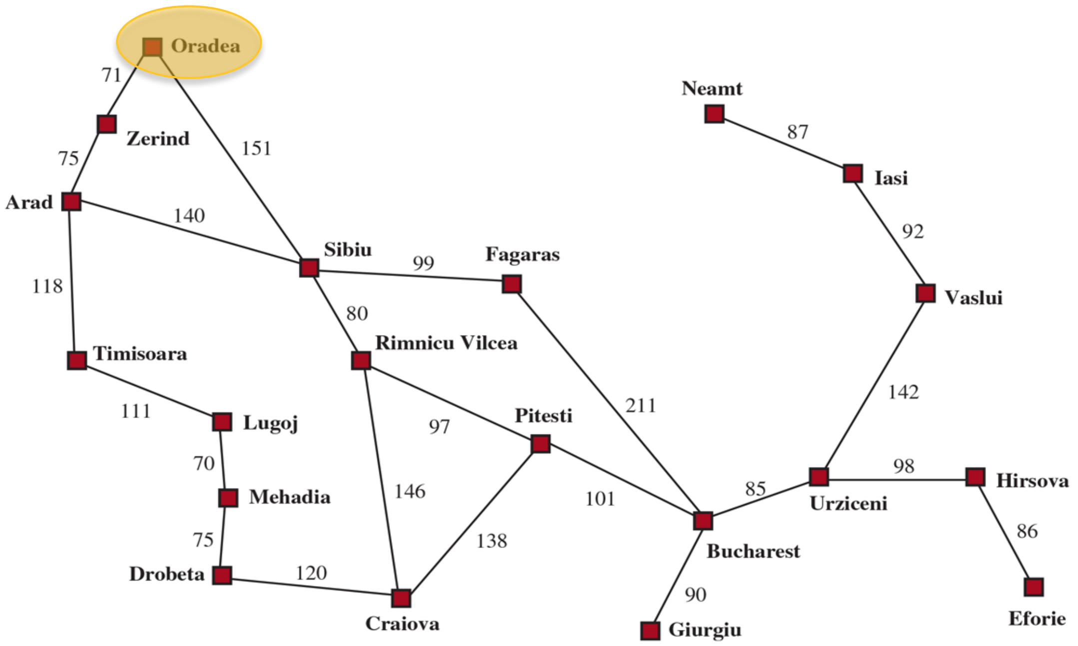

Fig 3.1, Russell & Norvig's Textbook

```
s = Oradea, a = ToZerind
Result(s,a) = {Oradea,ToZerind}
            = Zerind
```

A State Space

Initial State

Goal State(s)

Actions

Transition Model

Action-Cost Function

Defining the Search Problem

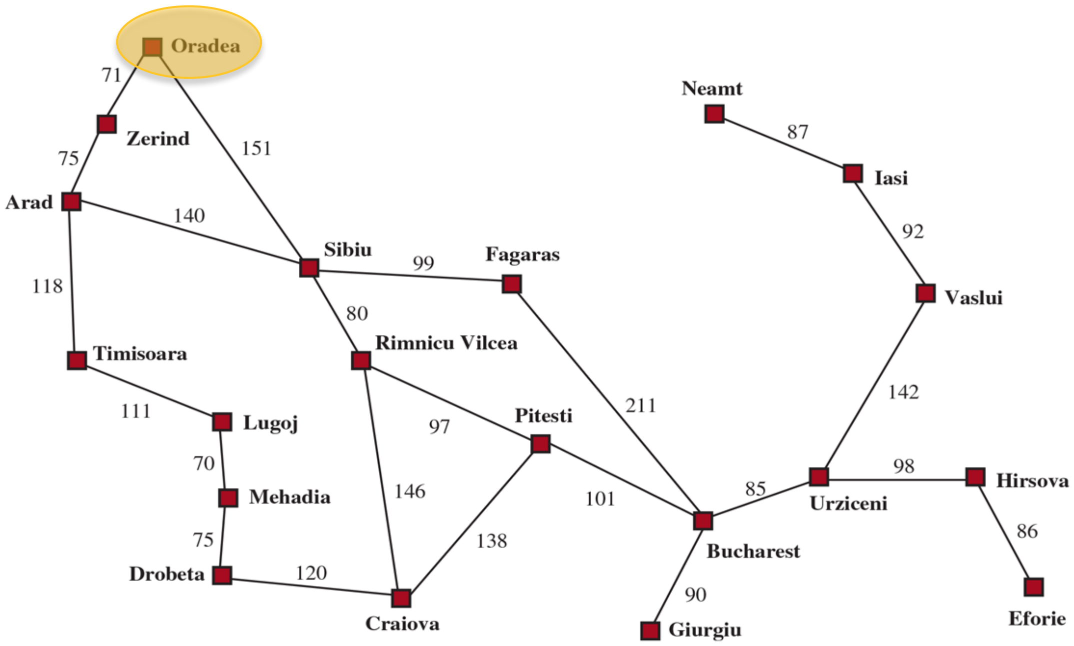

Fig 3.1, Russell & Norvig's Textbook

```
s = Oradea,
a = ToZerind,
s'= Zerind

cost(s,a,s') = 71 miles
```

A State Space

Initial State

Goal State(s)

Actions

Transition Model

Action-Cost Function

Defining the Search Problem

Example Problems

- Robo-Vacuum

- Sliding-Tile Puzzle

- Route Fining

- Automatic Motor Part Assembly

Formally Define the Search Problem for the Sliding-Tile Puzzle.

Search Algorithm

It is an algorithm that takes a search problem as an input and it returns a solution to the problem, or an indication that the problem has no solution.

Some search algorithms represents the problem as **a tree structure**. The root holds the **initial state**. The **nodes** represent different **states** that are reached by taking certain **actions**. Actions are represented by the **edges** connecting the nodes.

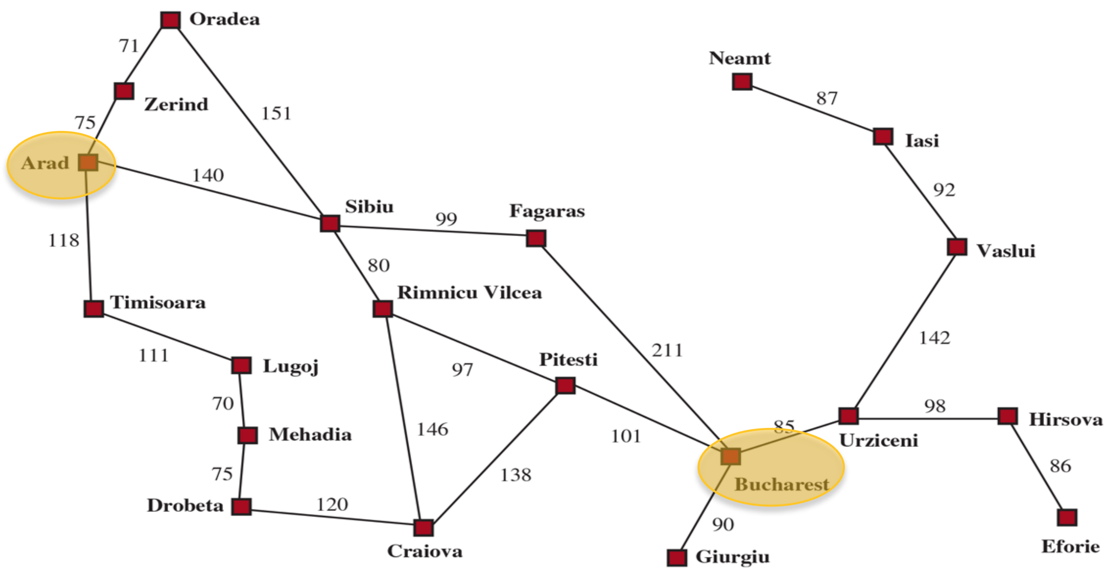

Fig 3.1, Russell & Norvig's Textbook

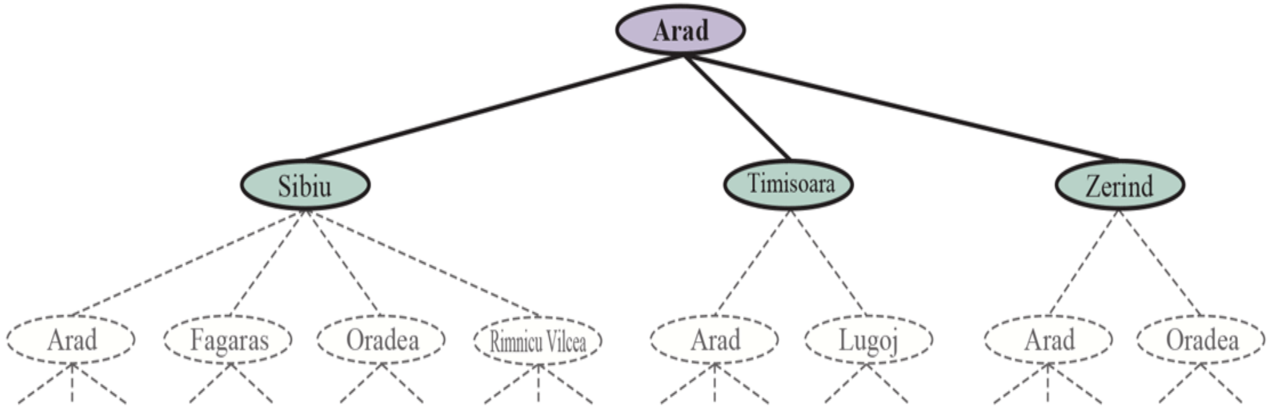

Fig 3.4, Russell & Norvig's Textbook

A search algorithm works by **expanding** nodes in the search tree. Each expanded node leads to a **new state**. The algorithm keeps **checking** if any of the expanded states is a **goal state**. The path from the initial state to a goal state is the **solution path**.

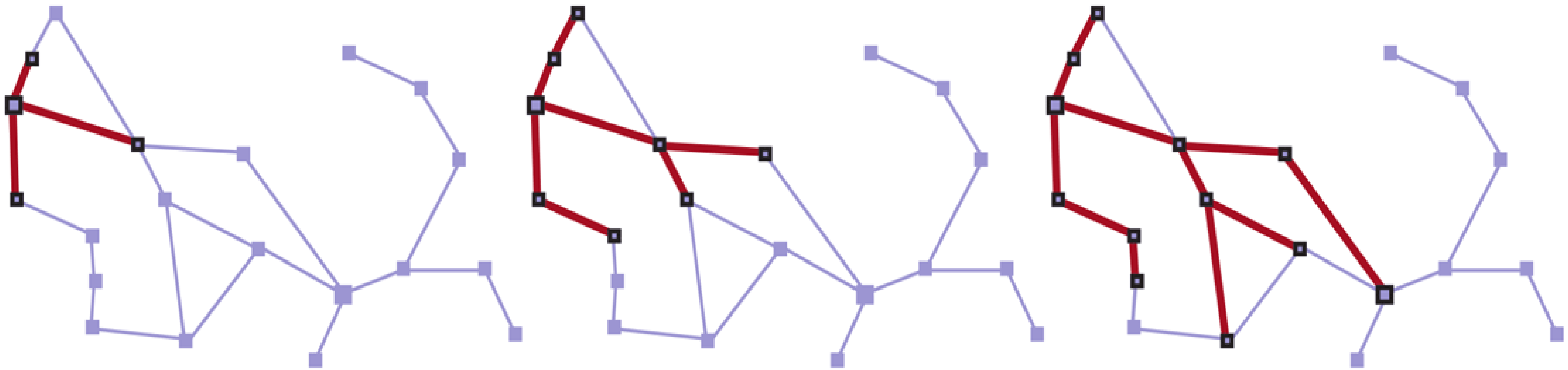

Fig 3.5, Russell & Norvig's Textbook

Which node should a search algorithm expand first??

Best-first Search

It is an algorithm that decides which node to expand based on some evaluation function f(n). The node that has the minimum f(n) is expanded first.

Best-first Search

Best-first Search

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Fig 3.7, Russell & Norvig's Textbook

AI Agent Program

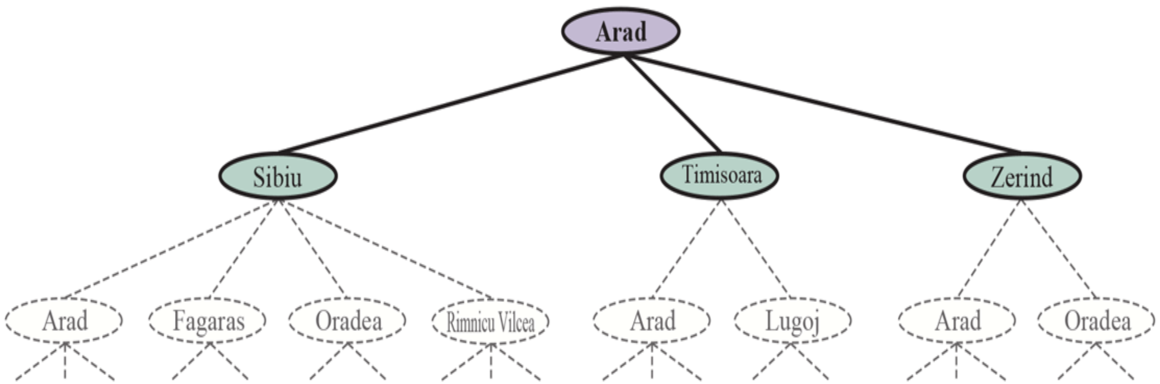

Fig 3.4, Russell & Norvig's Textbook

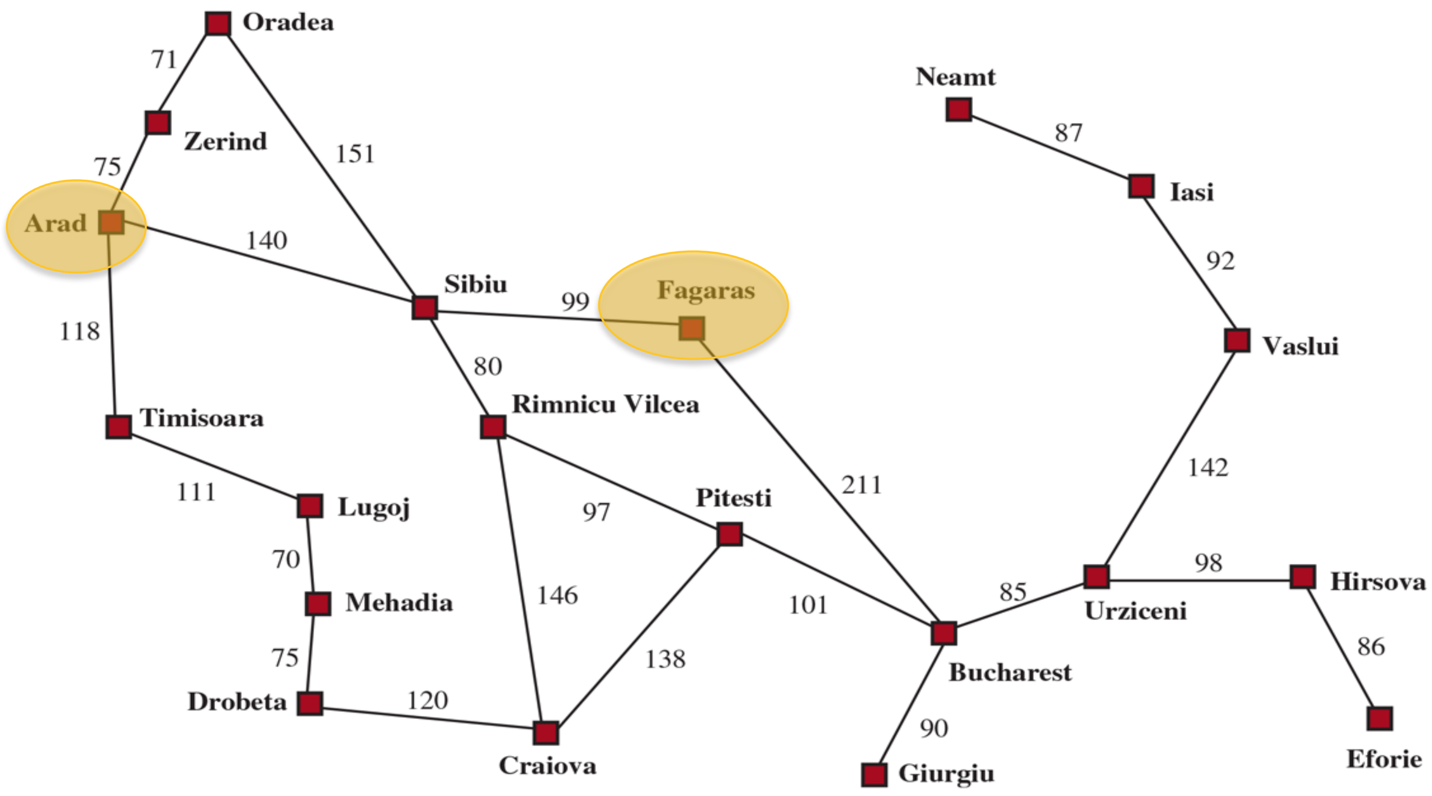

Fig 3.1, Russell & Norvig's Textbook

The search might return to state that was previously reached:

Arad➔ Sibiu➔ Arad

The search might reach one state through different paths:

Arad➔ Sibiu➔ Fagaras

Arad➔ Zerind➔ Oradea➔ Sibiu➔ Fagaras

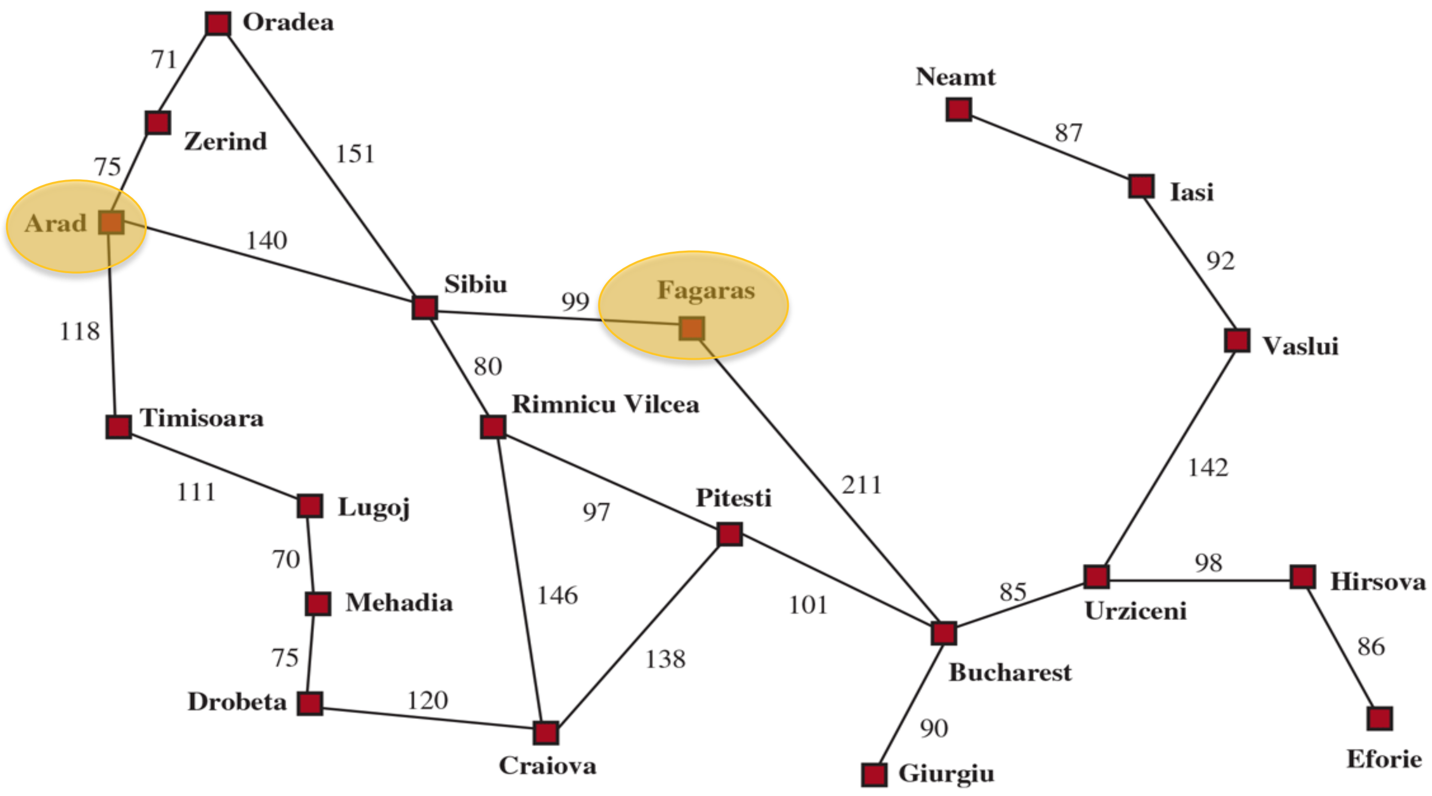

Fig 3.1, Russell & Norvig's Textbook

Th … te
th

Ar

Cycles & Redundant Paths

Th … he
st

Ar

Arad➔ Zerind➔ Orac

Cycles & Redundant Paths

Cycles or redundant paths lead to infinite, useless search paths! The program may get into an infinite loop.

“Algorithms that can’t remember the past are doomed to repeat it.”

~ Ch3, Russell and Norvig’s

The algorithm should remember all visited states, detect redundant paths, keep only the most optimum path to each state.