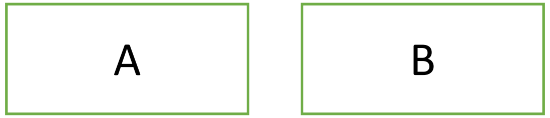
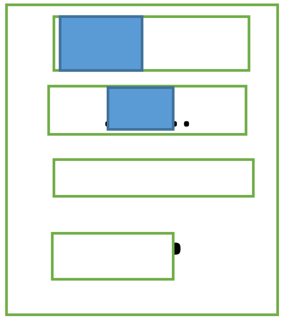
Agents - states

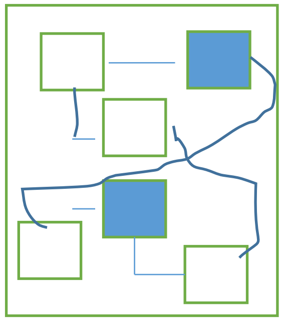
- Atomic



- Factored



- Structured



Atomic: each state of the world is indivisible. No internal structure
Ex: finding a path from Kashmir to Kanyakumari via some sequence of cities.
State of the world – name of the city we are in – discernible property, identical or different from other state.
Algorithms underlying search, game playing, hidden Markov models, Markov decision processes work with atomic state representations

Factored: State consists of a vector of attribute values.
Constraint satisfaction algorithms, propositional logic, planning, Bayesian Networks, Machine Learning Algorithms work with Factored state representations

Structured: State includes objects each of which may have attributes of its own as well as relationships to other objects
First order logic, first order probability models, knowledge based learning, NLP

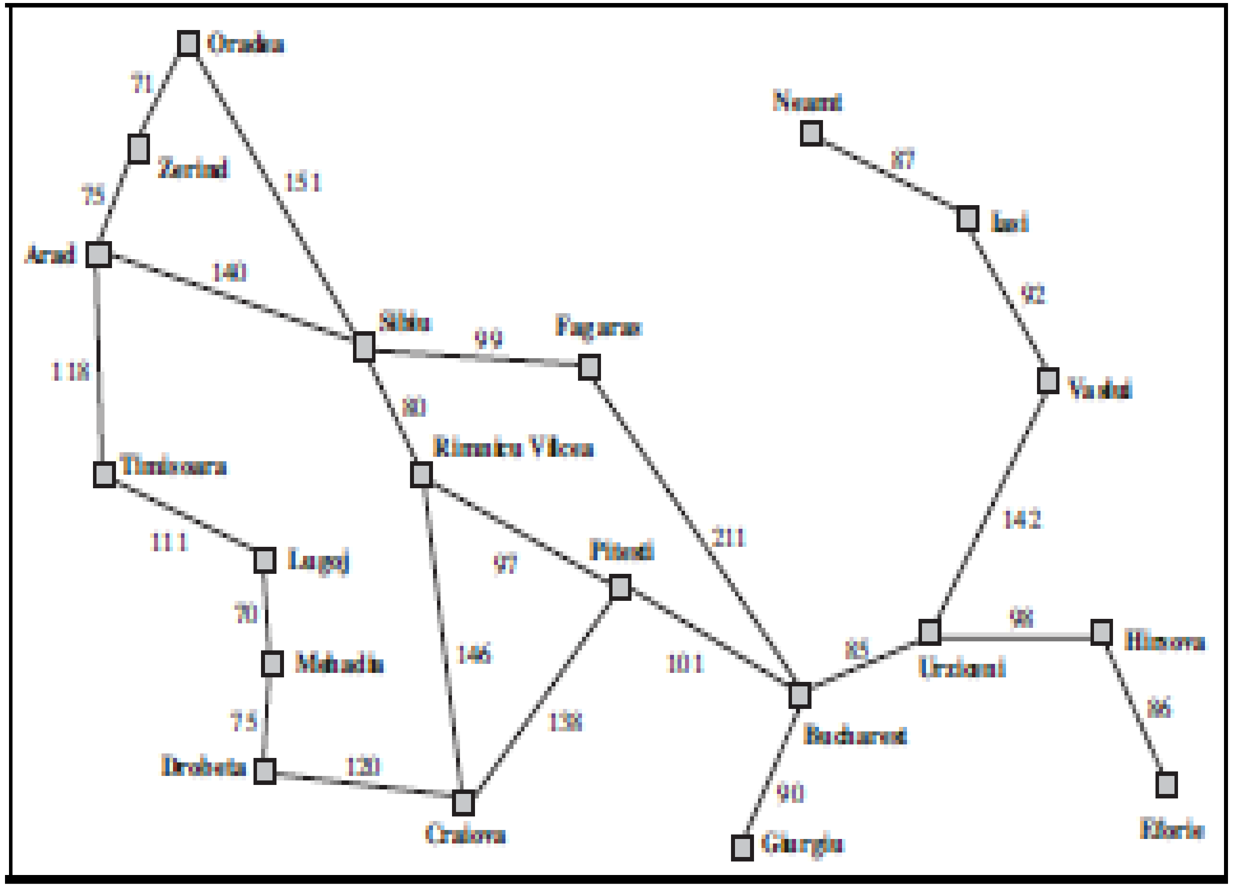
Goal based Agent – Problem Solving Agent

- Uses Atomic representations for the states of the world
- Problems/ Task environment - simple
- Solutions – Agents –
  - General purpose search algorithms –
    - Uninformed
    - Informed
  - A fixed sequence of actions
- Intelligent agents are supposed to maximize their performance measure
- Achieving the above is simplified if the agent can adapt a goal and aim at satisfying it.
- Goals help organize behaviour by limiting the objectives the agent is trying to achieve, hence the actions it needs to consider
- **GOAL FORMULATION**
  - The first step of problem solving, based on the current situation and the agents performance measure
  - Goal – a set of world states – exactly those states in which the goal is satisfied
- Agent's task is to find out how to act, now and in the future, so that it reaches a goal state
  - Before it can do this, agent needs to decide what sorts of actions and states it should consider

Problem Solving agent

- What level of actions?
  - Move left foot forward an inch?, turn the steering wheel one degree left?
  - Too much of uncertainty in the world at this level of detail, and there would be many steps in the solution
- PROBLEM FORMULATION
  - The process of deciding what actions and states to consider, given a goal
  - Agent may consider actions at the level of driving from one major town to another
  - Each state, therefore, corresponds to ‘being in a particular town’

Example – Driving In Romania



Goal: Drive to Bucharest

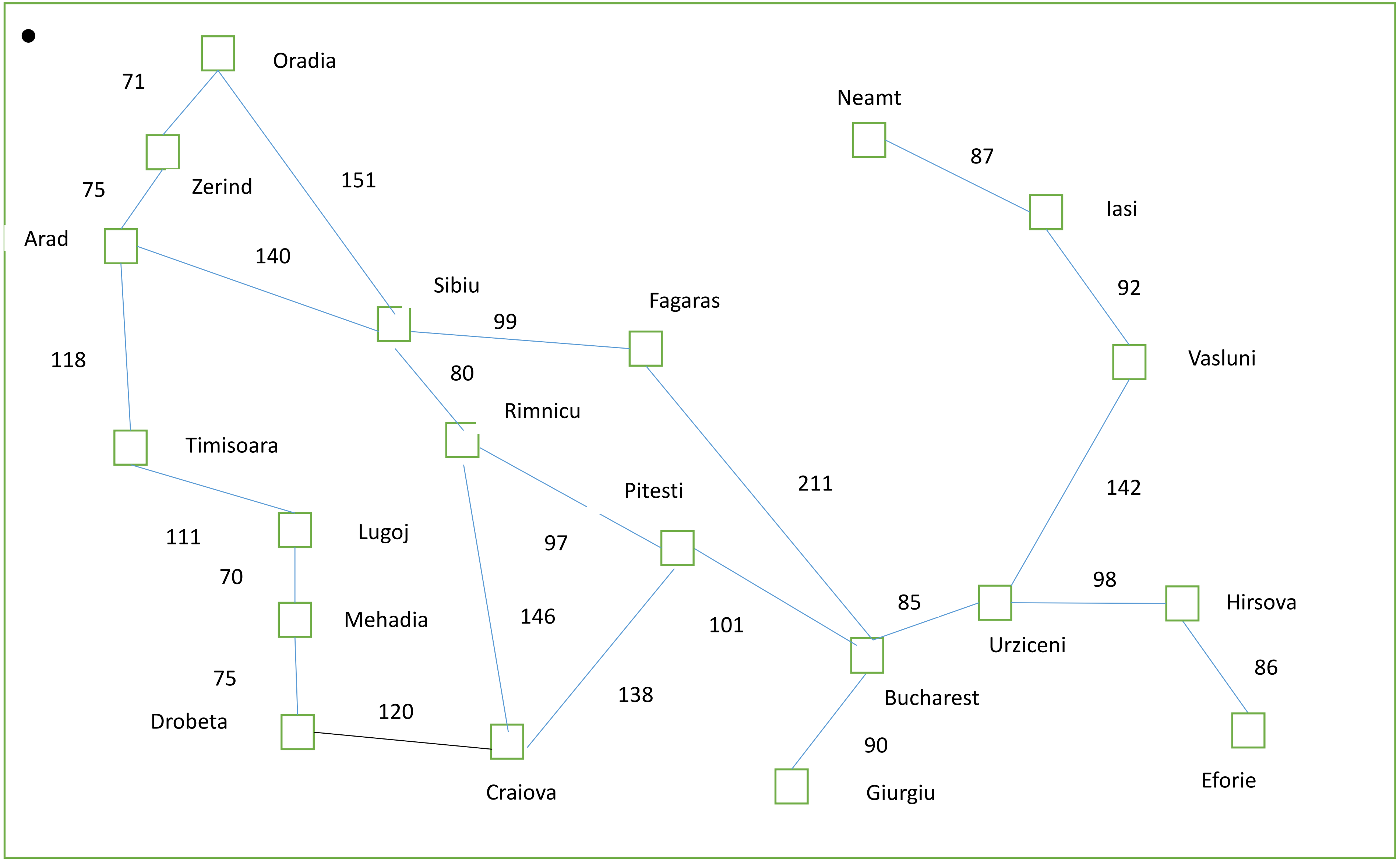
Current State: Arad

3 roads from Arad

Arad →Zerind
Arad → Sibiu
Arad → Timisoara

None achieve the goal!

Example – Driving In Romania



Example

- Environment
    - Unless the agent is familiar with the geography of Romania, it will not know which road to follow
    - Agent will not know which of its possible actions is best, as it does not yet know enough about the state that results from taking each action
- If the agent has no additional information – if the **environment is unknown** – then agent has no choice but to try one of the actions at random!
    - Map of Romania is with the agent.
    - Map provides agent with information about the states it might get itself into and the actions it can take
    - Agent can use this information to consider subsequent stages of a hypothetical journey via each of the three towns trying to find a journey that will eventually gets to Bucharest
    - Once it has found a path on the map, from Arad to Bucharest, it can achieve its goal by carrying out the driving actions that correspond to the legs of the journey.
- An agent with several immediate options of unknown value, can decide what to do by first examining future actions that eventually lead to states of known value
    - Examining future actions means – we need to be more specific of the environment – assume **environment is Observable,** so that agent always known the current state - each city has some indication to let agent know it presence when driver arrives at that place.
- The environment is assumed to be **Discrete**
    - At any given state, there are only finitely many actions to choose from – in navigating in Romania, each city is connected to a small number of other cities

Problem Solving Agent

- Assumptions
- Assume environment is **Known** – agent knows which states are reached by each action
- Assume the environment is **deterministic** – each action has exactly one outcome – means if the agent chooses to drive from Arad to Sibiu, it does end up at Sibiu.
- Under these assumptions the solution to any problem is a fixed sequence of actions

"If the agent knows the initial state, and the environment is known and deterministic, it knows exactly where it will be after first action, and what it will perceive. Since only one percept is possible after first action, the solution can specify only one possible second action, and so on"

- The process of looking for a sequence of actions that reaches the goal is called **SEARCH**
- A search algorithm takes the problem as input and returns a solution in the form of action sequence.
- Once a solution is found, the actions it recommends can be carried out – Execution Phase
- **FORMULATE – SEARCH – EXECUTE** design for the agent
- After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. Then, the agent uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do.

Problem Solving Agent

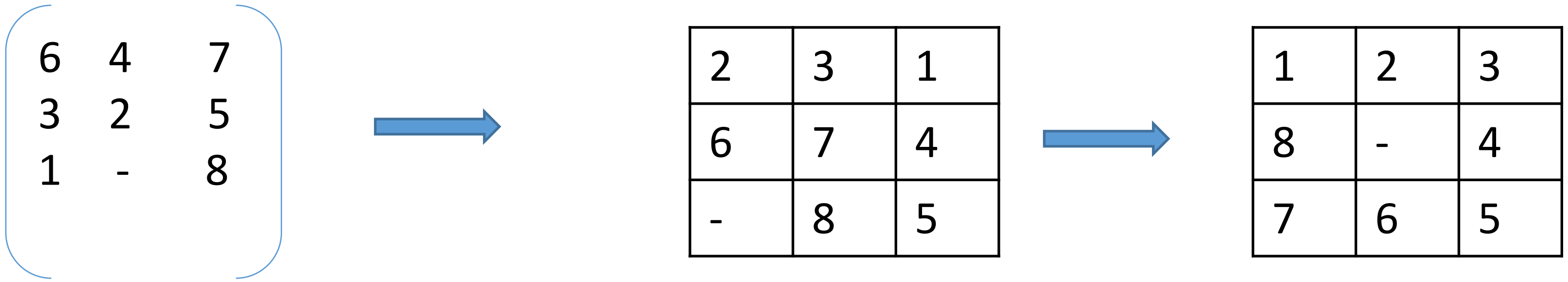
```
Function Simple-Problem-Solving-Agent (percept) returns an action
    persistent: sequence,    an action sequence, initially empty
                state,       some description of the current world state
                goal,        a goal initially null
                problem,     a problem formulation
    state ← Update-State (state, percept)
    if sequence is empty then
        goal ← Formulate-Goal (state)
        problem ← Formulate-Problem (state, goal)
        sequence ← Search ( problem)
    if sequence = failure then return NULL action
    action ← First (sequence)
    sequence ← rest (sequence)
    return action
```

Slide-bar puzzles

initial state

intermediate state

goal state



$$\begin{pmatrix} 6 & 4 & 7 \\ 3 & 2 & 5 \\ 1 & - & 8 \end{pmatrix}$$

2	3	1
6	7	4
-	8	5

1	2	3
8	-	4
7	6	5

Slide bar Puzzle Problem:

numbers in the cells required to be arranged in a particular format from any given arbitrary format.

State: Format

Actions:

change of state can be performed by movement of bar to its left, right, up, down positions depending on its particular position

Slide-bar puzzles

8 – puzzle

2	3	1
7	8	4
6	-	5

→

1	2	3
6	5	4
7	8	-

15 puzzle

4	5	1	2
11	13	14	6
12	-	15	7
10	3	8	9

→

1	2	3	4
12	13	14	5
11	-	15	6
10	9	8	7

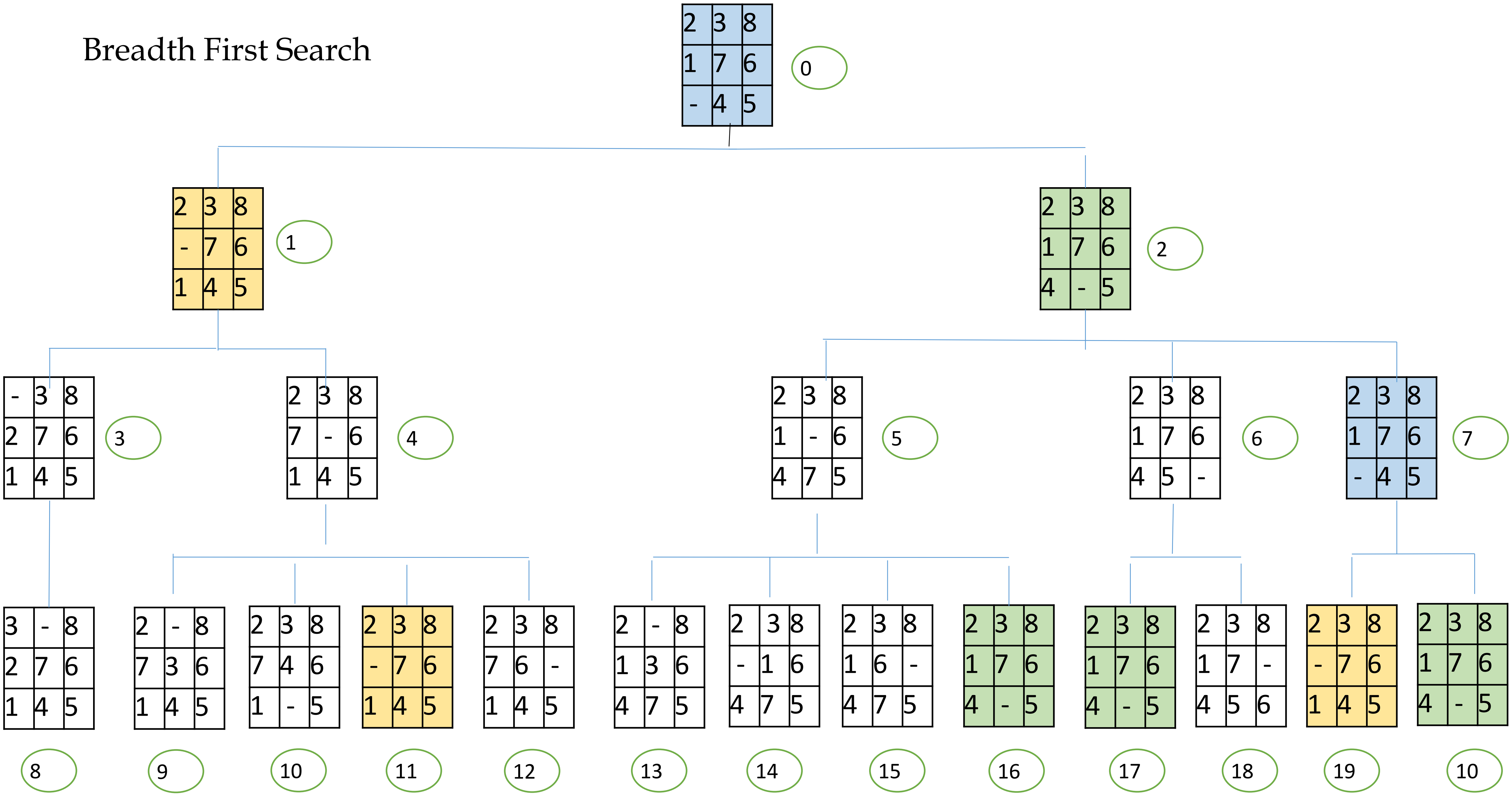
24 puzzle

5	18	20	21	24
4	19	-	22	23
3	17	16	15	14
2	10	11	12	13
1	6	7	8	9

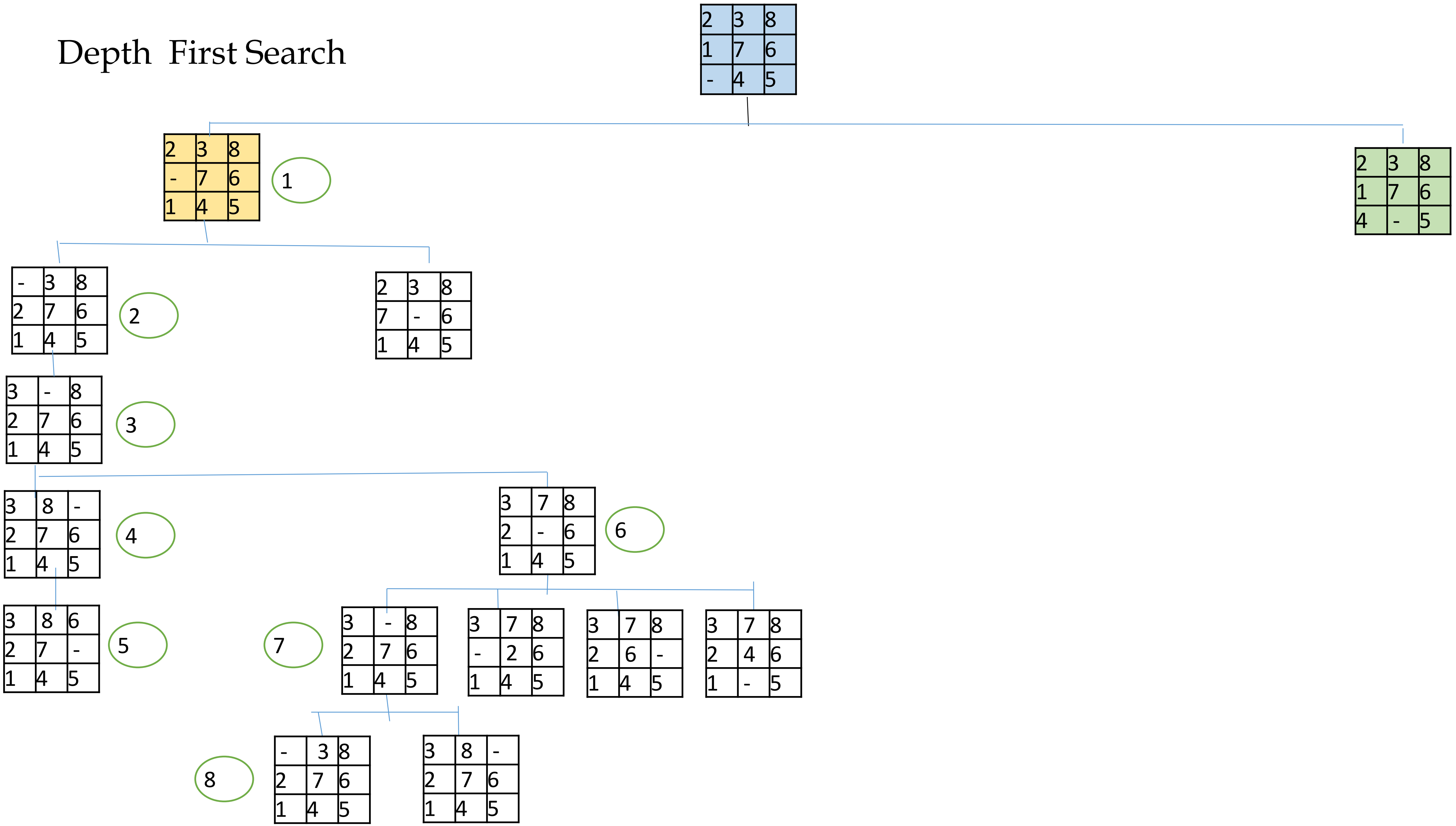
→

24	23	22	21	20
15	16	17	18	19
14	13	12	11	10
5	6	7	8	9
4	3	2	1	-

Breadth First Search



Depth First Search



Breadth First Search

- All nodes at a given level are explored before moving to the next level
- One can always find a minimal path length solution when one exists
- Uses a generic structure to hold all generated but still unexplored nodes
- Search Process:
    - Expand initial state
    - If none of these nodes are goal nodes, then all the nodes are expanded to the next level (by applying all applicable operators to produce all successors of each of the nodes turn by turn)
    - The process continues till the goal node is achieved
- Number of nodes grows exponentially from one level to another with increasing depth
- Time requirement is also exponential
- Requires unlimited memory and time

Breadth First Search

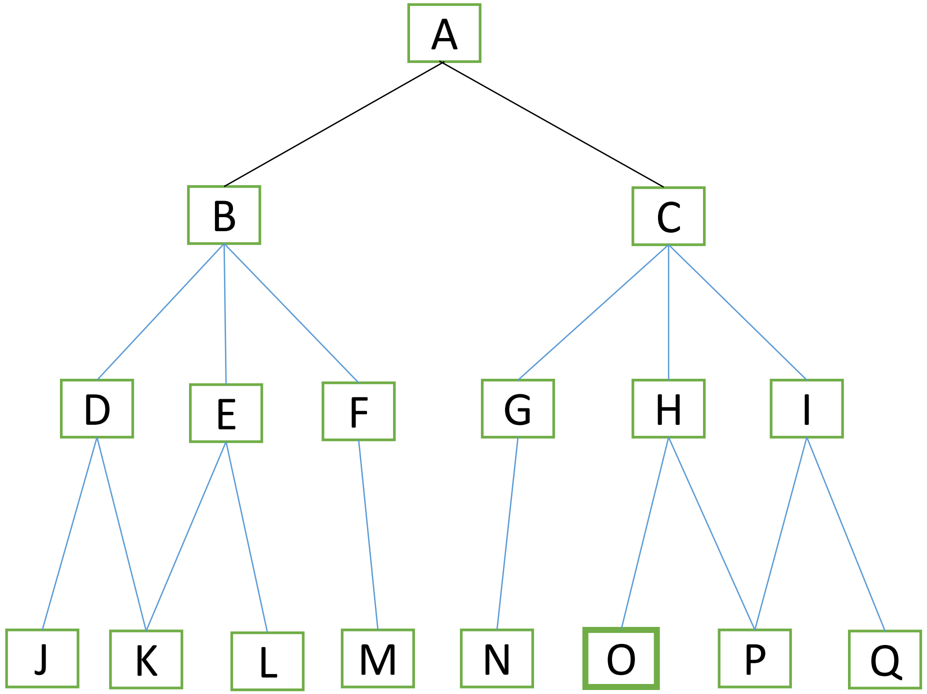
- Algorithm

```
Begin initial state: open
initialise : = [start]
closed: = [ ]
/start with open containing the first initial state until a goal is found or there are none left on open
do: remove
/remove the particular node (M) from open
get goal: if M is a goal then return success
generate children of M:
put M on closed
```

If the children of M are already on closed or open, ignore them. Put remaining children on left end of open.

QUEUE structure

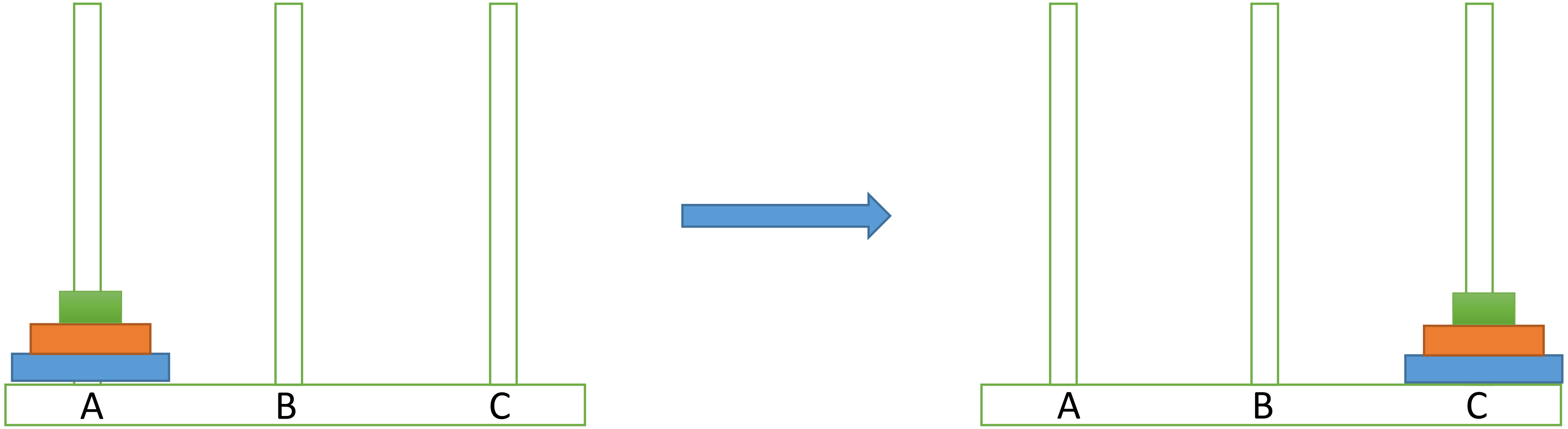
Breadth First Search

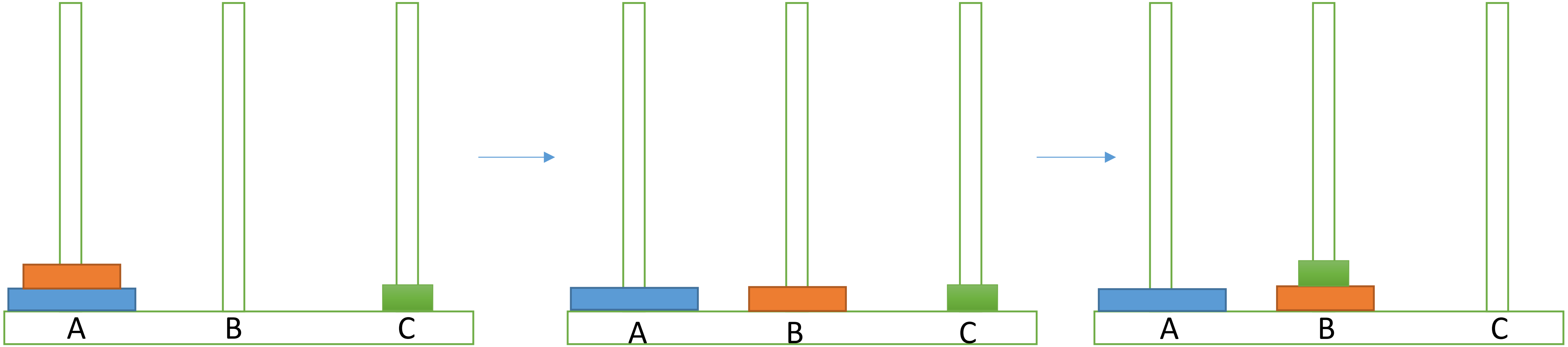


open [A] ; closed [ ]
open [B, C] ; closed [A]
open [C, D, E, F] ; closed [B, A]
open [D, E, F, G, H, I] ; closed [C, B, A]
open [E, F, G, H, I, J, K] ; closed [D, C, B, A]
open [F, G, H, I, J, K, L] ; closed [E, D, C, B, A] / note K is already open
open [G, H, I, J, K, L, M] ; closed [F, E, D, C, B, A]
open [H, I, J, K, L, N] ; closed [G, F, E, D, C, B, A]
open [I, J, K, L, O, P] ; closed [H, G, F, E, D, C, B, A]
open [J, K, L, O, P, Q] ; closed [I, H, G, F, E, D, C, B, A]
open [K, L, O, P, Q] ; closed [J, I, H, G, F, E, D, C, B, A]
open [L, O, P, Q] ; closed [K, J, I, H, G, F, E, D, C, B, A]
open [O, P, Q] ; closed [L, K, J, I, H, G, F, E, D, C, B, A]

Tower of Hanoi

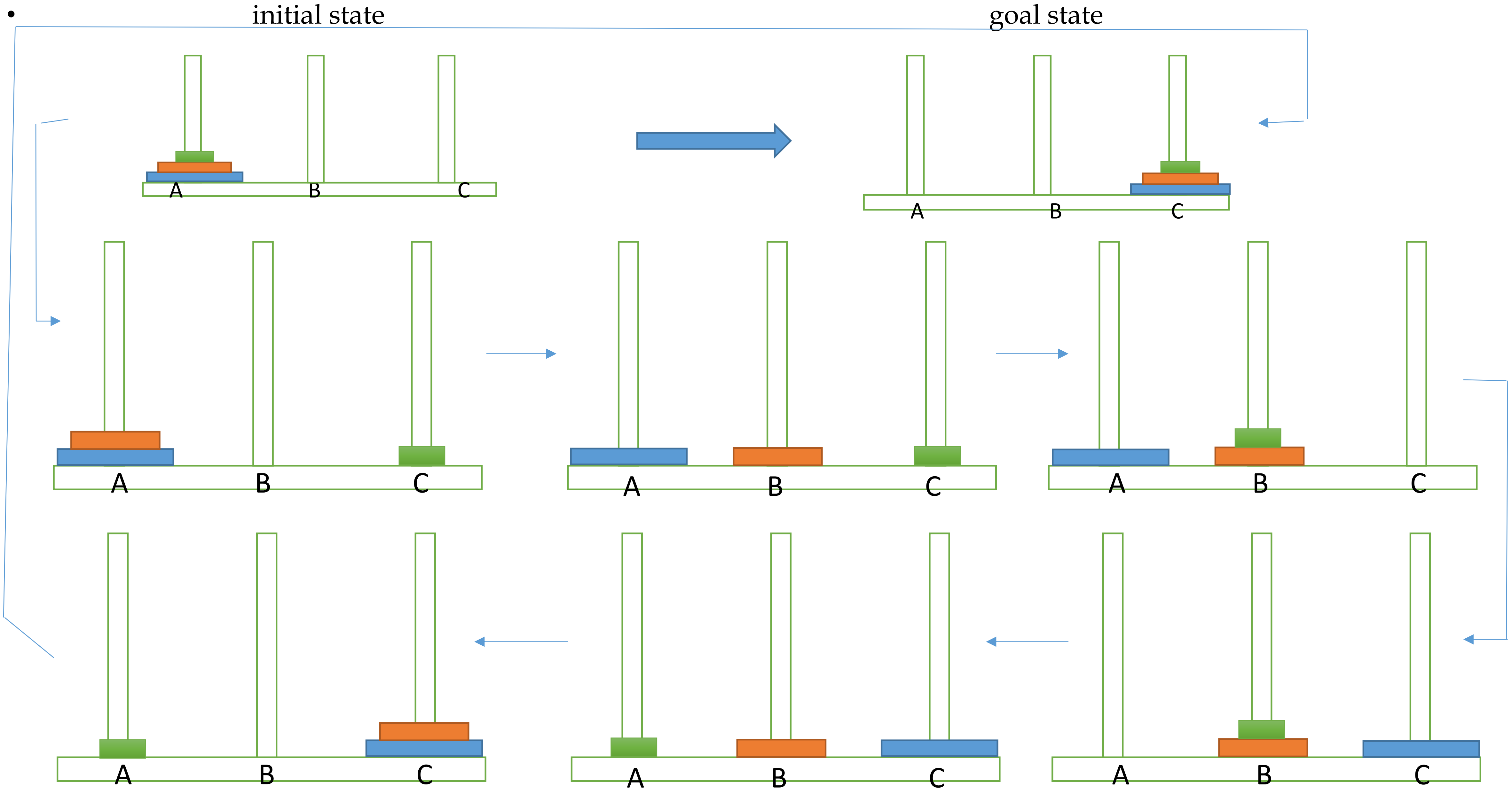
- initial state (a a a) goal state (c c c)





Tower of Hanoi

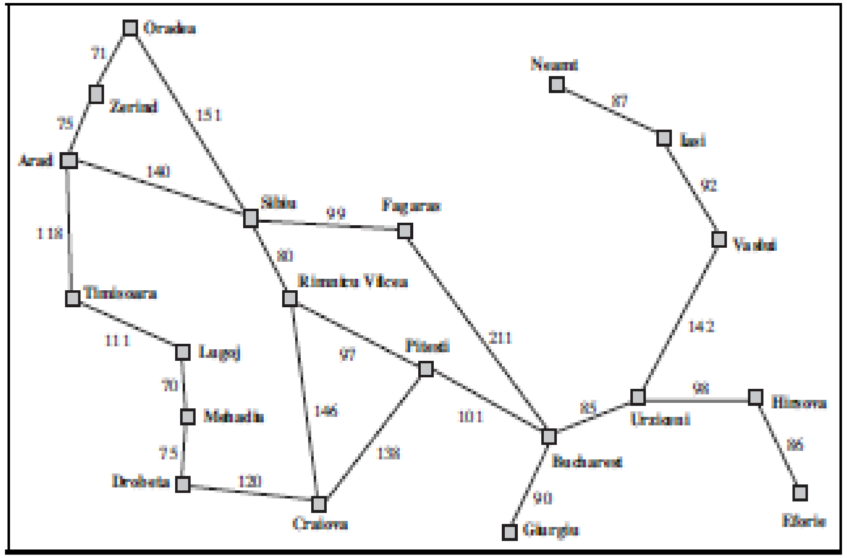
-



Tower of Hanoi

- Current State:
  - Three pegs in the courtyard of the ministry. One of them contained 64 disks, placed in order of lowering diameter from bottom, through the holes in the disk.
- Task of the monks:
  - transfer all the disks from one peg to another to be placed in the same order using the third peg as an intermediate temporary placing stand.
- Conditions:
  - Only one disk can be moved at a time
  - No disk should have above it, another disk with larger diameter

Problems and Solutions



- 5 components
- Initial State
  - that Agent starts in ex: In(Arad)
- Actions
  - Possible actions available/applicable to the agent given a particular state ‘s’.
  - Actions(s) to return set of actions that can be executed in state ‘s’
  - Ex: **s** = In (Arad);   actions applicable for **s**   = { go (Sibiu), go (Timisoara), go (Zerind) }
- Transition Model
  - Description of what each action does
  - Specified by Function   - **result (s, a)**   - returns the state, **s’** that results from doing the action **a** in state **s**
  - Successor refers to any state, s’ reachable from **a** given state, **s** by a single action, **a**
  - Ex: Result (in (Arad), go (Zerind)) = In (Zerind)
  - State space: initial state + actions + transition model   - set of all states reachable from the initial state by any sequence of actions
  - State space forms a directed network/graph –
    - nodes ae states, links between nodes are actions
  - Ex: map of Romania can be viewed as a state space graph. Each city a state and each road driving action
  - Path in the state space is a sequence of states connected by a sequence of actions

Problems and Solutions

- Goal Test
  - Determines whether a given state is a goal state.
  - There could be an explicit set of possible goal states, goal test checks whether the given state is one of them
  - Ex: agents goal in Romania - the singleton set: { in (Bucharest)}
  - Goal can be an abstract property - ex: chess – goal is to reach a state called checkmate, where the opponent's king is under attack and can't escape
- Path Cost Function
  - Assigns a numeric cost to each path
  - Cost function reflects its own performance measure
  - Ex: for the agent trying to get to Bucharest cost of a path – length in km?
  - Path cost is sum of the costs of the individual actions (step cost) along the path.
  - C (s, a, s′) : Step cost of taking action a in state s to reach s′ (C non negative?)

Problem definition – initial state + actions + transition model + goal test + path cost

Problem solving algorithm takes Problem definition as input

Solution to a problem - an action sequence that leads from initial state to a goal state

Solution quality - measured by path cost function

An optimal solution has the lowest path cost among solutions

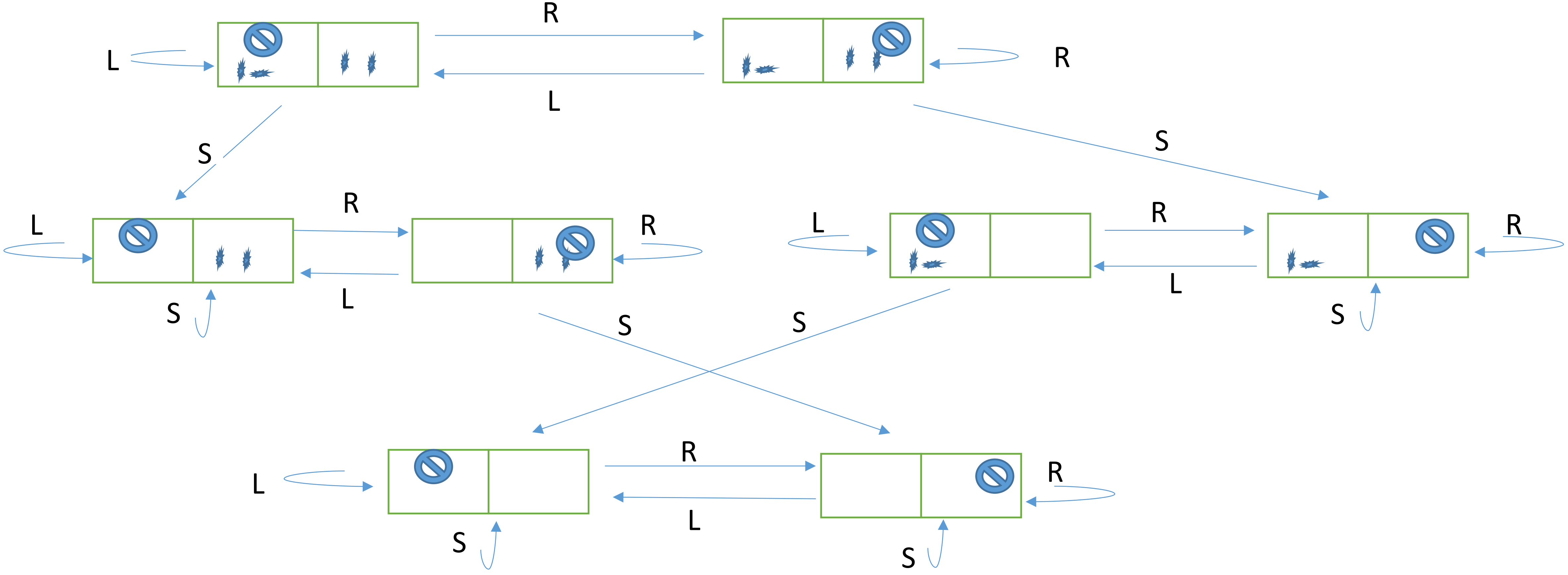
Formulating Problems

- Level of abstraction
- Abstraction
  - Process of removing detail from a representation
- Good Abstraction
  - process of removing as much detail as possible from a representation while retaining validity and ensuring that the abstract actions are easy to carry out
- Ex:
  - In (Arad) – **state** abstraction . Ignored condition of road, weather, scenery out of window, …
  - Change in location – **action** abstraction . Ignored slowing down fro traffic signals, turning on radio, …
  - Path from Arad to Sibiu to Rimincu to Pitesti to Bucharest - abstraction of **solution**

Vacuum World

- Problem Formulation
    - State:
        - Determined by agent location and dirt location
        - Agent is in one of the 2 locations; each of the location might or might not contain dirt
        - Number of possible world states? 2×2^2
        - A large environment with n locations – possible states? $n \times 2^n$
    - Initial state: any state can be designated as initial state
    - Actions: each state has 3 actions – Move Left, Move Right, Suck Dirt (can include up down actions also)
    - Transition Model:
        - actions have their expected effect,
        - except moving left in left most square, moving right in right most square, suck dirt in clean square have no effect
    - Goal test: checks whether all squares are clean
    - Path cost: each step costs 1. path cost is the number of steps in the path
- Vacuum word has discrete locations, discrete dirt, reliable cleaning, never gets dirtier any again.

Vacuum World



Sliding Block puzzles: 8 puzzle

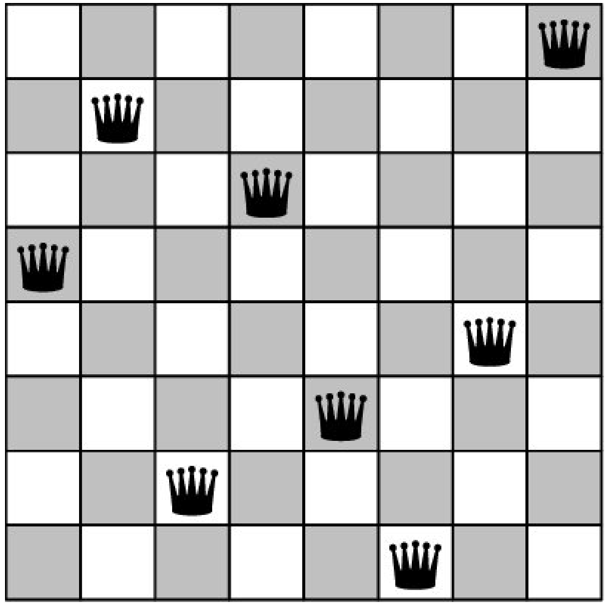
- 8 Puzzle problem formulation
    - 3 x 3 board, eight numbered tiles and a blank space
    - **States**: location of each of the eight tiles and the blank in one of the nine squares
    - **Initial state**: any state can be designated as initial state (any given goal can be reached from exactly half of the possible initial states.)
    - **Actions**: movement of blank space left, right, up, down
    - Transition model: given a state and action, returns the resulting state
    - Goal test: checks whether the state matches the goal configuration
    - Path cost: each step costs 1. path cost is the number for steps in the path
- Abstraction: description of rules in the puzzle
- Sliding Block problems -
    - Used as test problems for new AI search algorithms
    - NP complete problems
- 8 puzzle - Number of states – 9! Reachable states = 9!/2 = 181440 - solvable
- 15 puzzle - Number of states – 16! Reachable states = 16!/2 = 1.3 trillion states – random instances can be solved optimally!
- 25 puzzle - Number of states – 25! Reachable states = 25!/2 ~ 10^{25} states - random instances take several hrs to solve optimally

8 Queens Problem

- Goal of 8 queens problem is to place 8 queens on a chess board such that no queen attacks any other. A queen attacks any piece in the same row, column or diagonal
- Problem Formulation
  - Incremental formulation
    - Starts with an empty state
    - Involves operators that augment the state description --- Each action adds a queen to the state
  - Complete state formulation
    - Starts with all 8 queens on the board
    - Actions move the queens around
  - Path cost – does not matter. Only the final state counts
- **Incremental formulation - 1**
  - States: any arrangement of 0 to 8 queens on the board
  - Initial state: no queen on the board
  - Actions: add a queen to any empty square
  - Transition model: returns the board with a queen added to the specified square
  - Goal test: 8 queens on the board, none attacked
- Possible sequences to investigate --- $64 \times 63 \times \ldots \times 57 \sim 1.8 \times 10^{14}$

8 Queens Problem

- Goal of 8 queens problem is to place 8 queens on a chess board such that no queen attacks any other. A queen attacks any piece in the same row, column or diagonal
- **Incremental formulation – 2 :** prohibit placing a queen in any square that is already attacked.
  - States: all possible arrangements of n queens (0<= n <= 8) of 0 to 8 queens, one per column in the left most n columns, with no queen attacking another
  - Initial state: any of the states
  - Actions: add a queen to any square in the left most empty column such that it is not attacked by any other queen
  - Transition model: returns the board with a queen added to the specified square
  - Goal test: 8 queens on the board, none attacked
- Possible sequences/state space to investigate --- 2057



Knuth’s Conjecture

How infinite state spaces can arise

- starting with number 4 , a sequence of factorial, square root and floor operations will reach any desired positive integer
- ex: Floor[Sqrt (Sqrt (Sqrt (Sqrt (Sqrt (4!)!)] = 5

Problem Definition

states:	positive integers
Initial state:	4
Actions:	Apply factorial, Square root, or Floor Operations
Transition model:	as given by the mathematical definitions of the operators
Goal test:	state is the desired integer

Real world problems

Route finding problem

defined in terms of specified locations and transitions along links between them

used in websites, in-car systems, video streaming in computer networks, military operations' planning, airline travel planning

Ex: Travel planning website (additional complexities – contingency plans, alternate flights)

States: [location/airport, current time, fair base, status]

(fair base - because cost of a flight segment may depend on previous segments,

status can be domestic/international)

Initial state: specified in user query   - source location/start

Actions: take any flight from current location, in any seat class, leaving after current time, leaving enough time for within airport transfer

Transition model: the state resulting from taking a flight will have the flight's destination as the current location, and the flight's arrival time as the current time

Goal test: are we at the final destination specified by the user?

Path cost: monetary cost, waiting time, flight time, …

Real world problems

Touring Problem

ex: visit every city in Romania(20 cities) at least once starting and ending in Bucharest

Actions: trips between adjacent cities (similar to route finding)

State Space: each state must include not just the current location, but also a set of cities the agent has visited

Initial state: in (Bucharest), visited ({Bucharest})

Intermediate state: in (Vaslni), visited ({ Bucharest, Urziceni, Vaslni})

Goal test: whether the agent is in Bucharest all 20 cities have been visited

Real world problems

Travelling Sales person Problem

Touring problem + find shortest tour (NP Hard Problem)

TSP algorithms used in

planning trips for sales persons

planning movements of automated circuit board drills

Stocking machines on shop floors

Real world problems

VLSI layout: positioning components and connectors on a chip to minimize area, circuit delays, stray capacitances, maximize manufacturing yield

Logical design → layout problem (cell lay out, channel routing)

Primitive components of the circuit are grouped into cells, each of which performs some recognized function.

Each cell has a fixed foot print (size, shape)and requires a certain number of connections to each of the other cells

Problem: place cells on the chip in such a way that they do not overlap and there is room between cells for the connecting wires.

Channel routing - finds a specific route for each wire through the gaps between the cells

Real world problems

Robot Navigation

- Generalization of route finding problem
  - a robot can move in a continuous space(instead of discrete set of routes) with an infinite set of possible actions and states.
  - Circular robot moving on wheels – space is 2 D

Automatic assembly sequencing of complex objects

- aim: find an order in which to assemble the some object

Searching for Solutions

Solution

  an action sequence

Search algorithms

  work by considering various possible action sequences

Possible action sequences

  starting from the initial state from a search tree

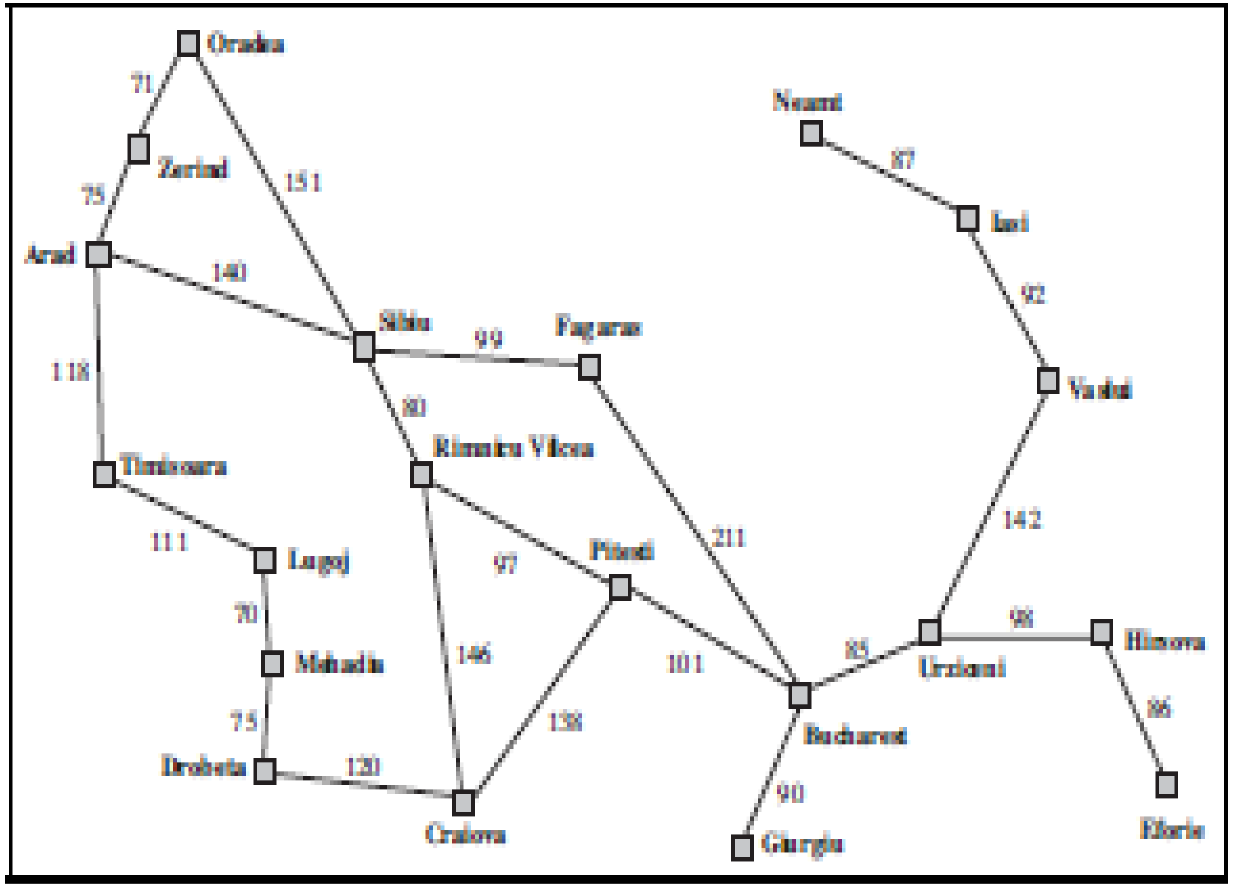
Search Tree

  initial state – root node

  Branches - actions

  Nodes – states in the state space of the problem

Example – Driving In Romania



Goal: Drive to Bucharest

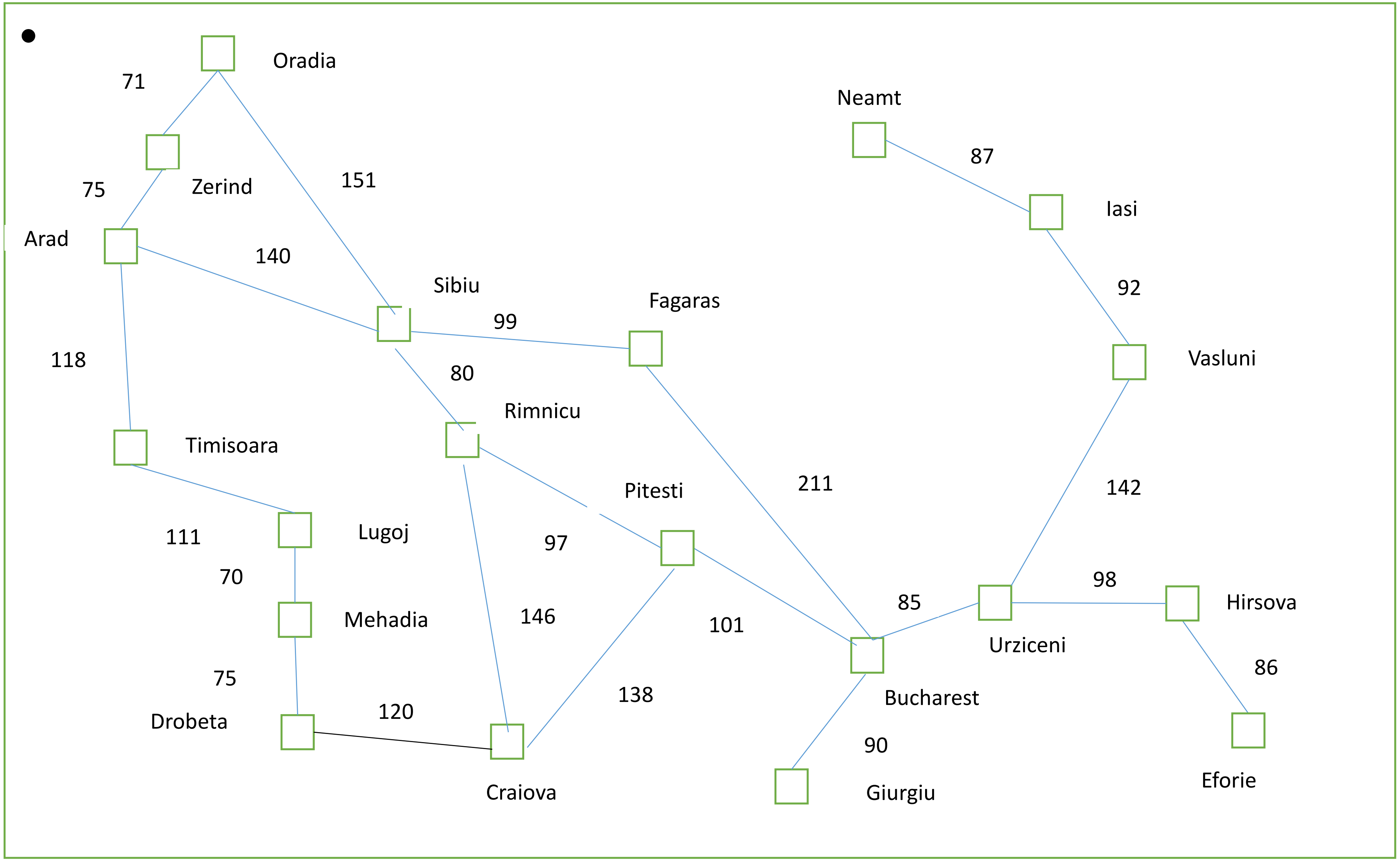
Current State: Arad

3 roads from Arad

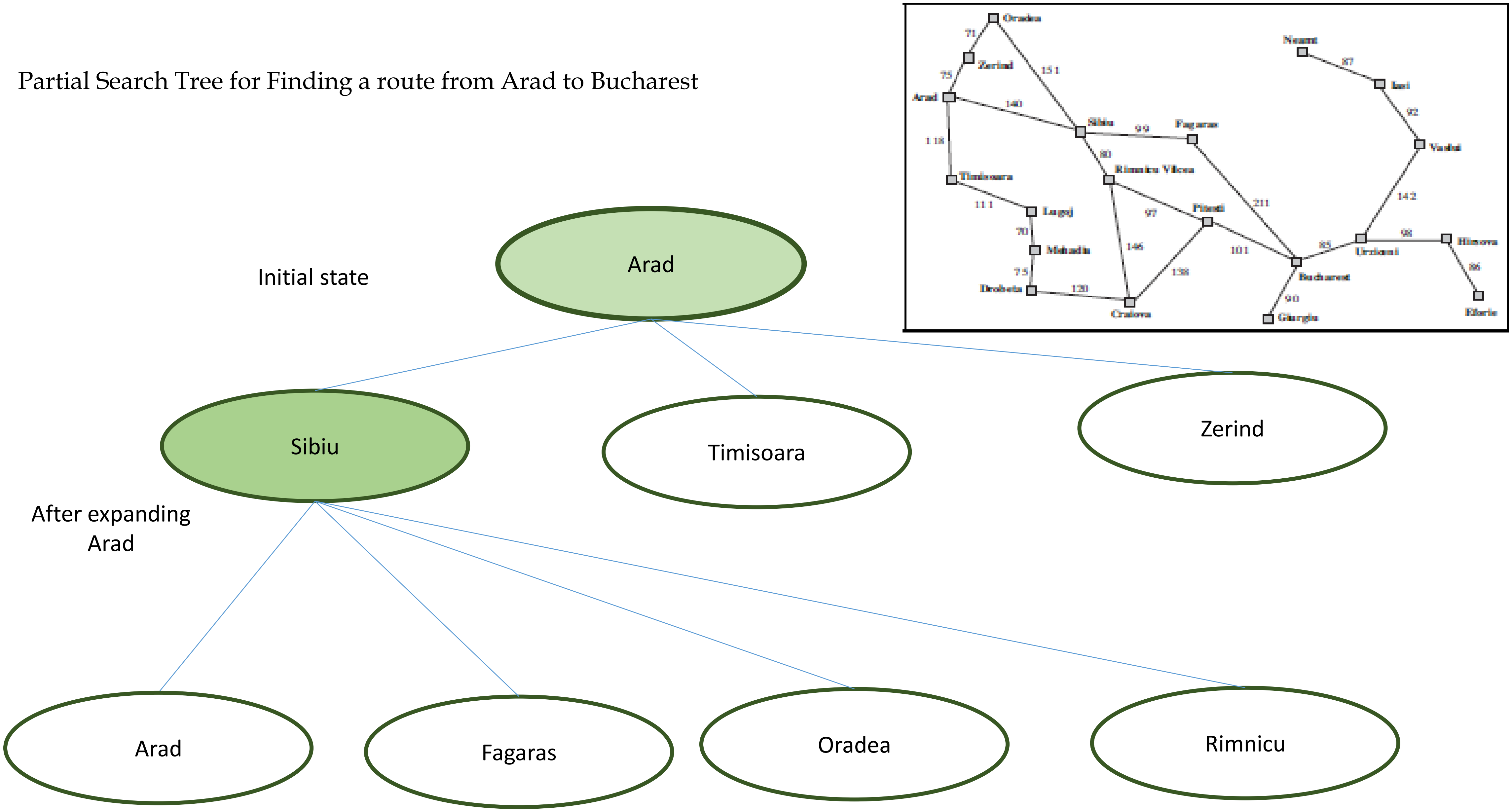
Arad →Zerind
Arad → Sibiu
Arad → Timisoara

None achieve the goal!

Example – Driving In Romania



Partial Search Tree for Finding a route from Arad to Bucharest



Initial state

Arad

Sibiu

Timisoara

Zerind

After expanding Arad

Arad

Fagaras

Oradea

Rimnicu

After expanding Sibiu

Route finding from Arad to Bucharest – growing the search Tree

1. Initial state - In(Arad)

    test whether it is a goal state

2. Consider various actions that can be taken

    by expanding the current state (by applying each permissible action to the current state, there by generating new set of states

    In Arad) → { In(Sibiu), In(Timisoara), In(Zerind) }

3. **Choose which of these possibilities to consider further**

    the essence of search – following up one action now

    putting others aside for later,

    in case the first choice does not lead to the solution

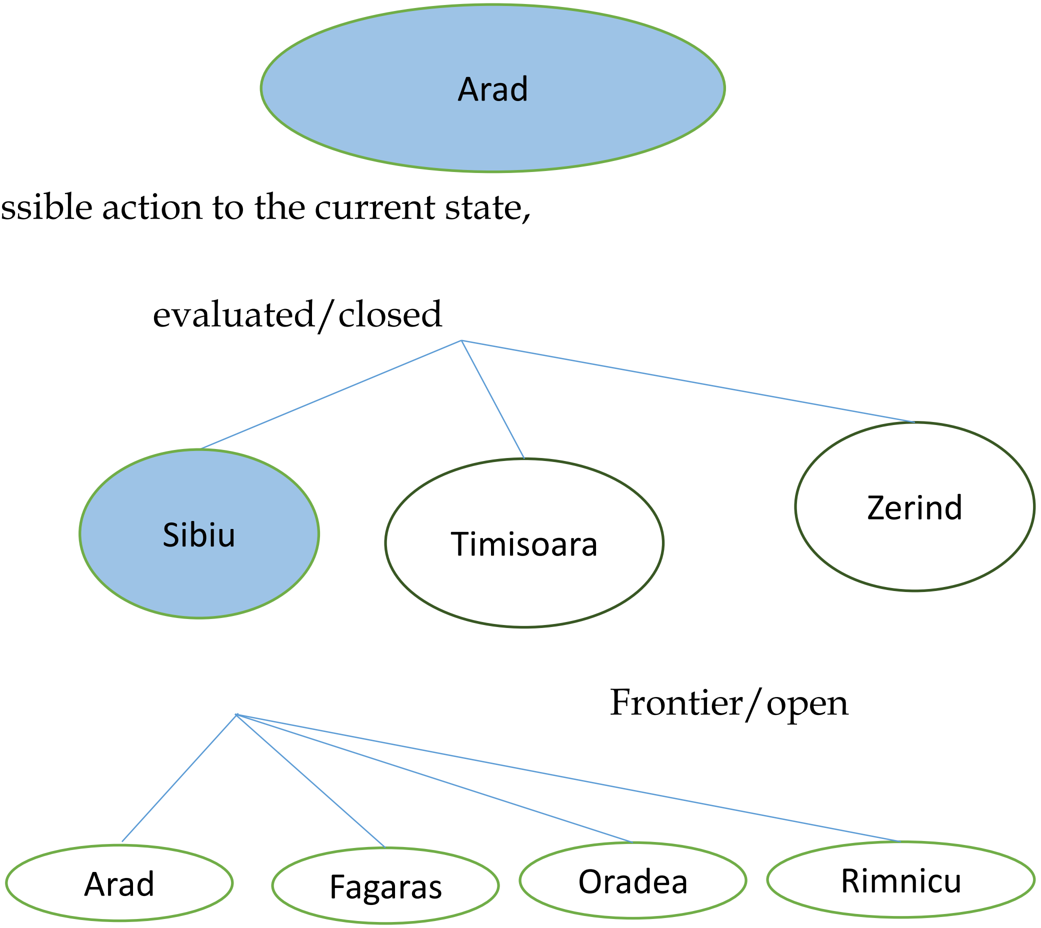
4. Assume we choose Sibiu first

    Check – is it a goal state? (it is not)

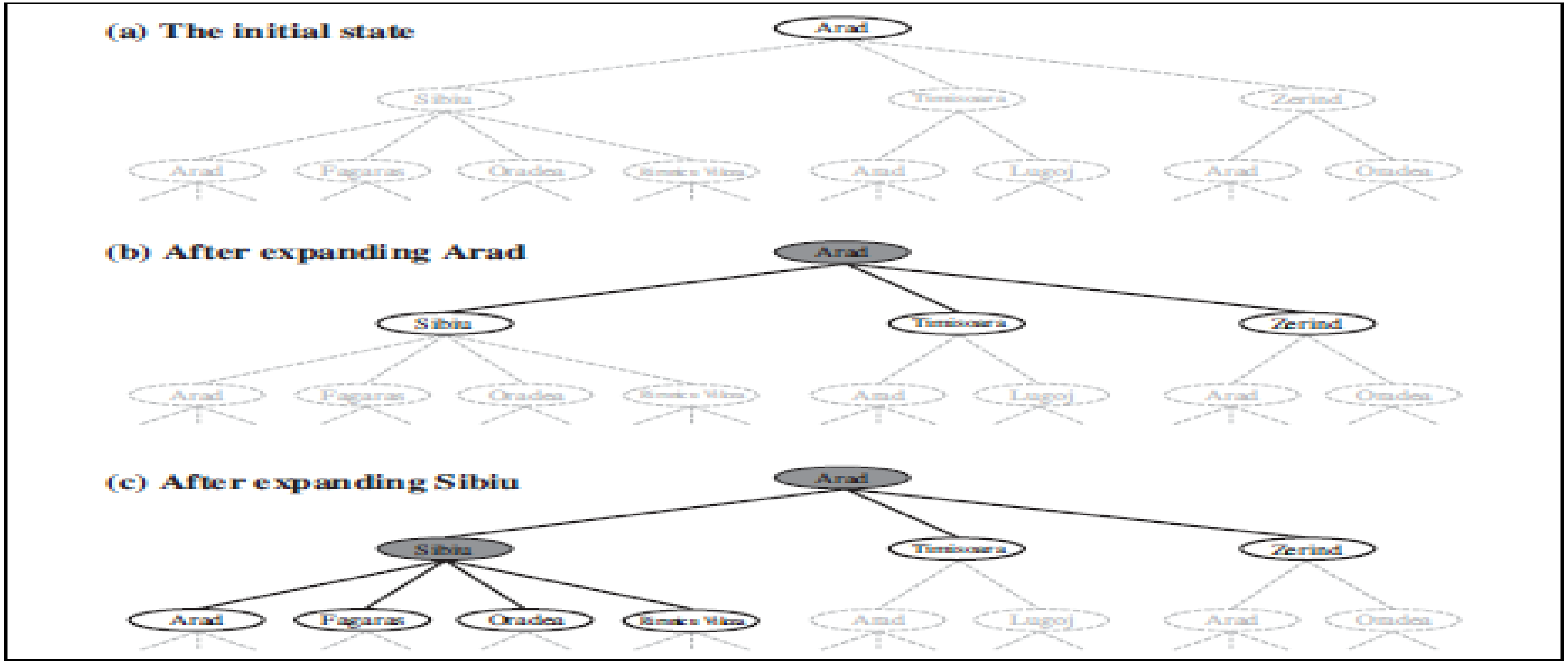
    Then expand In(Sibiu) – you get

    { In(Arad), In(Fagaras), In(Oradea), In(Rimnicu)}

5. We can choose any of these four or go back and choose Timisoara or Zerind



Search Tree


(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu

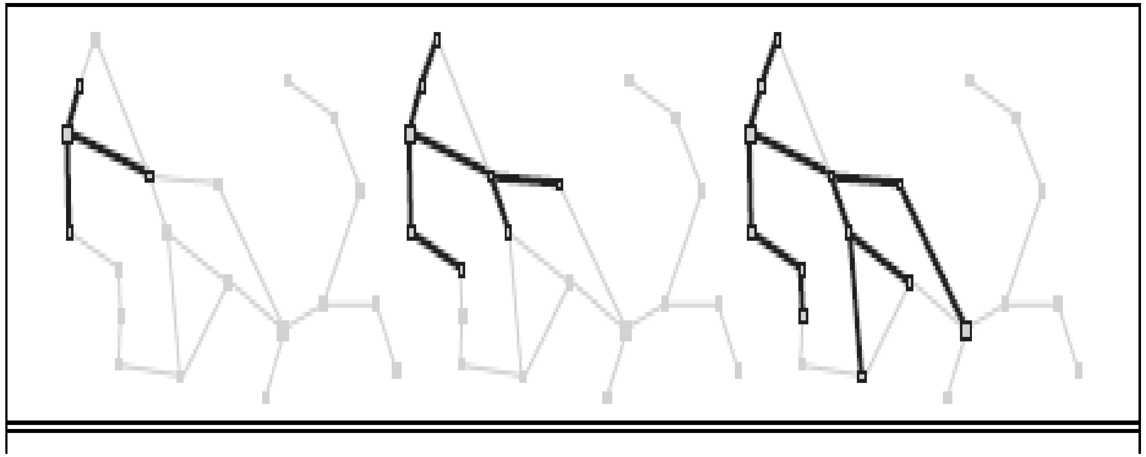
Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Route finding from Arad to Bucharest – growing the search Tree

Leaf node: a node with no children in the tree

Frontier / Open List: The set of all leaf nodes available for expansion at any given point

The process of expanding nodes on the frontier continues until either a solution is found, or there are no more states to expand.

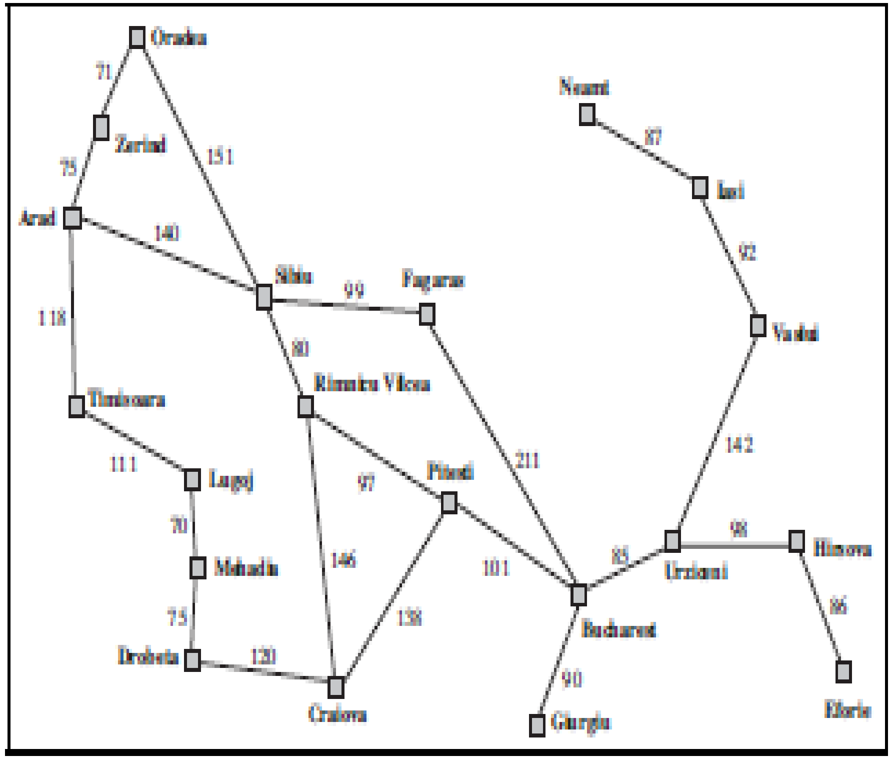


A sequence of search trees generated by a graph search on the Romania problem.

At each stage, we have extended each path by one step.

at the third stage, the northernmost city (Oradea) has become a dead end

both of its successors are already explored via other paths.



Loop Paths, Redundant Paths

Arad → Sibiu → Arad - loop – path - repeats a state.

Loop – paths make search tree infinite

Loop – paths a special case of redundant paths

Redundant paths: more than one path to get from one state to another state

ex: Arad → Sibiu (140 km), Arad → Zerind → Ordea → Sibiu (297 km)

Define problem itself so as to eliminate redundant paths

8 Queens Problem

Queen can be placed in any column → each state with n Queens placed can be reached by n! different paths

Each new Queen can be placed in the left most empty Column → each state can be reached through only one path!

Redundant paths are unavoidable in problems where actions are reversible

route finding problems, slide bar puzzles

Avoiding exploring redundant paths - remember where one has been

add explored set/closed set to tree search algorithm, to remember every expanded node

discard new nodes generated that match the previously generated nodes (in explored, frontier sets) (graph search)

Tree and Graph Search Algorithms – handling repeated states

```
function TREE-SEARCH(problem) returns a
        solution, or failure
    initialize the frontier using the initial state of
                                            problem
    loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the
        corresponding solution
    expand the chosen node, adding the resulting
    nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a
        solution, or failure
    initialize the frontier using the initial state of
                                            problem
    initialize the explored set to be empty
    loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the
        corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting
    nodes to the frontier  -   only if not in the frontier
    or explored set
```

Search Algorithms - basics

Data structure to keep track of the search Tree

Each node

- state – state in state space to which the node corresponds to
- parent – node in the search tree that generated this node
- action – action parent node took to generate this node
- path-cost – cost g(n) of the path from initial state to the node, as indicated by the parent pointers

```
Function Child-Node ( problem, parent, action) returns a child node
        return a node with
        state = problem. Result ( parent, state, action)
        parent = parent, action = action
        path-cost = parent. Path-cost + problem. Step-cost(parent, state, action)
```

Parent pointers string the nodes together into a tree structure, which allows the solution path to be extracted when a goal node is found

Search Algorithms - basics

Frontier / Open set -

set of nodes to be explored

search algorithm should be able to choose easily the next node to expand according to its preferred **strategy**

appropriate data structure – **QUEUE**

Operations on Queue -

Empty?(Queue) returns TRUE only if there are no elements in the queue

Pop (Queue) removes the first element of the queue and returns it

Insert (Element, Queue) inserts an element and returns the resulting queue

Queue –

characterized by the order in which the store the inserted nodes

Variants –

FIFO -	pops the oldest element of the queue
LIFO (Stack) –	pops the newest element of the queue
Priority Queue -	pops the element of the queue with highest priority according to some ordering function

Search Algorithms - basics

Explored set/closed set

- requires efficient checking for repeated states
- can be implemented with a hash table

Equality between states

- hash table needs to know
- visited cities - {Bucharest, Urziceni, Vaslui} is same as {Urziceni, Vaslui, Bucharest}
- sorted set?

Measures for Problem Solving Performance

- completeness – is the algorithm guaranteed to find a solution, when there is one?
- optimality – does the strategy find an optimal/ lowest cost solution?
- time complexity - how long does it take to find a solution?
- space complexity - how much memory is needed to perform the search?

Search Algorithms - basics

Time, space complexity measure

- state space graph size? |V| + |E| ? – not always feasible to know state space explicitly
- known states are – initial state, actions, transition model, goal state – state space maybe infinite
- measure in terms of
    - branching factor, **b** – maximum number of successors of any node
    - depth of shallowest goal node, **d** – number of steps along the path from the root to the goal node
    - maximum length of any path in the state space, **m**

Time

- measured in terms of the number of nodes generated during the search

Space

- measured in terms of maximum number of nodes stored in the memory

Effectiveness of a search algorithm

- Total cost = search cost + path cost of the solution found
- search cost – time complexity /and space complexity

Types of searches

Uninformed search strategies / blind searches

- strategies have no additional information about the states, beyond that provided in the problem definition
- All they do is generate successors and distinguish a goal state from a non-goal state
- all search strategies are distinguished by the order in which the nodes are expanded
- BFS, DFS, DLS, UCS,

Informed search strategies / heuristic search strategies

- strategies that know whether one non-goal state is ‘more promising’ than another

Breadth First Search

Root node is expanded first, then all the successors are expanded next, then their successors, and so on

All nodes are expanded at a given depth in the search tree, before any nodes the next level are expanded

BFS uses Graph search algorithm - shallowest unexpanded node is chosen for expansion –

achieved through FIFO queue for frontier

new nodes, deeper than their parents go to the back of the queue,

old nodes, shallower get expanded first

discards any new path to a state already in the frontier/open or explored/closed set

any such path must be at least as deep as the one already found, implies BFS always has the shallowest path to every node in the frontier

+ goal test applied to each node when it is generated, rather than when it is selected for expansion

Breadth First Search

BFS performance

Complete– if the shallowest goal node is at some finite depth d, BFS will eventually find it, after generating all shallow nodes (if b is finite)

Optimal – shallowest goal node is the optimal one, if all actions have the same cost/ path cost is non- decreasing function of depth (else not necessarily optional)

Time complexity - with a branch factor b and goal node at depth d, number of nodes generated are

$b + b^2 + b^3 + \ldots + b^d = O(b^d)$

if the algorithm were to apply goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be generated before the goal was detected, then

time complexity is $O(b^{d+1})$

Space Complexity $O(b^d)$

general graph – search stores every expanded node in explored/closed set - $O(b^{d-1})$

BFS – every generated node remains in memory

Explored/closed set - $O(b^{d-1})$

Frontier/Open set - $O(b^d)$

Using a tree search, instead of graph-search would not save much space and could cost a lot of time due to redundant paths

BFS Algorithm

```
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 )
```

BFS Algorithm

```
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 )
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier - only if not in the frontier or explored set
```

BFS

Exponential time and space complexity - O(b^d)

branching factor, b:	10
speed of computer:	1 million nodes /sec
memory for 1 node:	1000 bytes

Depth, d	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

Memory requirements are a bigger problem, while time is still a major factor

kilo, K 1000 mega, M = 1000K giga, G = 1000M Tera, T = 1000G peta, P = 1000T exa, E = 1000P

Exponential complexity problems can not be solved by uninformed search methods, but for small instances

Uniform Cost Search Approach

BFS optimal when all step costs are equal. For different step costs?

Instead of expanding the shallowed node, expand the node with lowest path cost g(n)

store nodes in the frontier in the order of priority of path cost - priority queue

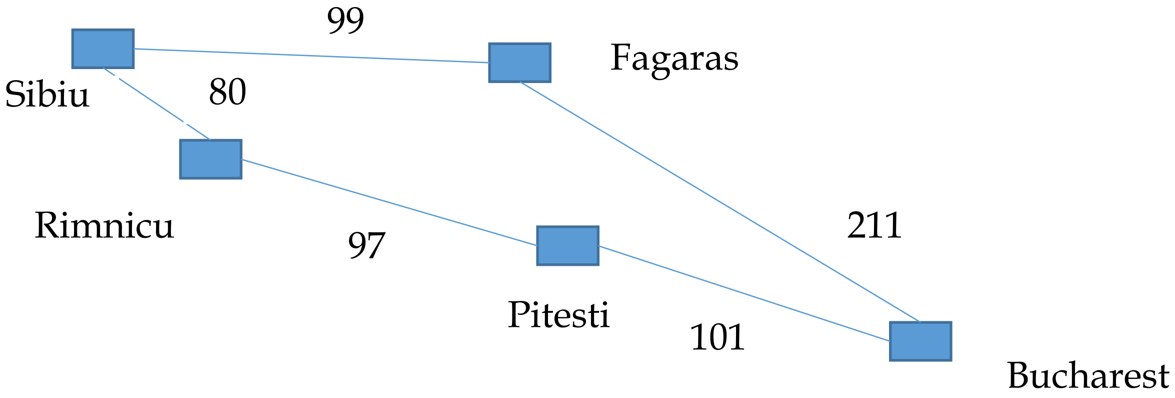
Differences from BFS

ordering frontier by path cost

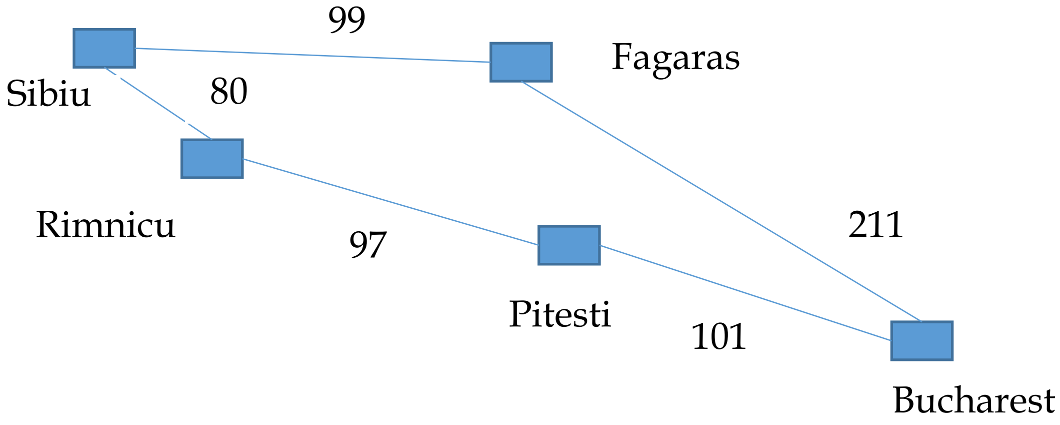
goal test is applied to a node when it is selected for expansion(similar to general graph-search), rather than when it is first generated

(as the first goal node that is generated may be on a sub-optimal path)

a test is added in case a better path is found to a node currently on frontier



Uniform Cost Search Approach

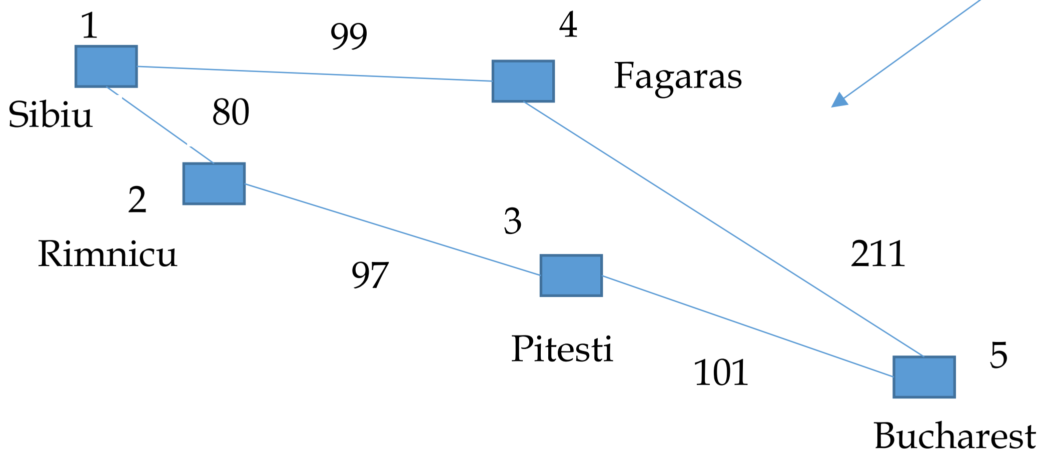


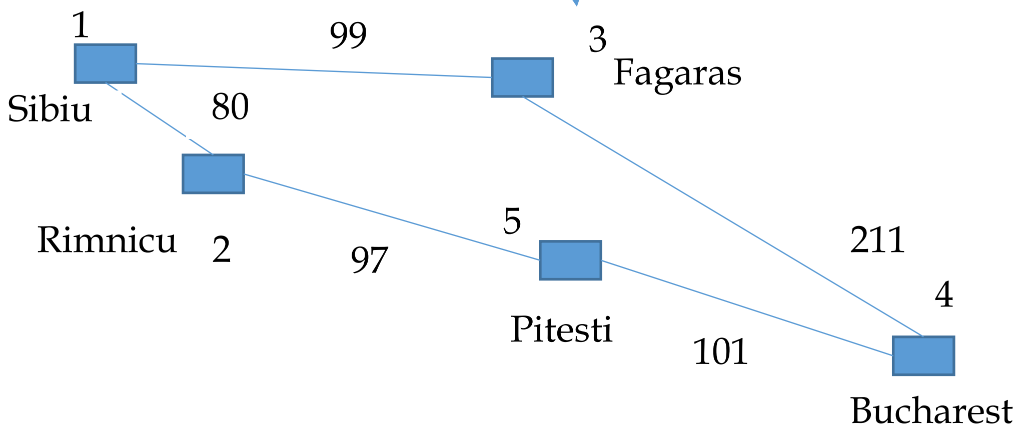
BFS	UCS
Frontier (R80, F99)	Frontier (R80, F99)
Frontier (F99, P97)	Frontier (R80 +P97, F99)
Frontier (P97, B211)	Frontier (**F99**, RP177)
Frontier (B211)	Frontier (**RP177**, F99+ **B211**)
	Frontier (**RP177+B101**, FB310)
	Frontier (**RPB278**, FB310)

UCS: goal node generated 1-2-3-4 -5

adds a second path 1-2-5-4

checks if the second path is better and returns it





Uniform Cost Search Approach

```
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
```

Data structure for frontier needs to support efficient membership testing

combine capabilities of a priority queue and a hash table

Uniform Cost Search Approach

```
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
```

```
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 )
```

UCS Performance

- Optimal in general
  - UCS expands paths nodes in order of their optimal path cost
  - first goal node selected for expansion must be the optimal solution
  - Whenever UCS selects a node for expansion, the optimal path to that node has been found
  - Step costs are non-negative; path never gets shorter as nodes are added
  - number of steps in a path does not matter, path cost matters
  - UCS will get stuck in infinite loop if there is a path with an infinite sequence of zero cost actions
    - A sequence of No Ops
- Completeness is guaranteed
  - provided the cost of each step exceeds some small positive constant ε
- UCS is guided by path costs rather than by depth, d
  - Assuming C* as the cost of the optimal solution
  - UCS time, space complexity in worst case - $O(b^{1+C^*/e})$ - can be much greater than $O(b^{d+1})$

- When all step costs are same, UCS is similar to BFS except
  - BFS stops as soon as it generates a goal, UCS examines all the nodes at the goals depth to see if one has lower cost
  - UCS does more work by expanding nodes at depth, d unnecessarily

Depth First Search

Expands deepest nodes in the frontier of the search tree

Search proceeds to the deepest level of the search tree where the nodes have no successors

As the nodes are expanded they are dropped from the frontier,

search backs up to the next deepest node, that still has unexplored successors

DFS using Graph search algorithm

/can be implemented with recursive function that calls itself on each of its children in turn

LIFO Queue/ Stack

most recently generated node is chosen for expansion

DFS is complete as Graph Search algorithm avoids repeated nodes and redundant paths

DFS using Tree Search Algorithm is not complete (infinite loops)

with a check for new states against those on the path from root node to the current node – avoids infinite loops infinite spaces , but does not avoid redundant paths

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

ex: DFS for Knuth's conjecture – 4 problem, DFS would keep applying ! Operator forever

Depth First Search

DFS both versions Non- Optimal

- Will explore entire left sub tree even if a shallower node is a goal node
- if a deeper node is also a goal node, DFS will return it as a solution, instead of the shallower one (a better solution)

Time Complexity

- bounded by the size of the state space.
- may generate all of the O(b^m) nodes in the search tree
- (m – maximum depth of any node >> d, depth of shallowest goal node)

Space Complexity

- for Graph search similar to BFS
- for Tree Search requires storage of O(bm) nodes

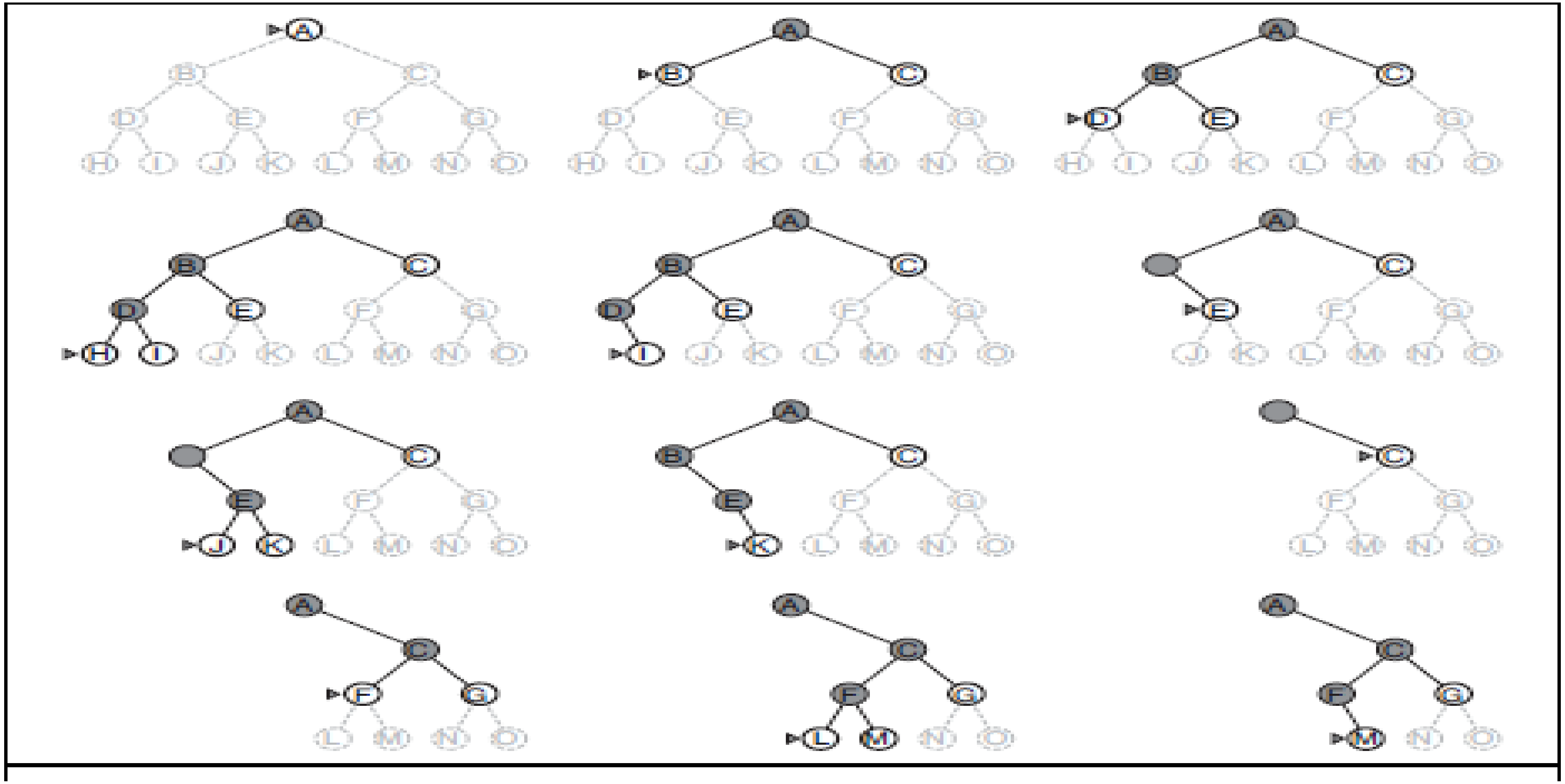
DFS at d = 16, b = 10 requires 156 k bytes, instead of 10 exabytes needed by BFS

DFS

Depth-first search on a binary tree. The unexplored region is shown in light gray.

Explored nodes with no descendants in the frontier are removed from memory.

Nodes at depth 3 have no successors and M is the only goal node.



Depth Limited Search

Back tracking variation for DFS:

only one successor is generated at a time rather than all successors, each partially expanded node remembers which successor to generate next.

Takes lesser memory, O(m) compared to DFS - O(bm)

Depth Limited search:

DFS fails in infinite state space. Hence predetermine depth limit, l (no further expansion)

solves infinite problem, but introduces incompleteness, if **l < d**

(shallowest goal is beyond the depth limit, when d is unknown)

non- optimal, if **l > d**

DFS is DLS with l = infinity

Time complexity - O(b^l)

Space Complexity – O(bl)

Failure conditions – DLS can terminate with

no solution

no solution within depth limit

L can be based on knowledge of the problem.

Ex: map of Romania has 20 cities, l can be 19

Study of map shows any city can be reached from any other city in at most 9 steps. L = 9.

Diameter of the state space – leads to efficient DLS

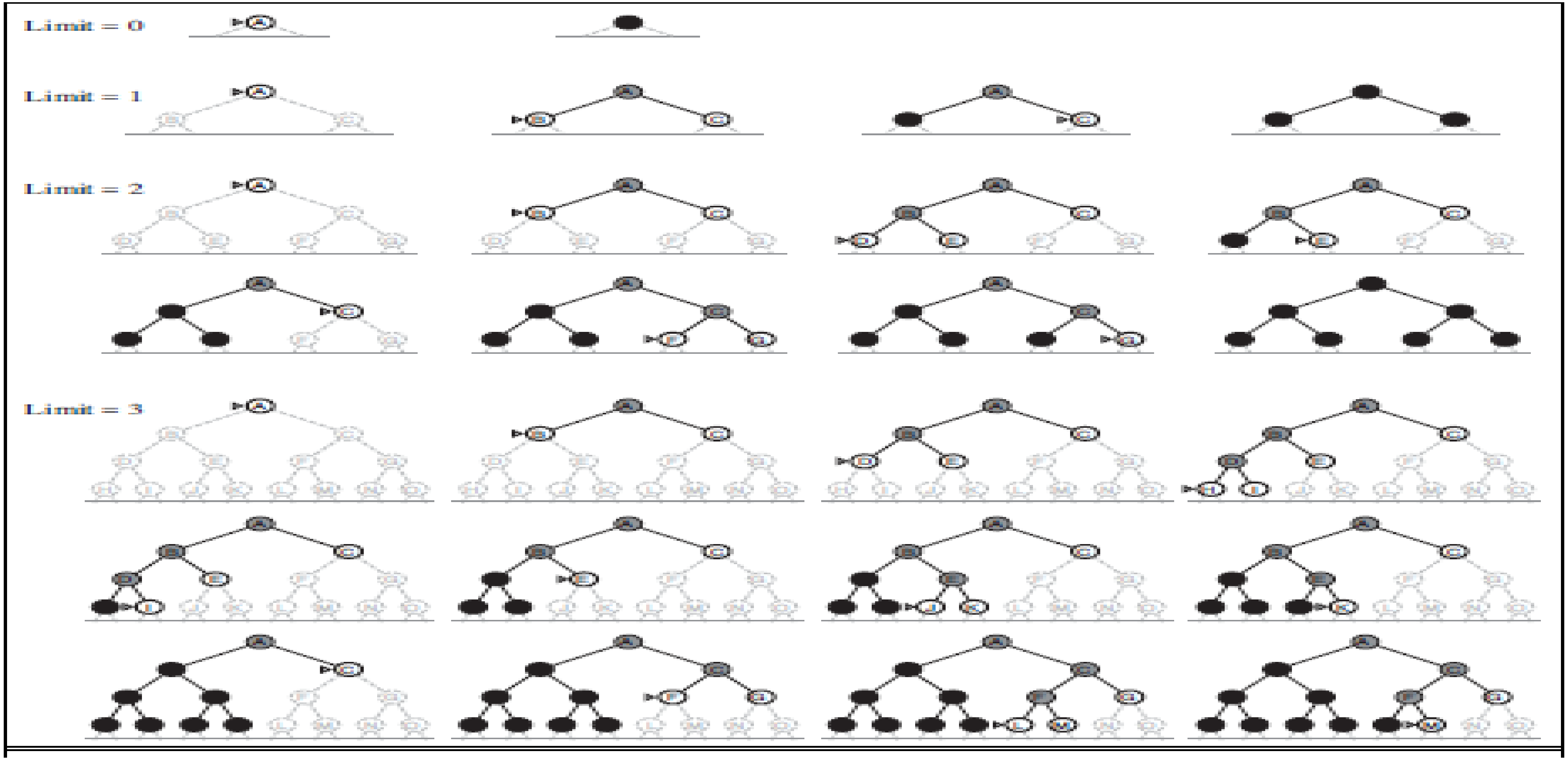
DLS – Recursive implementation

```
function DEPTH-LIMITED-SEARCH(problem, limit ) returns a solution, or failure/cutoff
        return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit )

function RECURSIVE-DLS(node, problem, limit ) returns a solution, or failure/cutoff
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        else if limit = 0 then return cutoff
        else
        cutoff occurred?←false
        for each action in problem.ACTIONS(node.STATE) do
        child ←CHILD-NODE(problem, node, action)
        result ←RECURSIVE-DLS(child , problem, limit − 1)
        if result = cutoff then cutoff occurred?←true
        else if result = failure then return result
        if cutoff occurred? then return cutoff else return failure
```

IDS - Iterative Deepening Search

Four iterations of iterative deepening search on a binary tree.



Iterative Deepening Search

IDS - combination of DFS and BFS

finds best depth limit, through gradual increase of limit

Space complexity - O(bd) like DFS

Complete – like BFS when b is finite

Optimal, when the path cost is non- decreasing function of the depth of the node

States are generated multiple times -

nodes at the bottom level generated once, those on next to bottom level generated twice, and so on, up to children of the root, which are generated d times.

total number of nodes generated in the worst case:

N(IDS) = d(b) + (d-1) b^2 + … + (1) b^d -- Complexity O( ) – asymptotically same as BFS

Ex: for b = 10 , d = 5

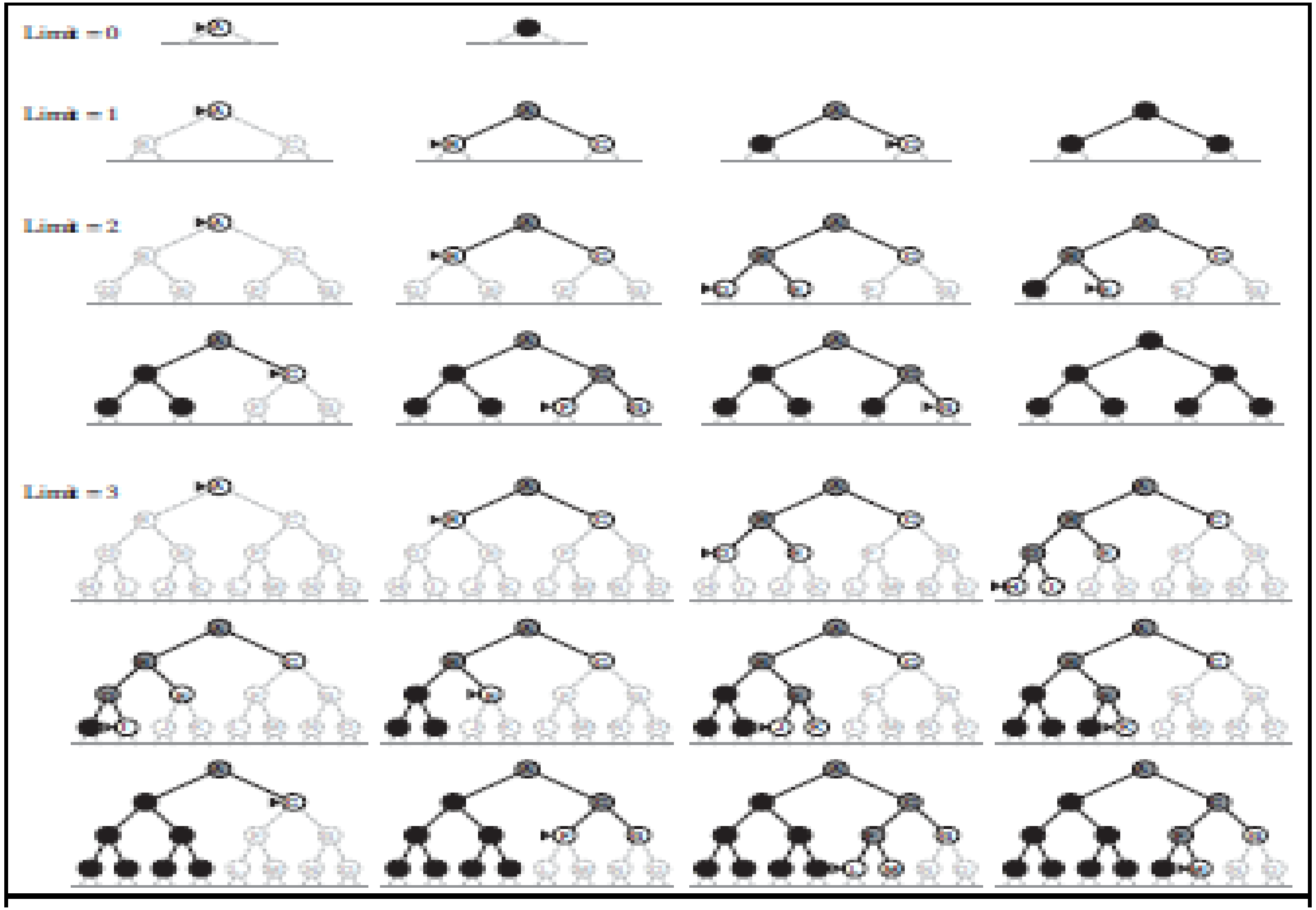
N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450

N(BFS) = 10 + 100 + 1000 + 10000 + 100000 = 111,110

IDS preferred un-informed search when search space is large and depth of solution is not known

IDS

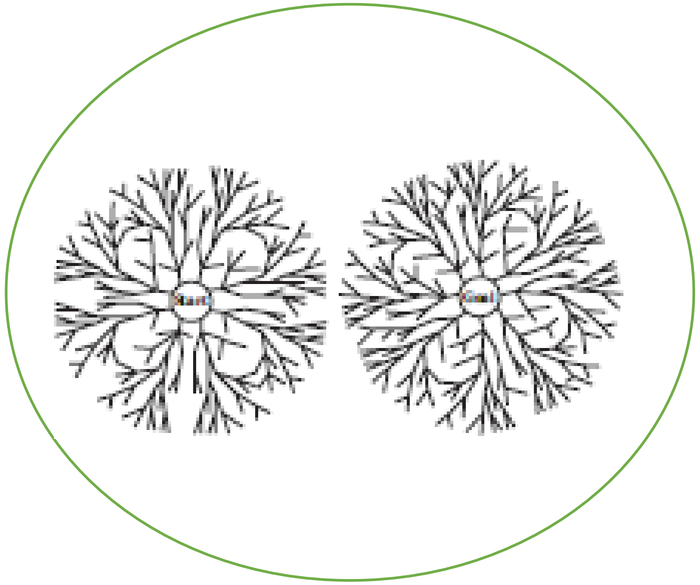
4 iterations of IDS



IDS

```
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
```

Bi Directional Search - BDS



Run two simultaneous searches

one in forward direction from initial state

the other backward from goal state

The two searches are expected to meet in the middle

$b^d >> b^{d/2} + b^{d/2}$

Goal test

replaced by a check to see whether the frontiers of the

two searches intersect

first such solution may not be optimal even if both searches are BFS. Additional search to make sure there isn’t another short cut across the gap, would be needed

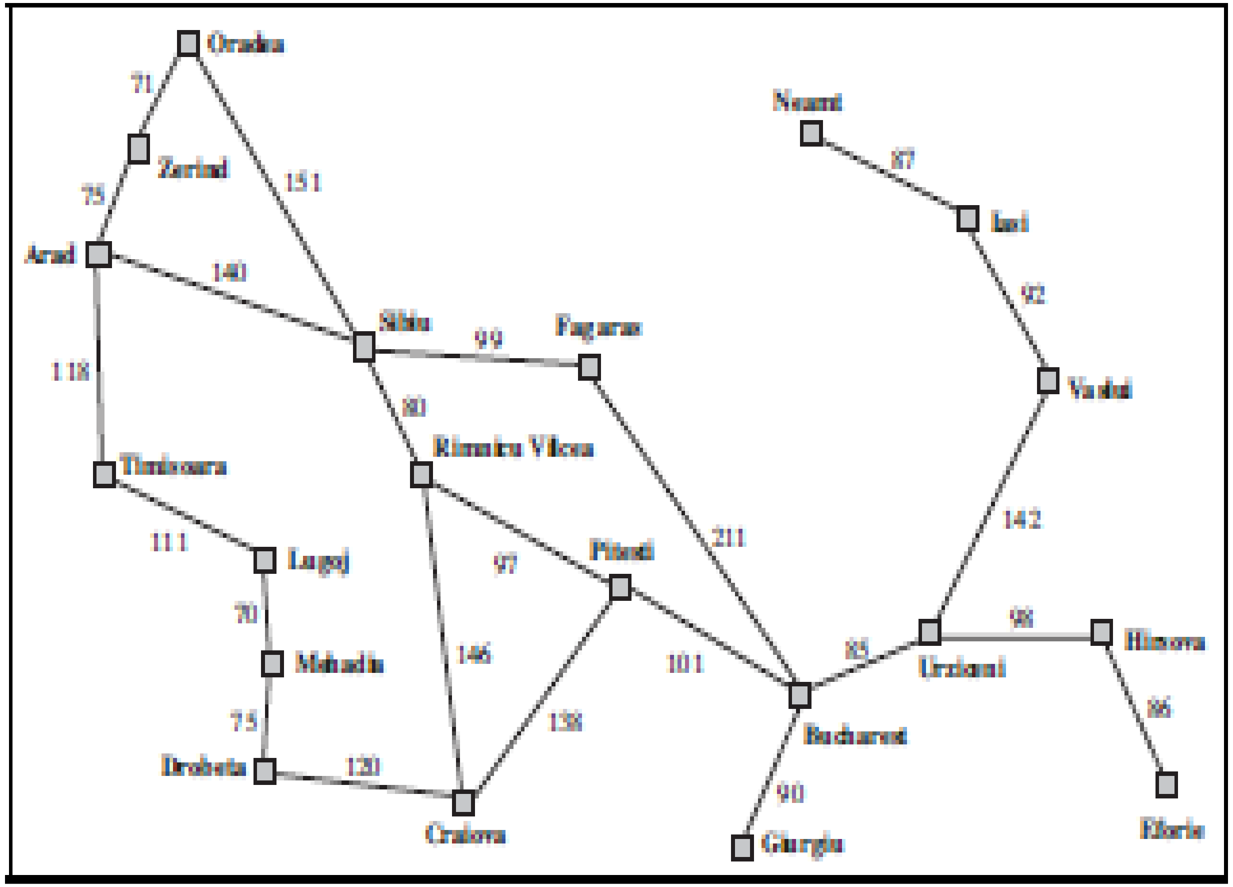
Ex: d = 6, BFS in each direction. In the worst case, the two searches meet when they have generated all the nodes at d =3. for b = 10

N(BDS) = 2(10+100+1000) = 2220

N(BFS) = 10^6 = 1111110

Time complexity for BDS using BFS - $O(b^{d/2})$; Space complexity - $O(b^{d/2})$

Example – Driving In Romania



Goal: Drive to Bucharest

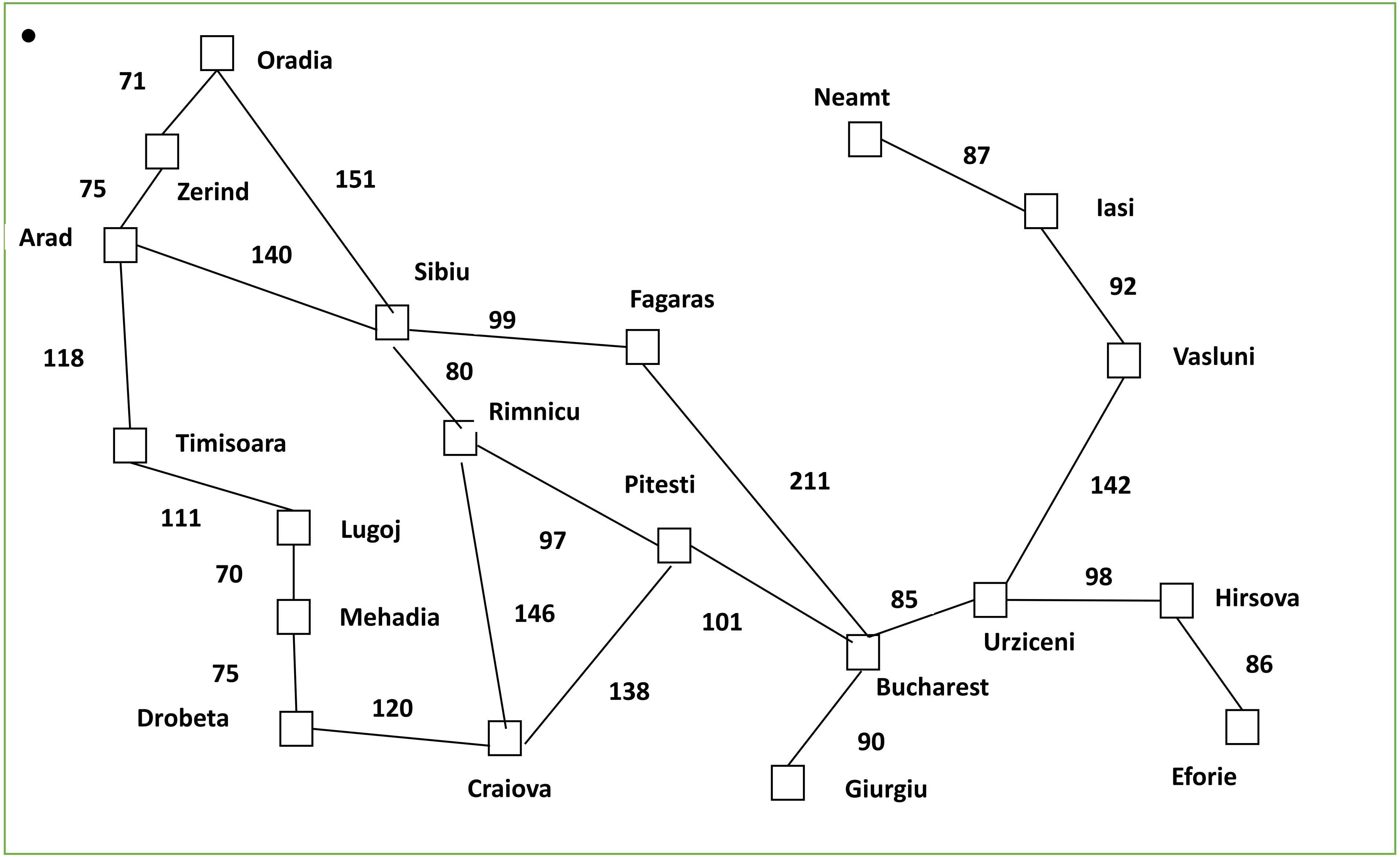
Current State: Arad

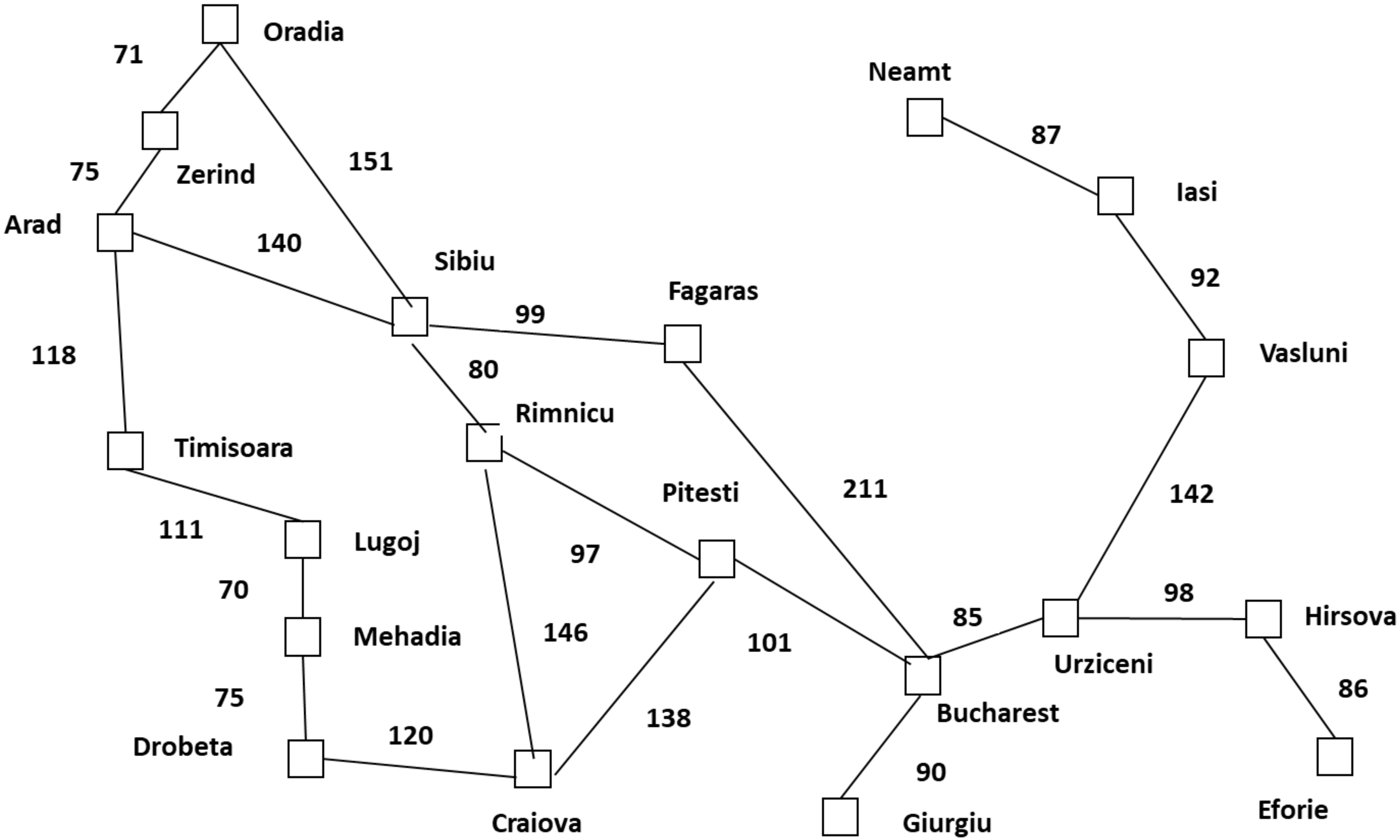
3 roads from Arad

Arad →Zerind
Arad → Sibiu
Arad → Timisoara

None achieve the goal!

Example – Driving In Romania




Oradia
71
151
Zerind
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu
Timisoara
111
Lugoj
70
Mehadia
75
Drobeta
120
Craiova
146
97
Pitesti
138
101
211
Bucharest
90
Giurgiu
85
Urziceni
98
Hirsova
86
Eforie
142
Vasluni
92
Iasi
87
Neamt

BDS

How to do bidirectional search?

Method for computing predecessors?

When all actions are reversible, predecessors are successors (ex : slide bar puzzle, Romania map)

One goal state – backward search is very much like forward state.

Comparing uninformed search strategies (using tree search)

Criterion	BFS	UCS	DFS	DLS	IDS	BDS
Complete?	Yes (if b is finite)	Yes (if b is finite, step costs >= ε for positive ε	No (yes if graph search is used and state space is finite)	No	Yes (if b is finite)	Yes (if b is finite, if both directions use BFS)
Time	$O(b^d)$	$O(b^{1+C^*/e})$	$O(b^m)$ (for graph search, bound by size of state space)	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+C^*/e})$	O(bm) (for graph search, bound by size of state space)	O(bl)	(O(bd)	$O(b^{d/2})$
Optimal?	Yes (If step costs are all identical)	Yes	No	No	Yes (If step costs are all identical)	Yes (If step costs are all identical, if both directions use BFS)

example

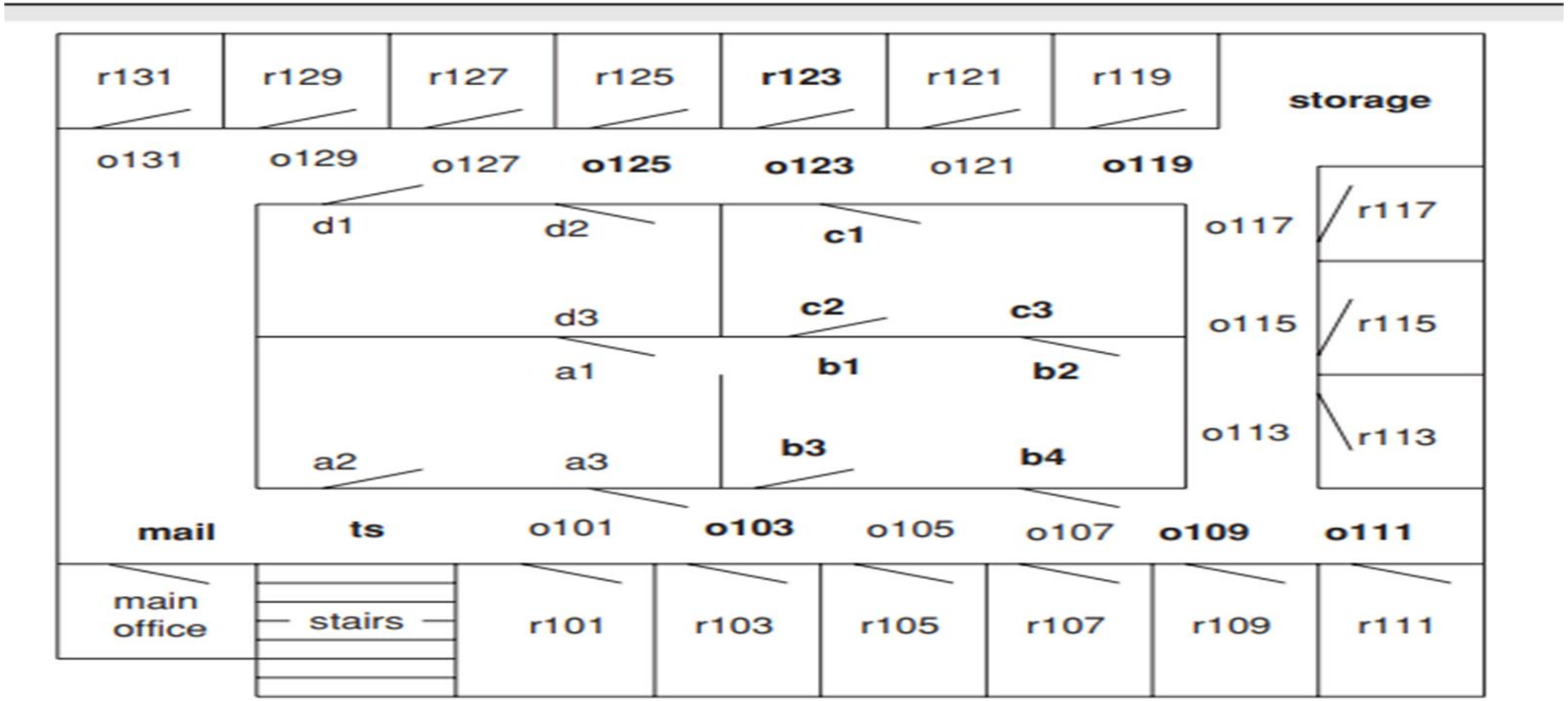
Consider the problem of finding a path in the grid shown in Figure below from the position s to the position g. A piece can move on the grid horizontally or vertically, one square at a time. No step may be made into a forbidden shaded area.

What is the first path found using a depth-first search from 's' to 'g', given that the order of the operators is up, left, right, and down. Assume there is cycle pruning/avoidance.

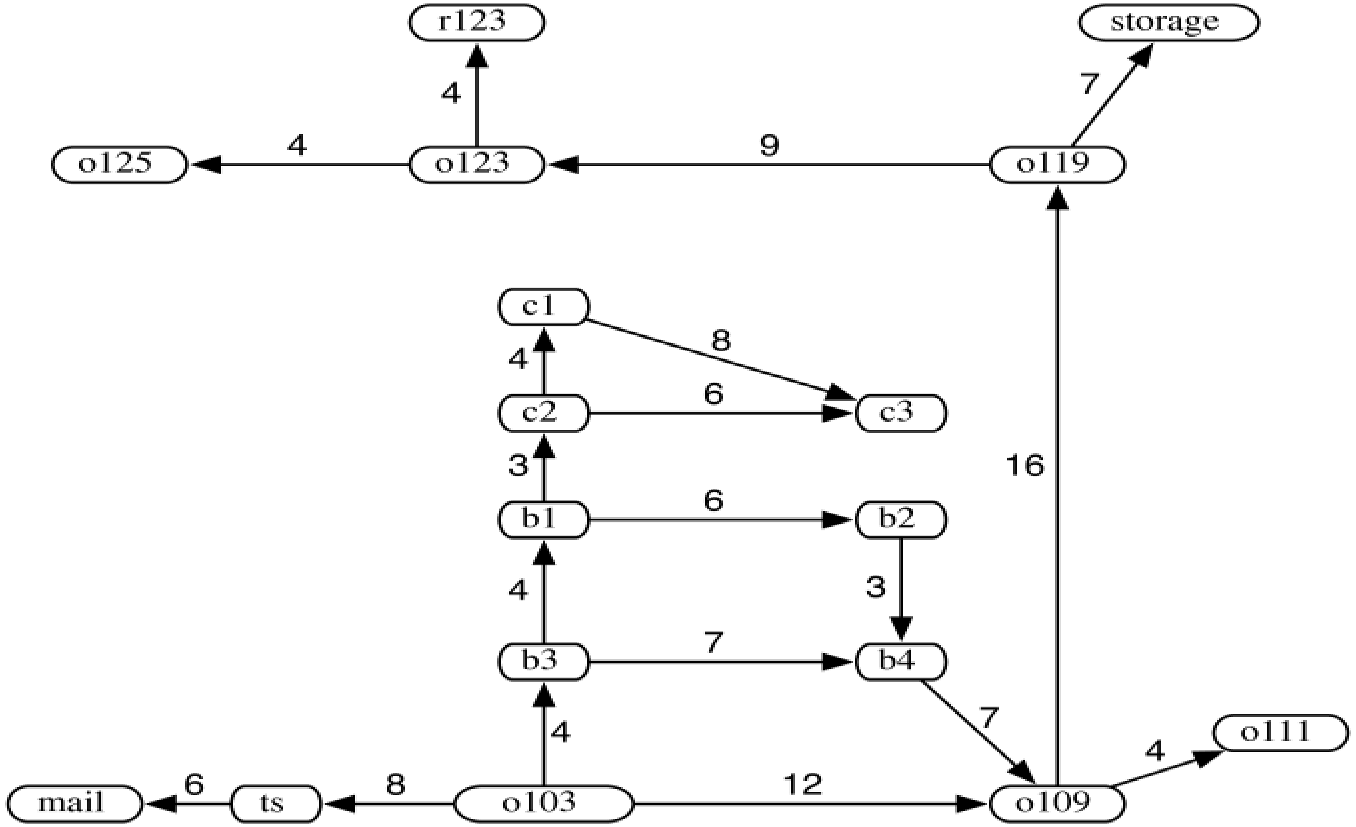
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	**g**	20	21	22	23	24
25	26	■	■	■	■	■	32
33	■	35	36	37	38	39	40
41	42	■	**s**	45	46	■	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Example

cc


r131
r129
r127
r125
r123
r121
r119
storage
o131
o129
o127
o125
o123
o121
o119
r117
o117
d1
d2
c1
d3
c2
c3
o115
r115
a1
b1
b2
o113
r113
a2
a3
b3
b4
mail
ts
o101
o103
o105
o107
o109
o111
main office
stairs
r101
r103
r105
r107
r109
r111

example


r123
storage
4
7
o125
4
o123
9
o119
c1
8
4
c2
6
c3
16
3
b1
6
b2
4
3
b3
7
b4
7
4
4
o111
mail
6
ts
8
o103
12
o109