MONASH University
Information Technology

FIT3080

Solving Problems by Searching

Reading, this week's activities, etc.

Reading for this week:

- Russell and Norvig (4th edition), chapter 3, 3.1 – 3.4

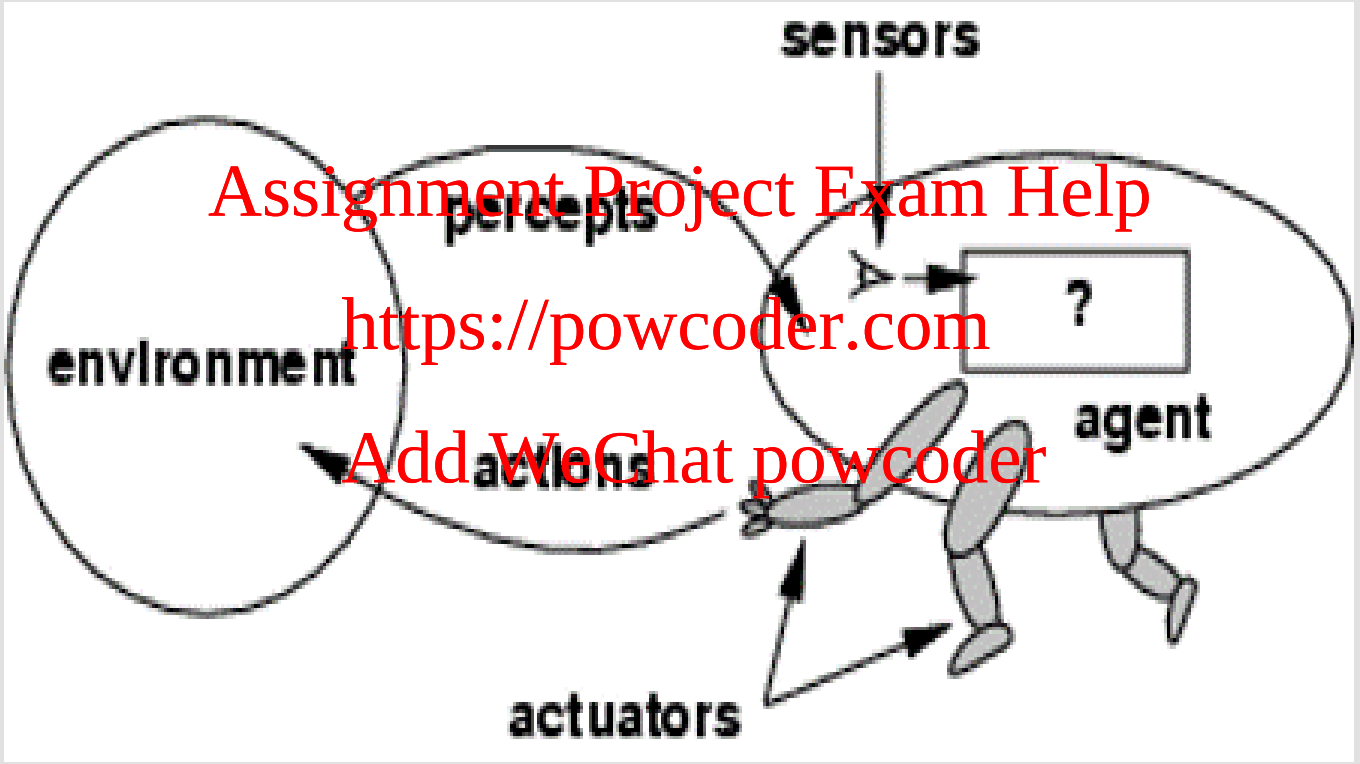
This week's activities:

- Labs / tutorials start this week

Other notes and reminders:

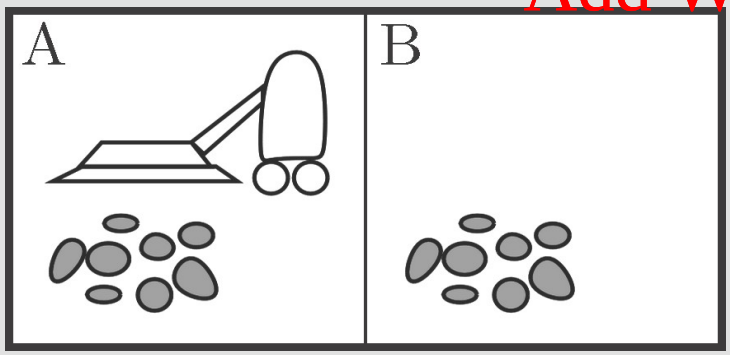
- Hurdle requirements (to pass)
- Academic integrity policies and consequences
  - > Academic integrity practice quizzes
- Special consideration policies and process

Last week: Agent Program

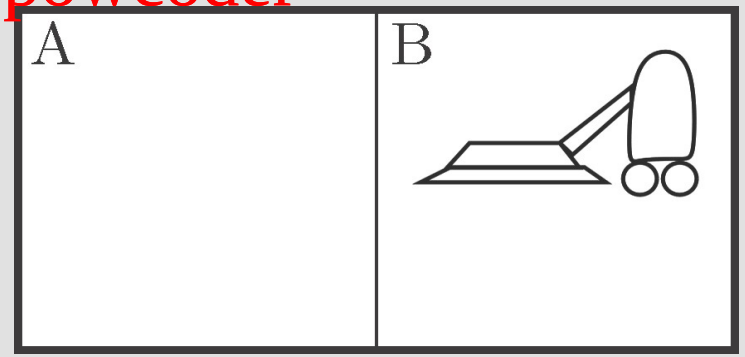


Last week: Vacuum-cleaner World and Agent

- **Percepts**: location and contents - e.g., [A, Dirty] or [B, Clean]
- **Actions:** Left, Right, Vacuum
- **Program:**
  - **if** status = Dirty **return** Vacuum
  - **else if** Location = A **return** Right
  - **else if** Location = B **return** Left



Initial state (of the environment)

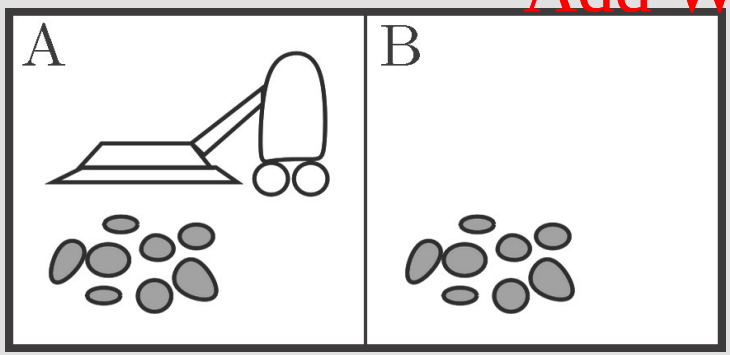


Desired state

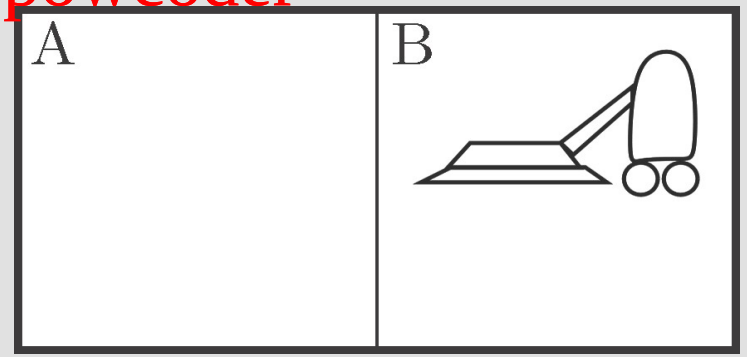
This week: Search

- **Percepts**: location and contents - e.g., [A, Dirty] or [B, Clean]
- **Actions:** Left, Right, Vacuum
- **Program:** Find a sequence of actions to achieve a desired state: e.g., [Vacuum, Right, Vacuum]

Search algorithms are general programs that compute such action sequences. We often call these sequences **plans**



Initial state (of the environment)



Desired state

Why Search?

- Effective at solving sequential problems
- Easily integrated with domain-specific insights
- Some influential applications of search:
  - Game AI
    - RTS, FPS and RPG all use search
    - Board games (Chess, Go, etc)
  - Robotics (e.g., planning for the Mars rover)
  - Scheduling (e.g., in public transport, for manufacturing)

This week: Outline

- Basic framework of a problem-solving agent
  - Problem formulation
  - Types of state spaces
  - Tree-Search and Graph-Search frameworks
  - Fundamental search strategies
    - > Breadth-first search (BFS)
    - > Uniform-cost search (UCS)
    - > Depth-first search (DFS)
    - > Depth-limited search (DLS)
    - > Depth-First Iterative Deepening (DFID)

MONASH University
Information Technology

FIT3080 – Artificial Intelligence

Problem Formulation

Problem-solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, 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 seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Problem Formulation (I)

Comprises decisions about:

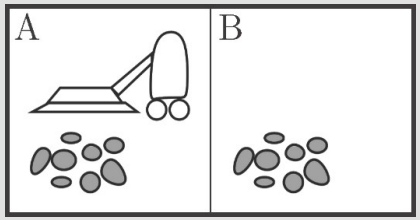
- which properties of the world matter
- how to best represent those properties
- which actions are possible
- how to best represent those actions
- cost model and objective function

Problem Formulation (II)

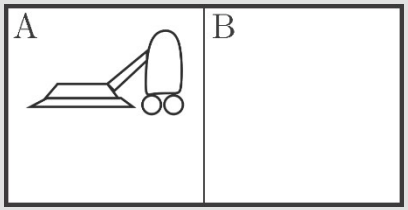
- Basic constituents: States, Goals, Actions, Constraints
- State space: the set of all states reachable from the initial state by any sequence of actions
- Path in the state space: any sequence of actions leading from one state to another
- Representing a problem
  - Initial state
  - Operators (Actions) and transition model
  - Constraints
  - Goal test
  - Path cost function
- A solution is a sequence of actions leading from the initial state to a goal state

Example: Vacuum world

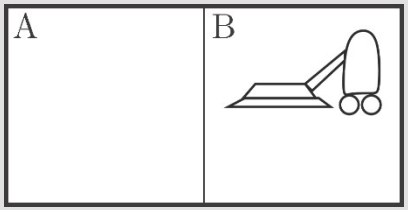
1. **Initial–State:**



2. **Formulate–Goal:**
either of



or

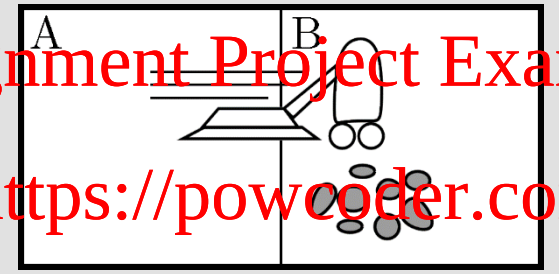


3. Formulate-Problem:

- **states**:
  (location, cleanliness, remaining dirty loc.)
- **actions**:
  {Left, Right, Vacuum}
- **transition model**:
  effect of action in state
- **path cost**:
  1 for every move

4. Solution sequence: [Vacuum, Right, Vacuum]

Example: Vacuum world



Abstracting away from unnecessary detail can drastically reduce the size of the state space

Selecting a State Space

- Real world is complex

  → state space is usually abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
- For *guaranteed realizability*, any real state must get to some real state
- (Abstract) solution = a solution that can be expanded into a set of real paths in the real world
- Each abstract action should be “easier” to perform than solving the original problem

Assumptions about the Environment

In our vacuum cleaner problem we assumed:

- Observable
- Known
- Single -agent
- Deterministic
- Sequential
- Static
- Discrete

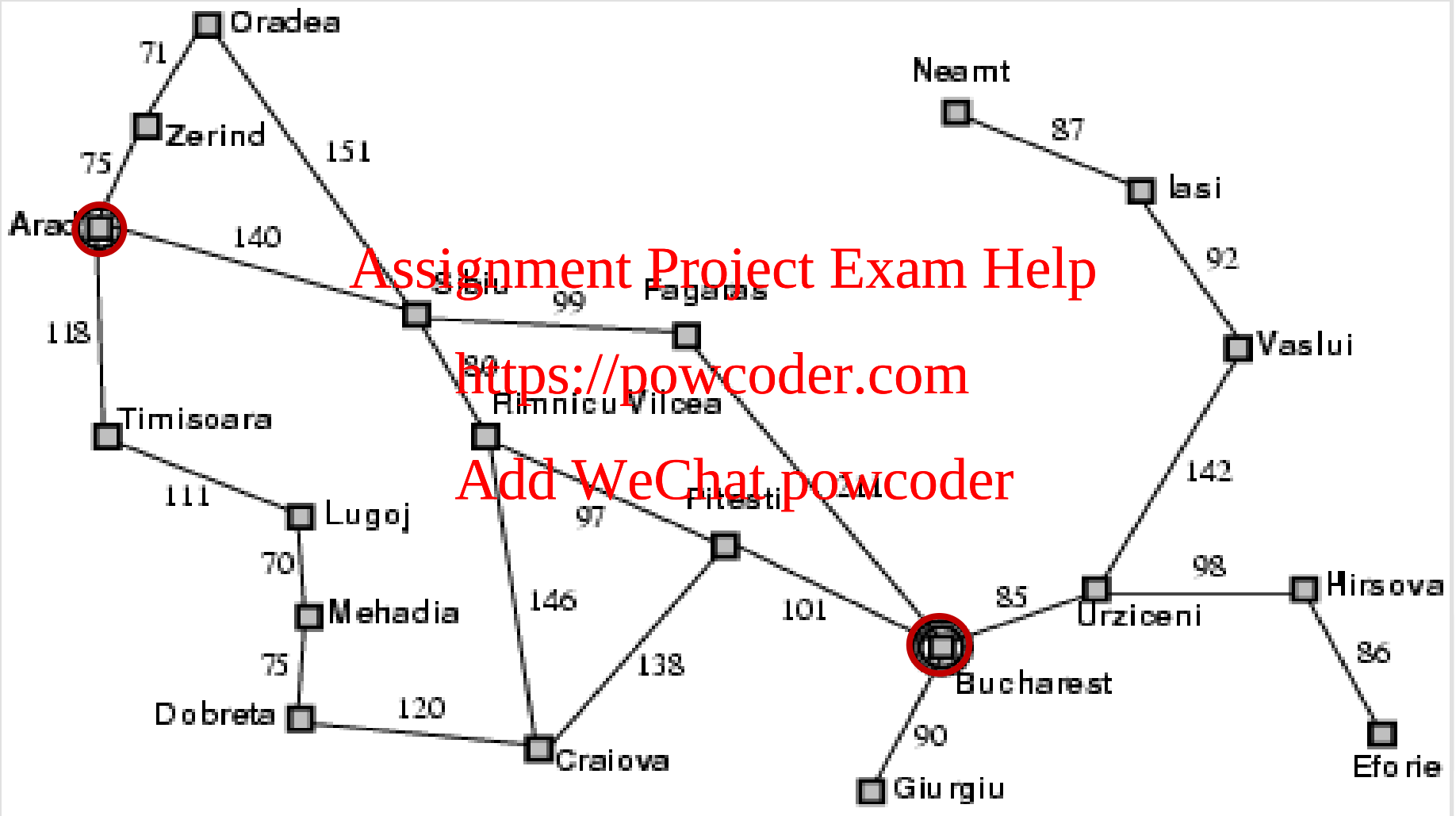
The formulation needs to reflect our assumptions!

Example: Romania

On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest.

- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Problem Formulation - Romania

1. initial state - e.g., “at Arad”
2. actions
   - e.g., {Go(Sibiu),Go(Timisoara), … }

   transition model
   - e.g., *Result(In(Arad),Go(Zerind))→In(Zerind)*
3. constraints – there are roads between cities
4. goal test can be
   - explicit, e.g., *In(Bucharest)*
   - implicit, e.g., *Checkmate(x)*
5. path cost (additive)
   - e.g., sum of distances, number of actions executed
   - *c(s,a,s’)* is the step cost of taking action *a* at state *s* to reach state *s’*, assumed to be ≥ 0

Problem Formulation – 8 Puzzle (I)

Start

5	4	
6	1	8
7	3	2

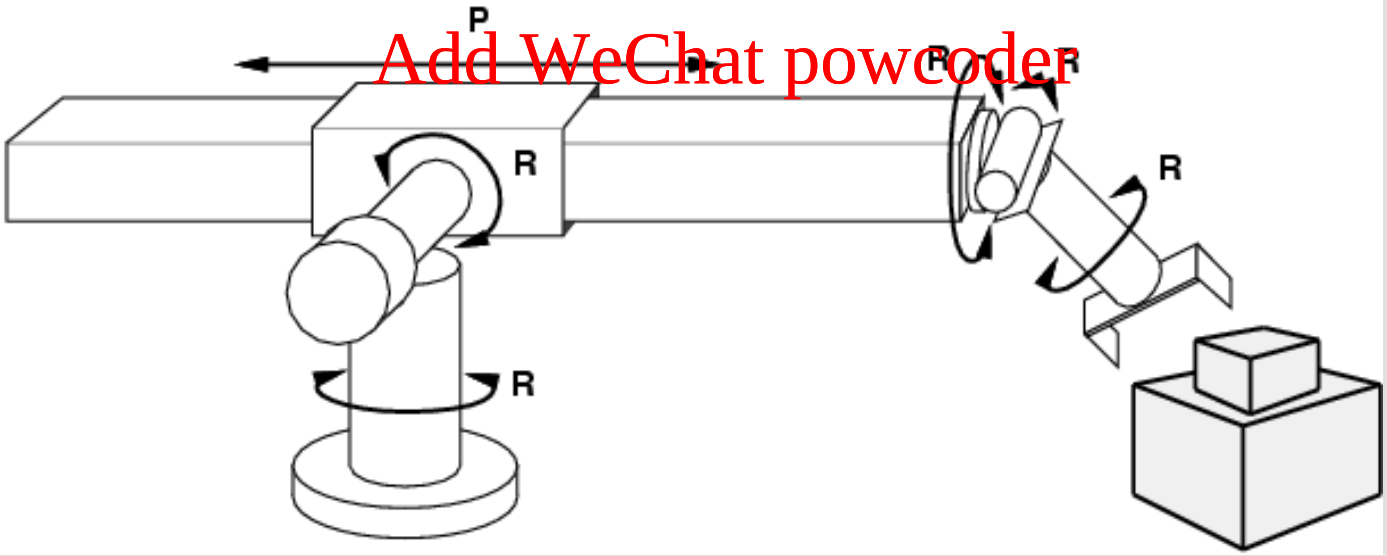
End

1	2	3
8		4
7	6	5

Problem Formulation – 8 Puzzle (II)

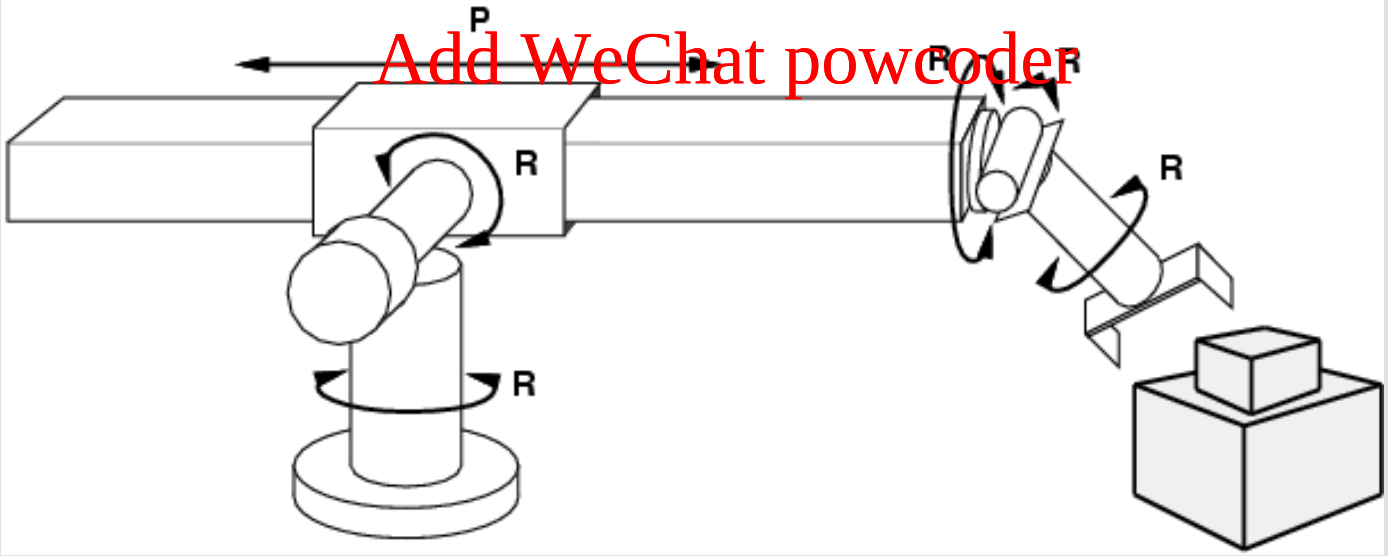
- **States**
  - Location of each of the 8 tiles in one of the 9 squares
- **Operators**
  - Possible moves of blank tile
- **Constraints**
  - A tile cannot move out of bounds
- **Goal test**
  - Have we reached the goal configuration?
- **Path cost**
  - If we want to minimize the number of steps, then cost of 1 per step

Problem Formulation: Robotic Assembly



Problem Formulation: Robotic Assembly

- states?: real-valued coordinates of robot joint angles; parts of the object to be assembled
- actions?: continuous motions of robot joints
- constraints?: arm cannot fully rotate up and down
- goal test?: complete assembly
- path cost?: time to execute



MONASH University
Information Technology

FIT3080 – Artificial Intelligence

Search Algorithms

Search Tree

- Tree – each node has at most one parent
- Root of search tree is the initial state
- Leaves are states without successors (the “fringe” or “frontier”)
- At each step, choose one leaf node to *expand*

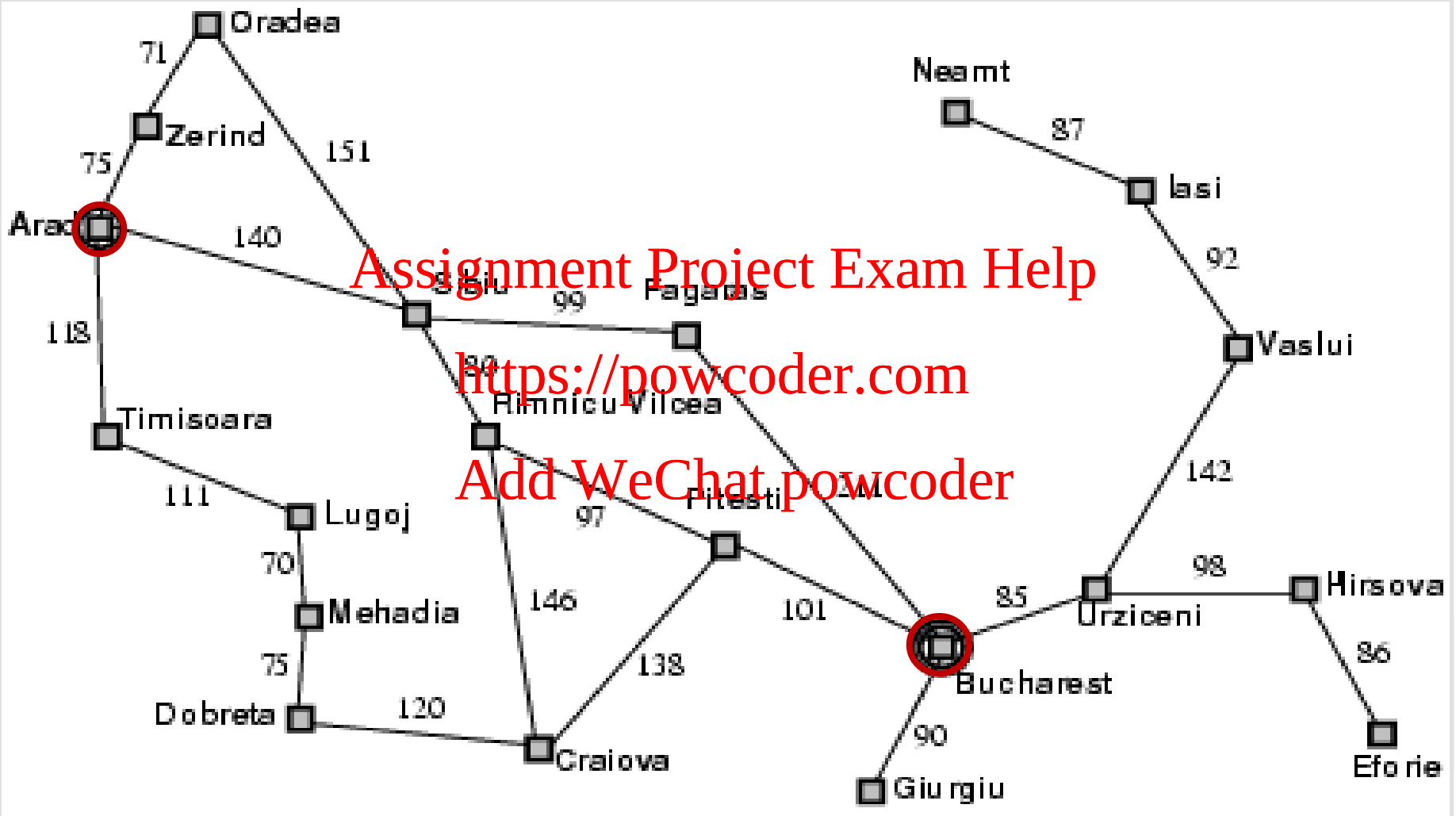
Basic Tree Search Algorithm

function TREE-SEARCH(*problem*) **returns** a solution or failure

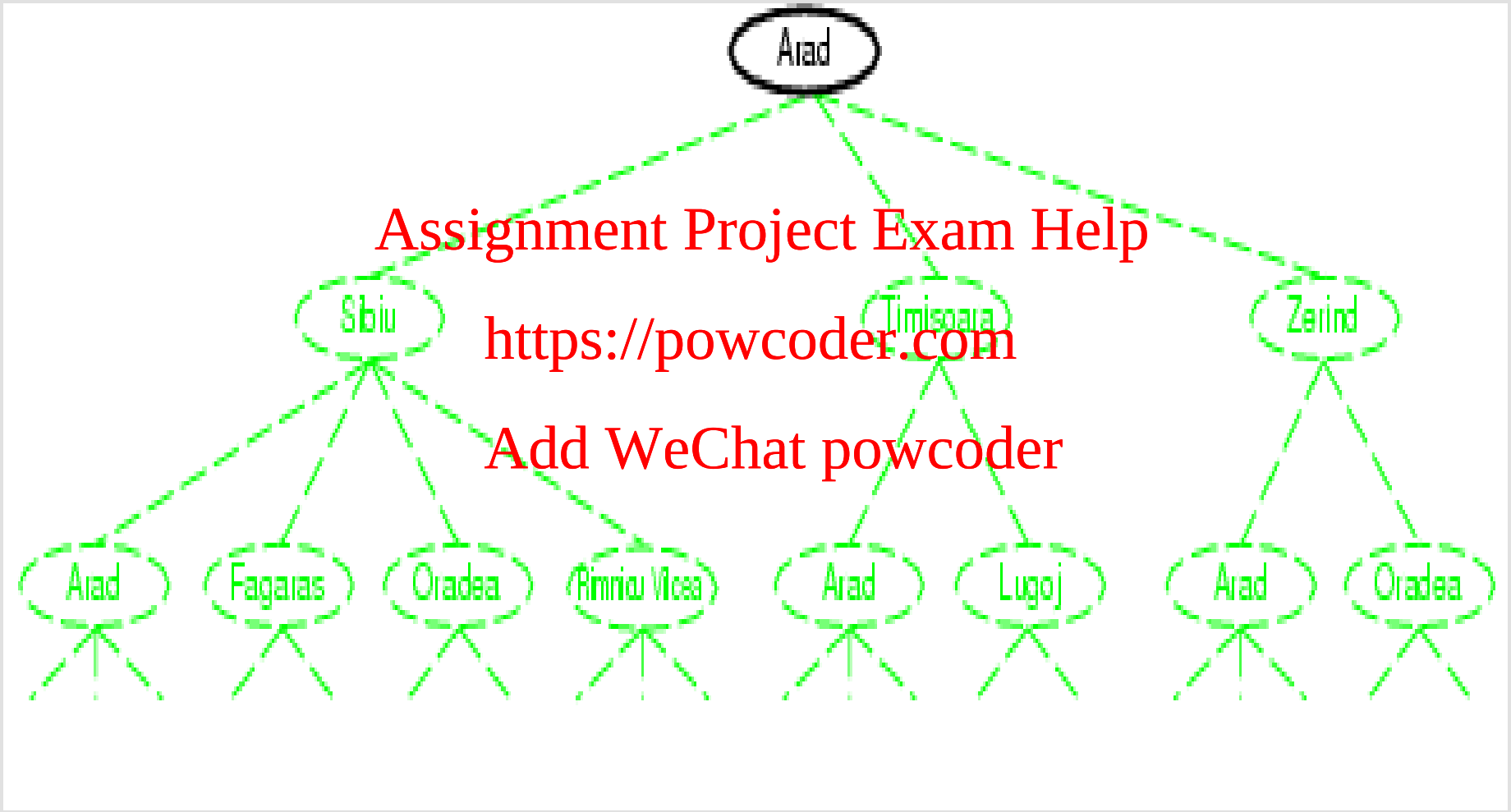
- Initialize the frontier using the initial state of *problem*
- **Loop**
  1. **if** the frontier is empty **then return** failure
  2. ***choose*** a leaf node and remove it from the frontier
  3. **if** the node contains a goal state **then return** the corresponding solution
  4. ***expand*** the chosen node, ***adding*** the resulting nodes to the frontier

  end

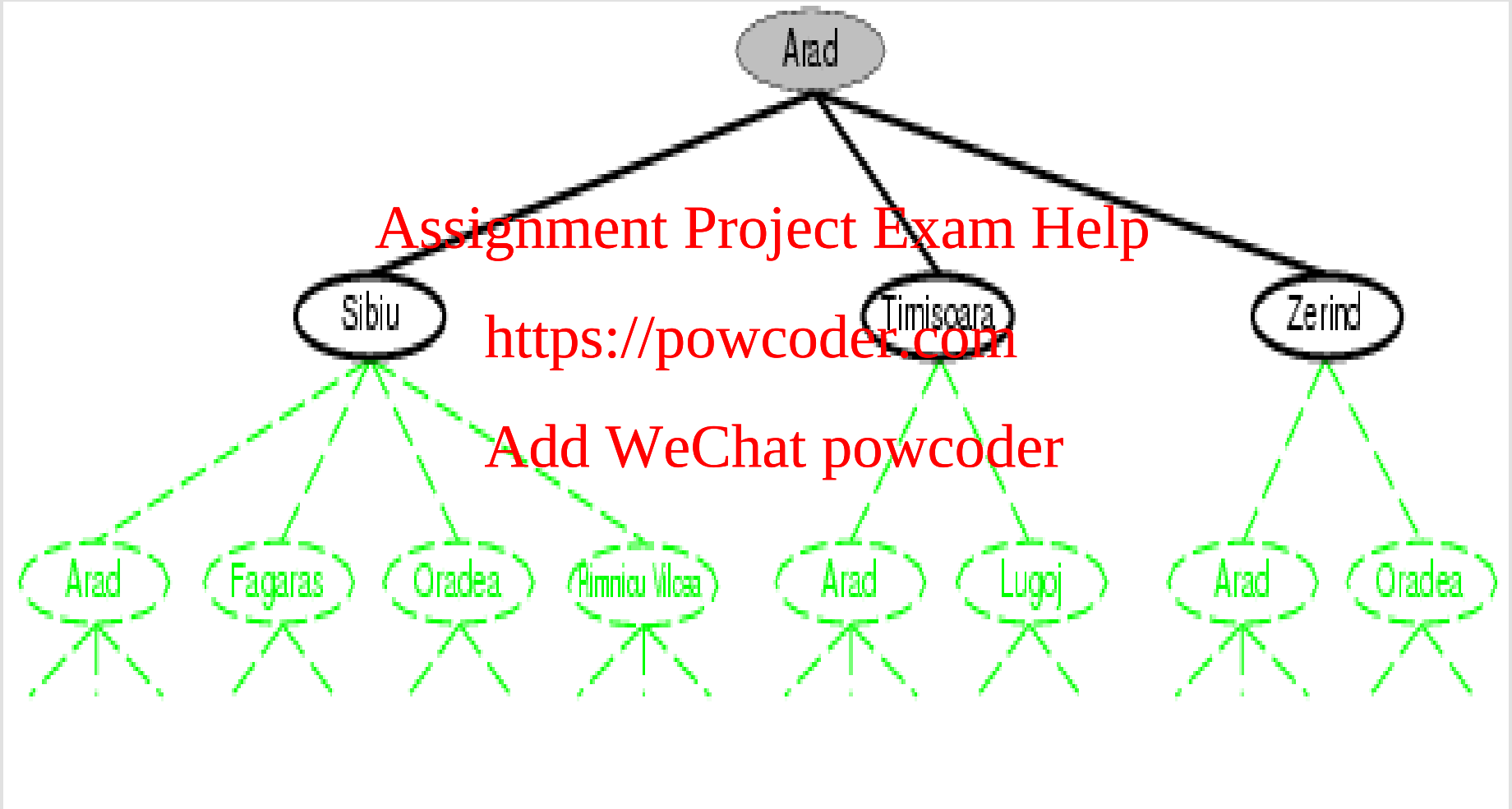
Example: Romania



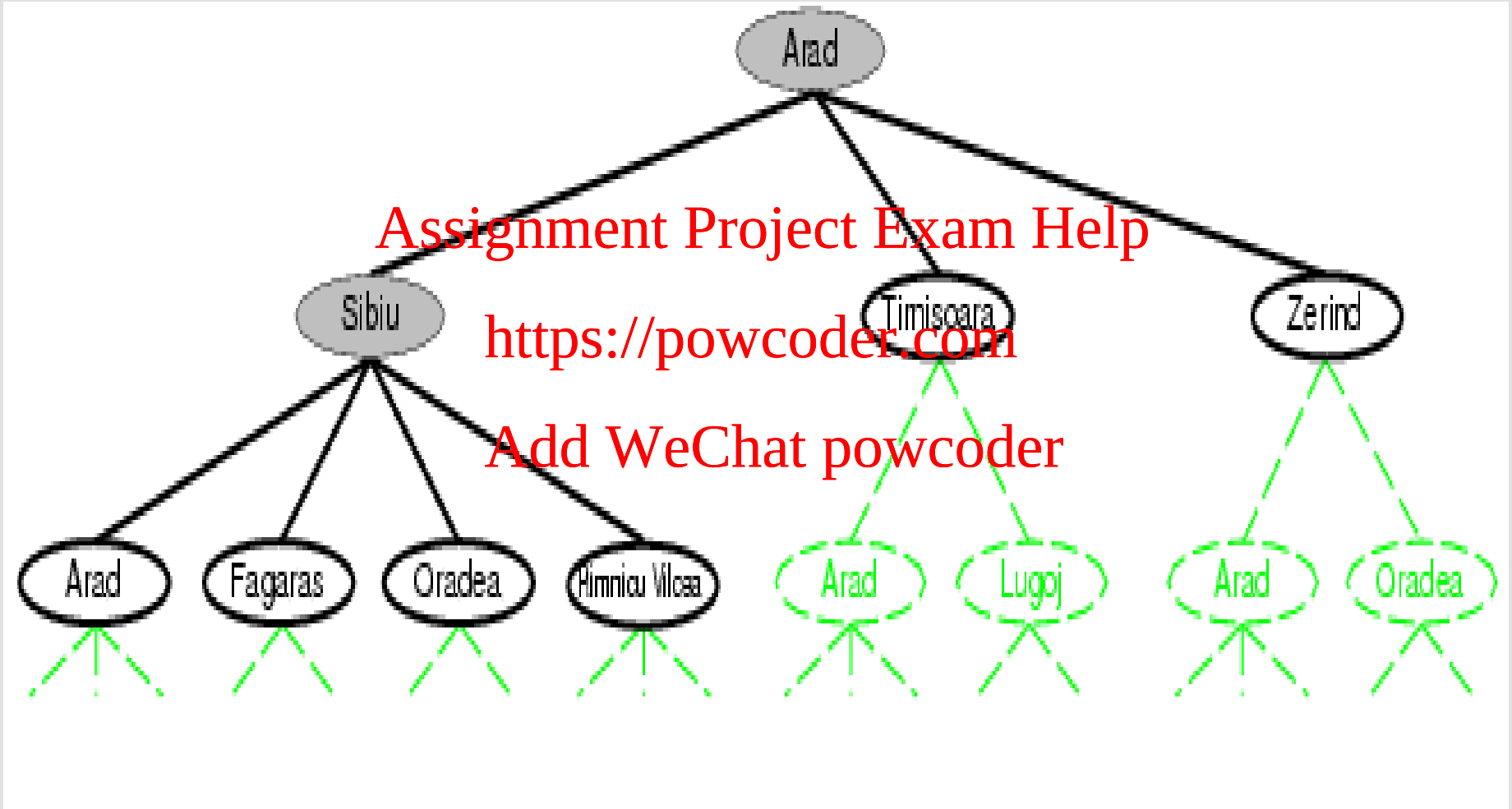
Example: Tree Search (Romania)



Example: Tree Search (Romania)

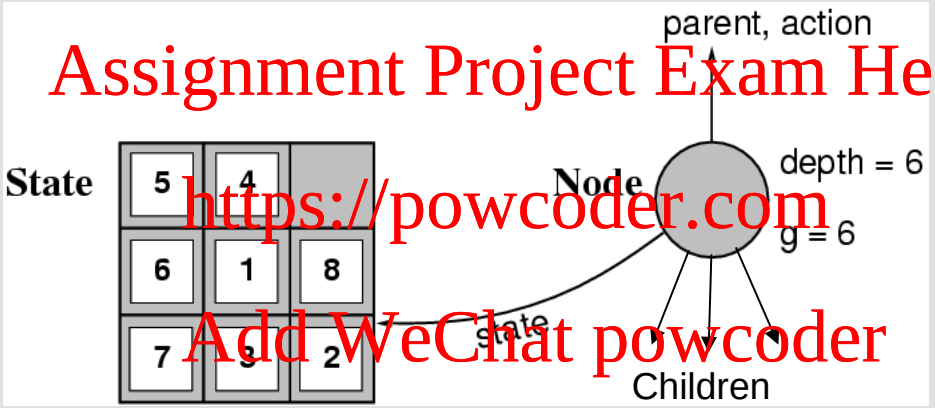


Example: Tree Search (Romania)



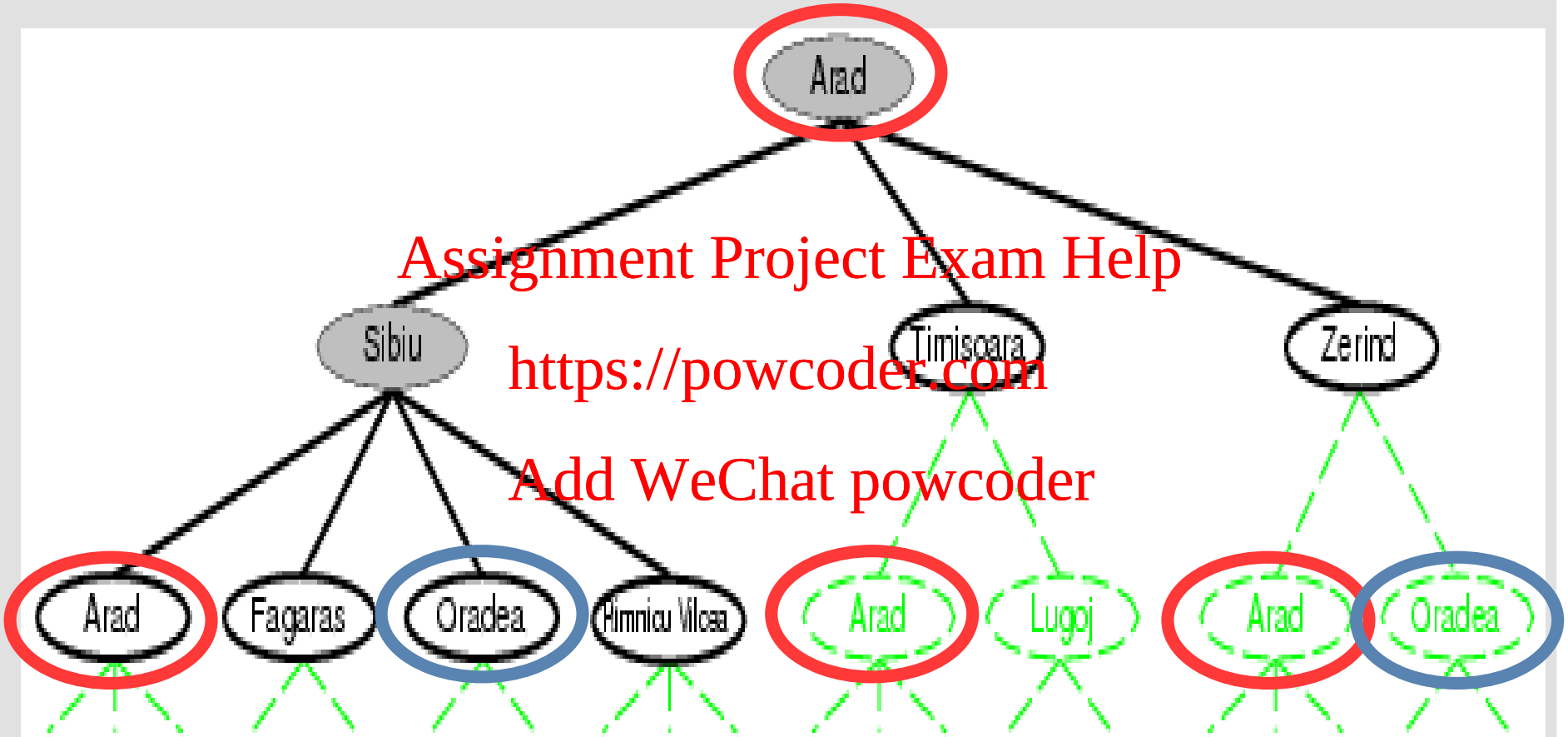
Implementation: States vs Nodes

- **state –** a (representation of a) physical configuration
- **node –** a data structure that is part of a search tree
  - includes *state, parent node, action, children, path cost g(x), depth*



- **The *Expand* function**
  - creates new nodes, fills in the various fields
  - uses ***SuccessorFn(Operators)*** to create the corresponding states
- **State space:** set of all reachable states
- **Search space:** set of all reachable nodes

Repeated States



Dealing with Repeated States

- **3 ways to deal with repeated states (ordered by cost and effectiveness):**
  - Do not return to the state you just came from
    → don't generate successors with same state as a node's parent
  - Do not create paths with cycles in them
    → don't generate successors with same state as any ancestor
  - Do not generate any state that was ever generated before
    → Use hashset – or some way of remembering - to check whether state has been visited

Graph Search Algorithm

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

Graph Search Algorithm

OPEN list

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

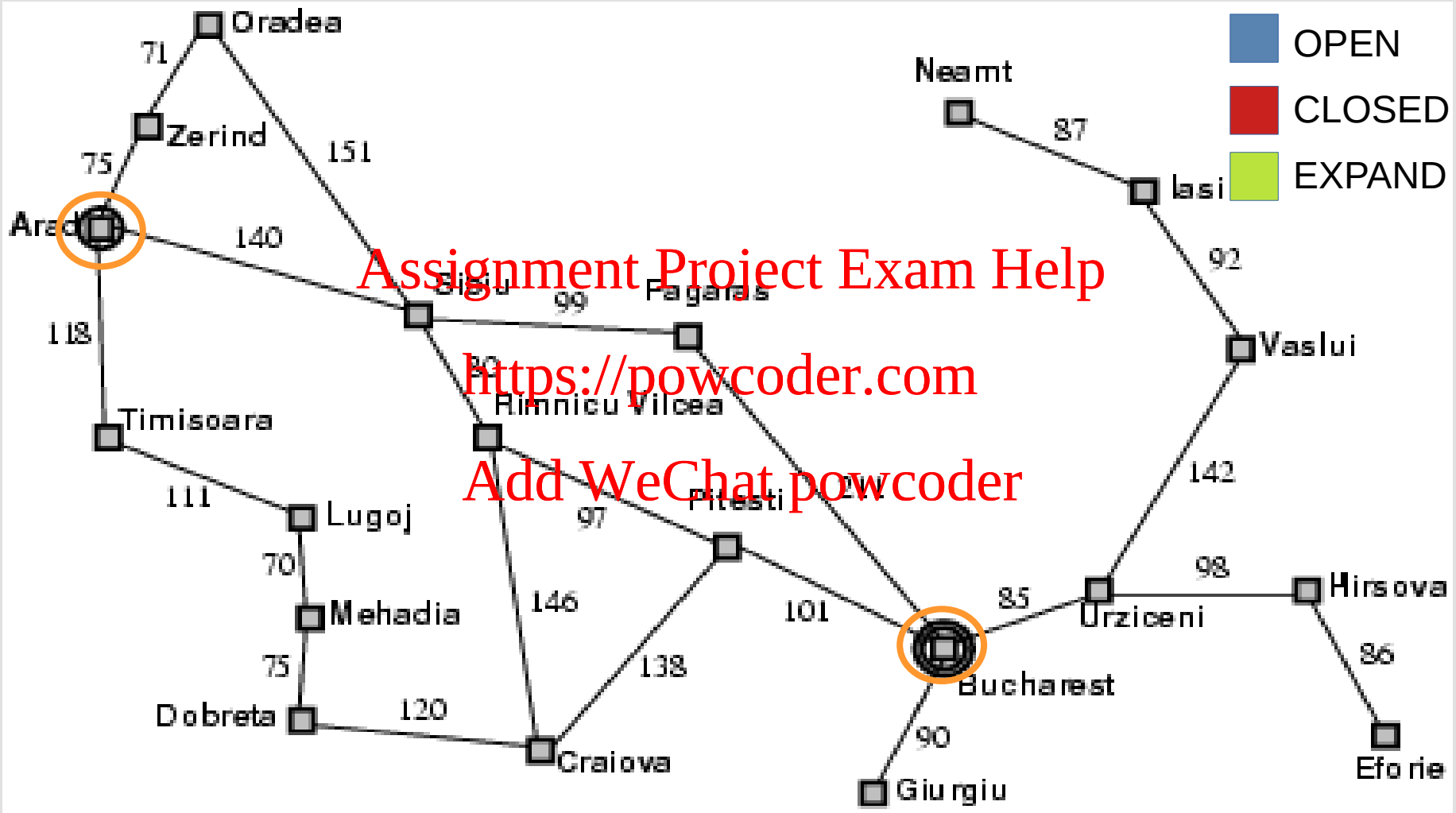
add the node to the explored set

expand the chosen node, adding the resulting nodes to the frontier

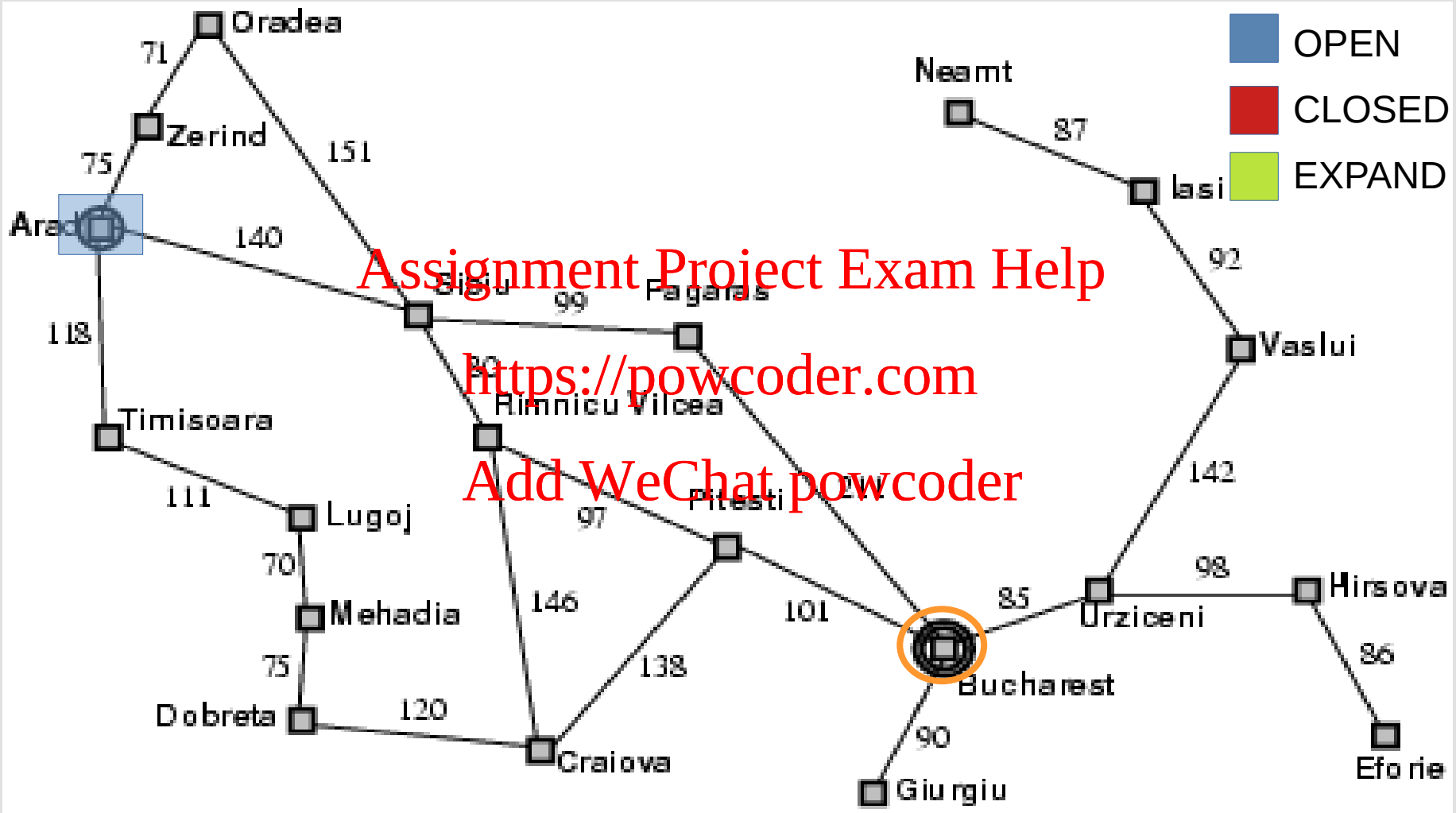
only if not in the frontier or explored set

CLOSED list

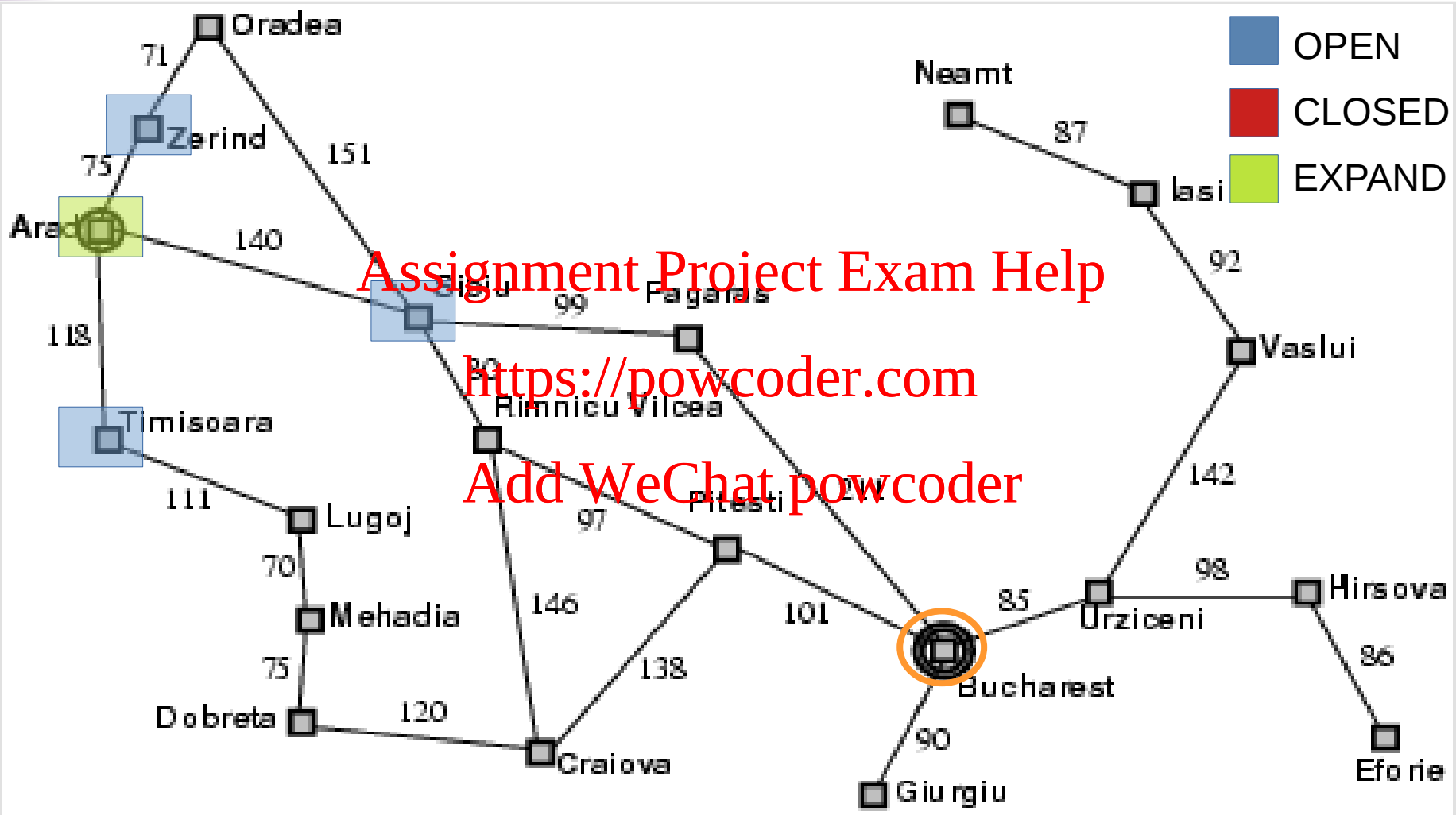
Graph Search Algorithm



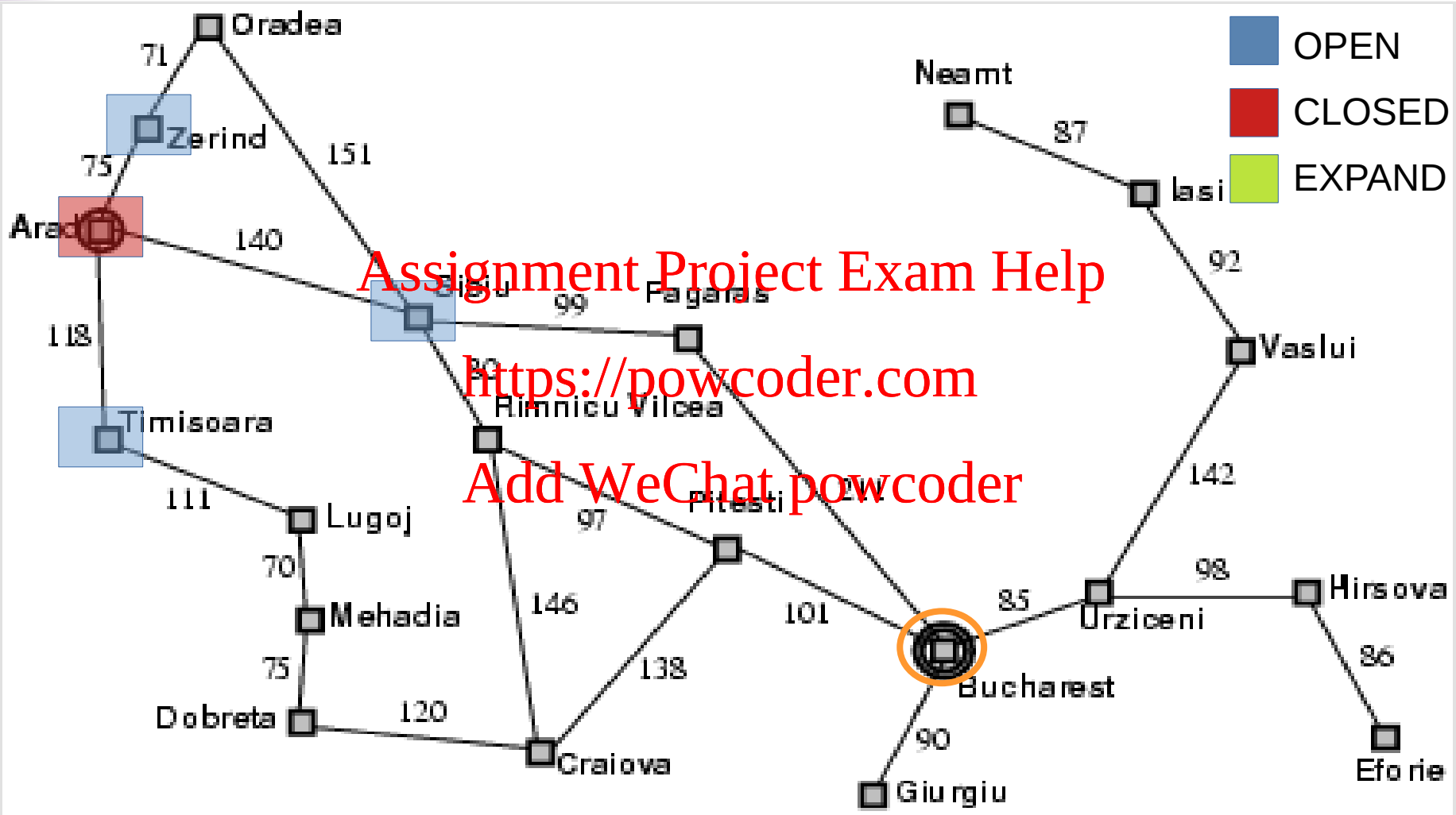
Graph Search Algorithm



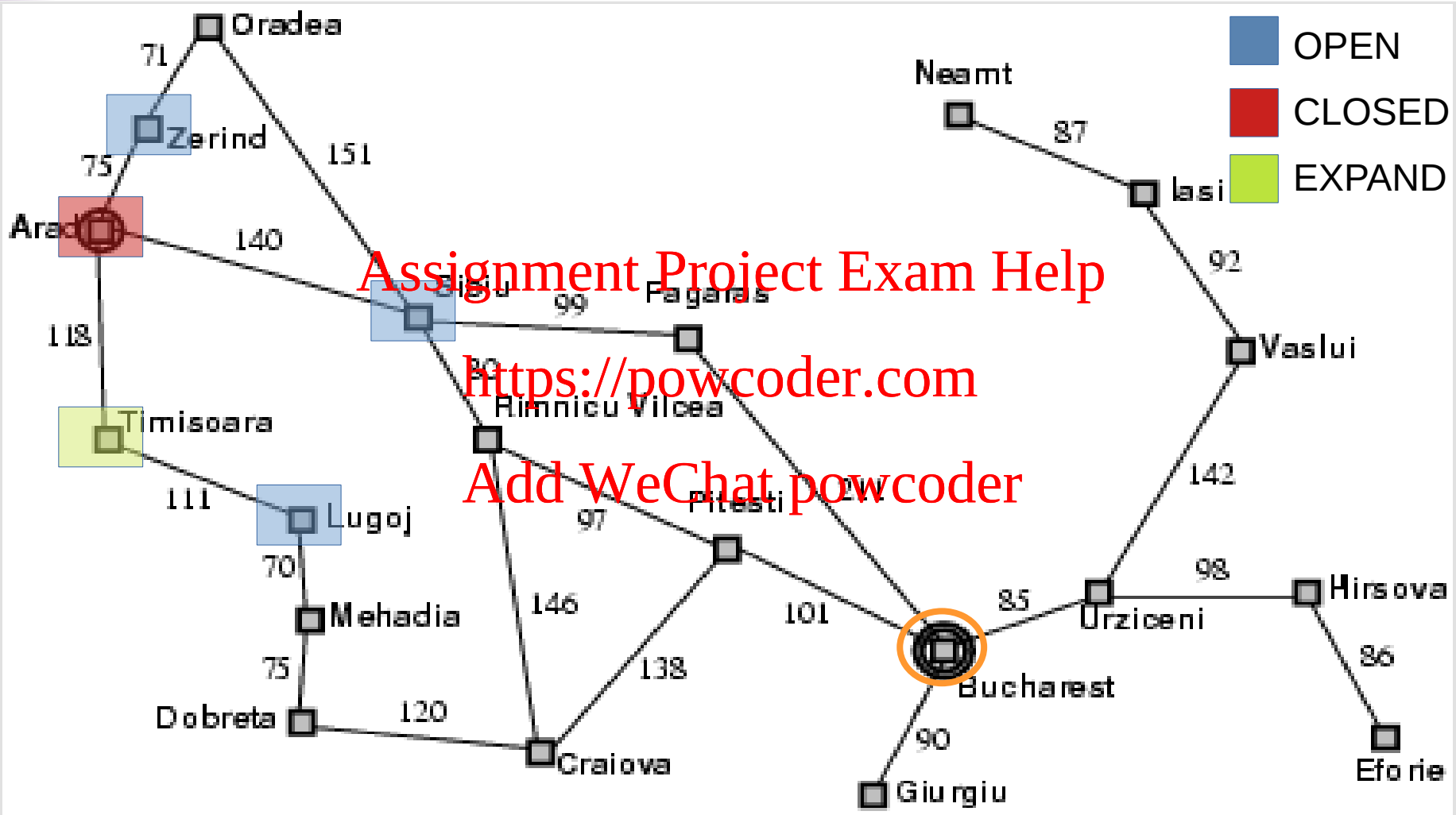
Graph Search Algorithm



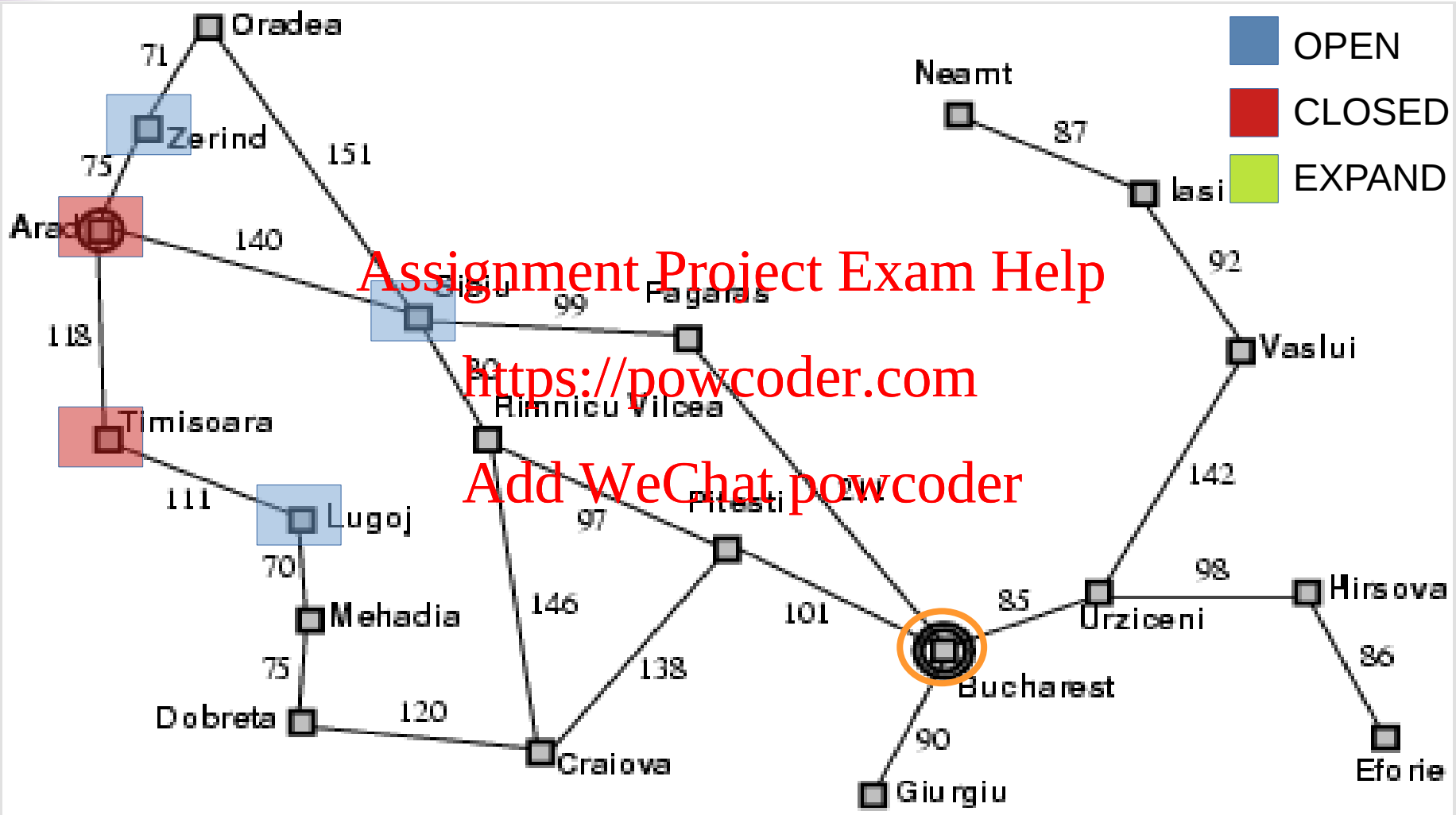
Graph Search Algorithm



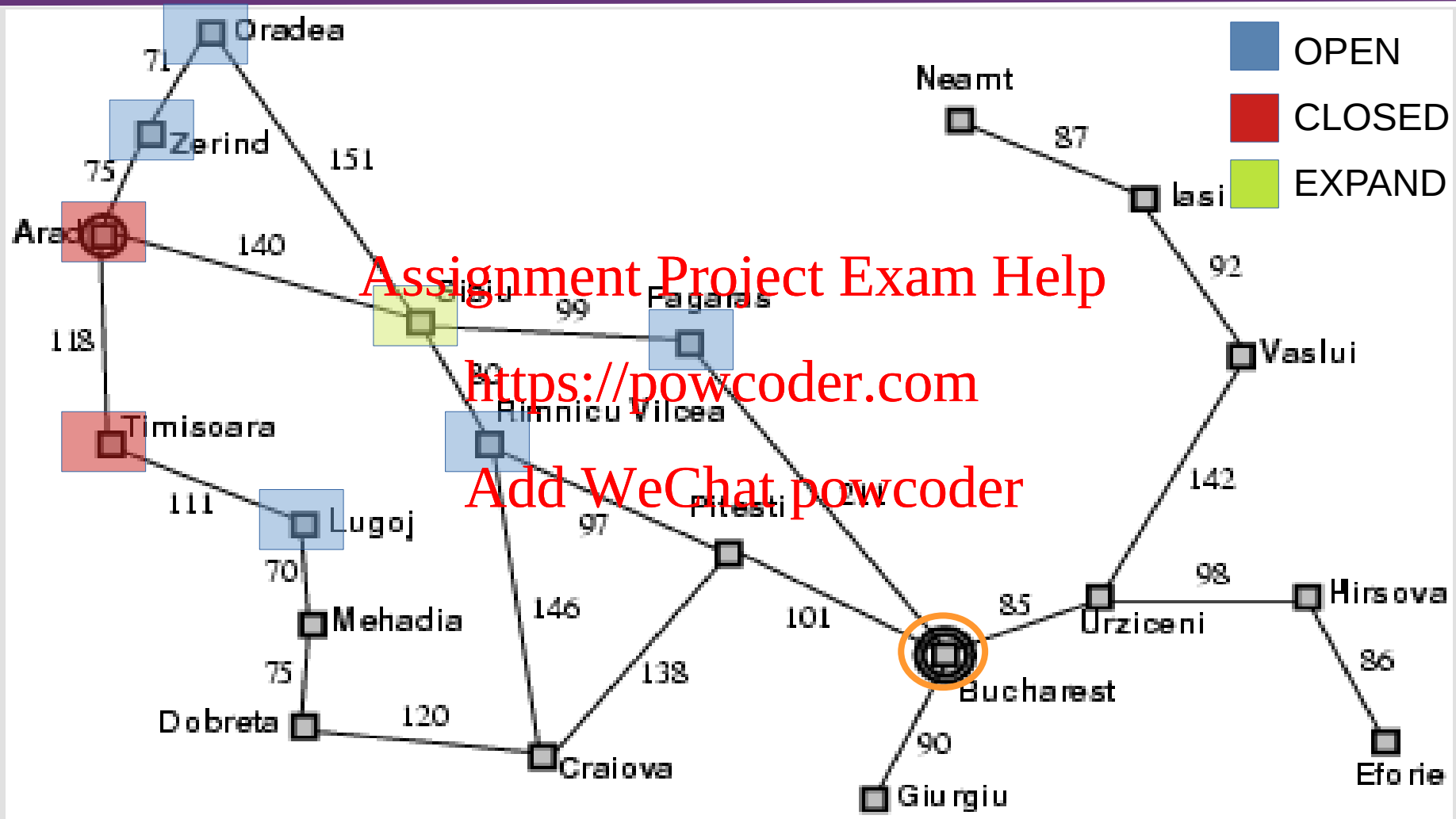
Graph Search Algorithm



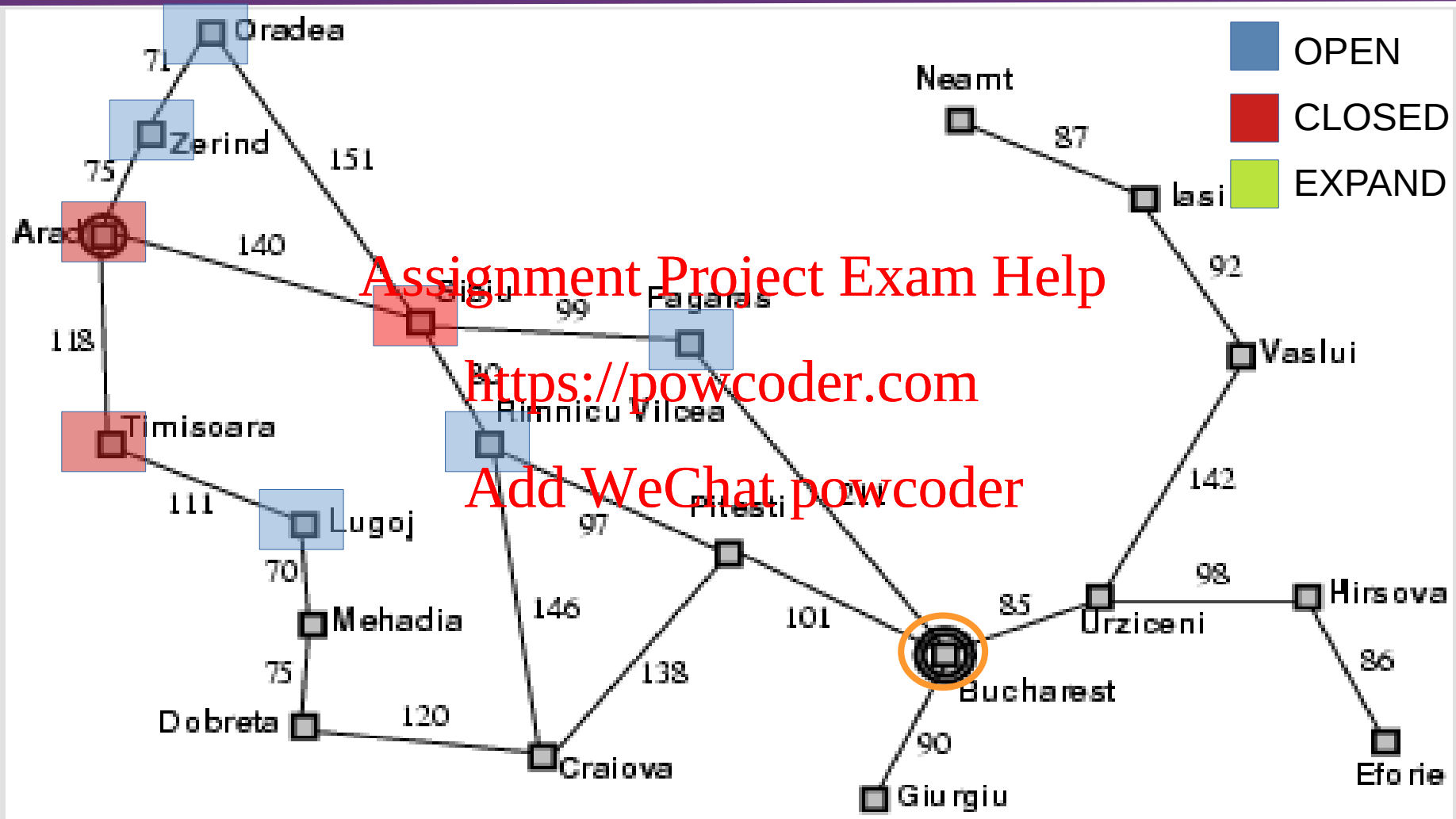
Graph Search Algorithm



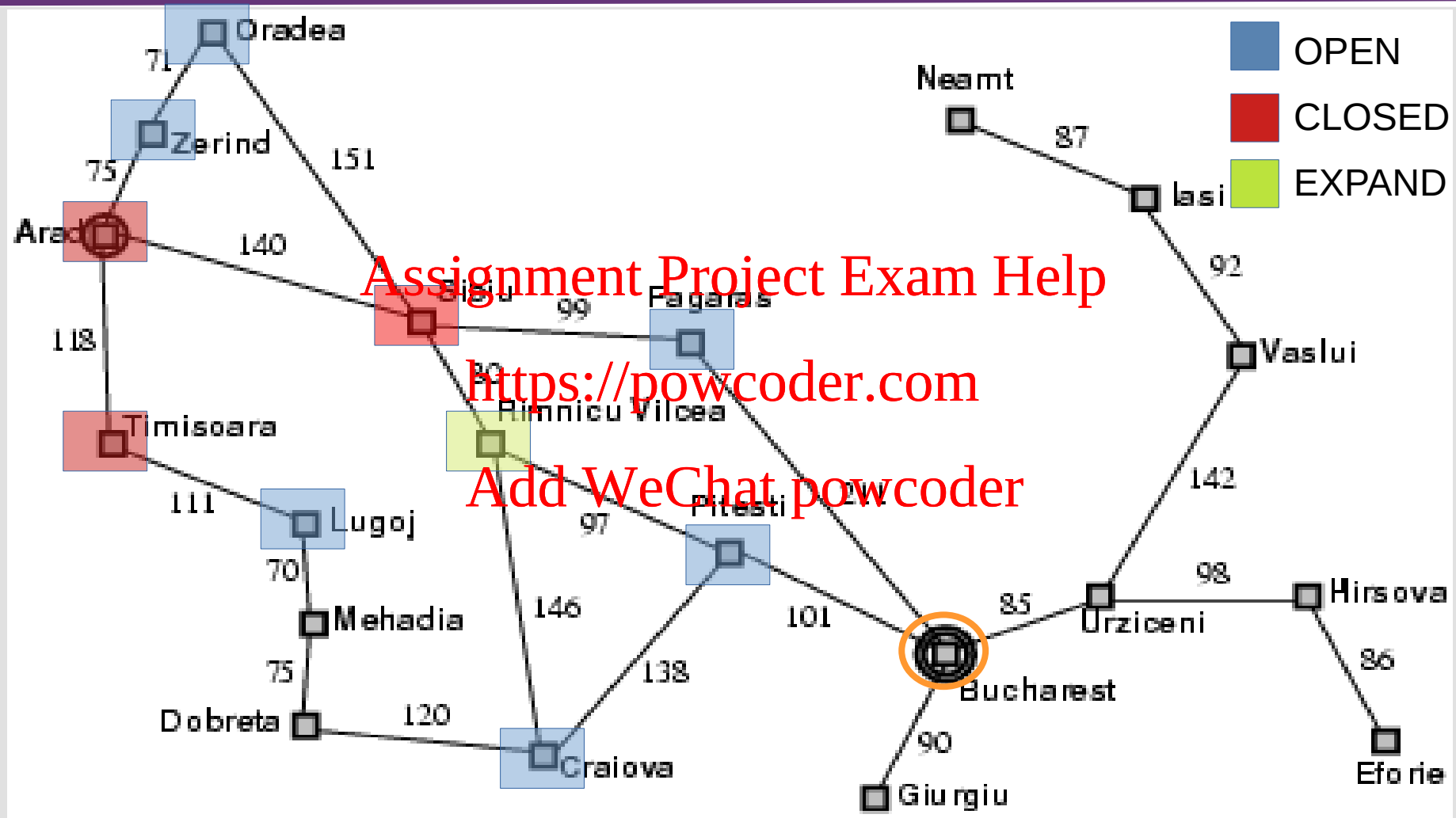
Graph Search Algorithm



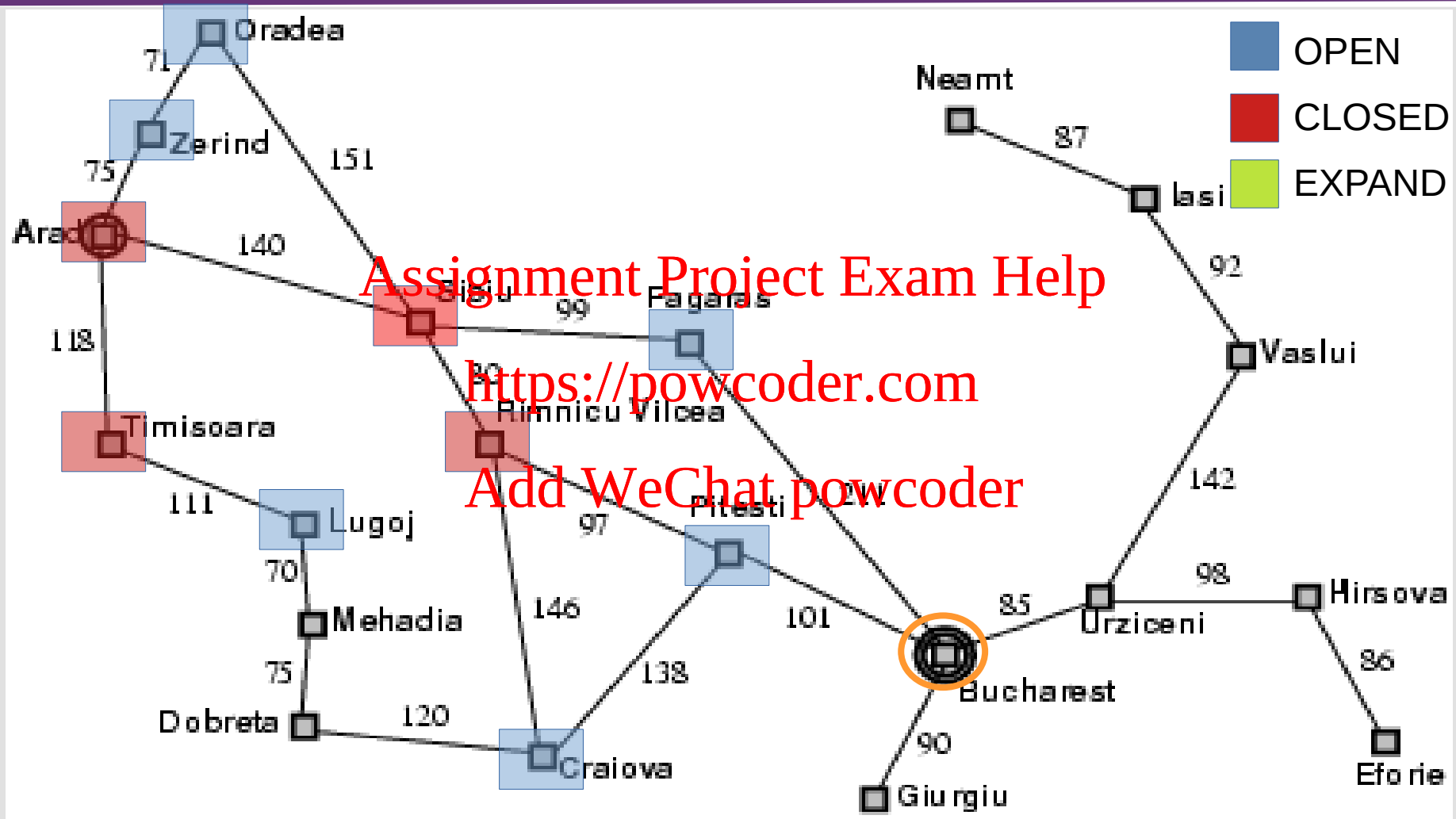
Graph Search Algorithm



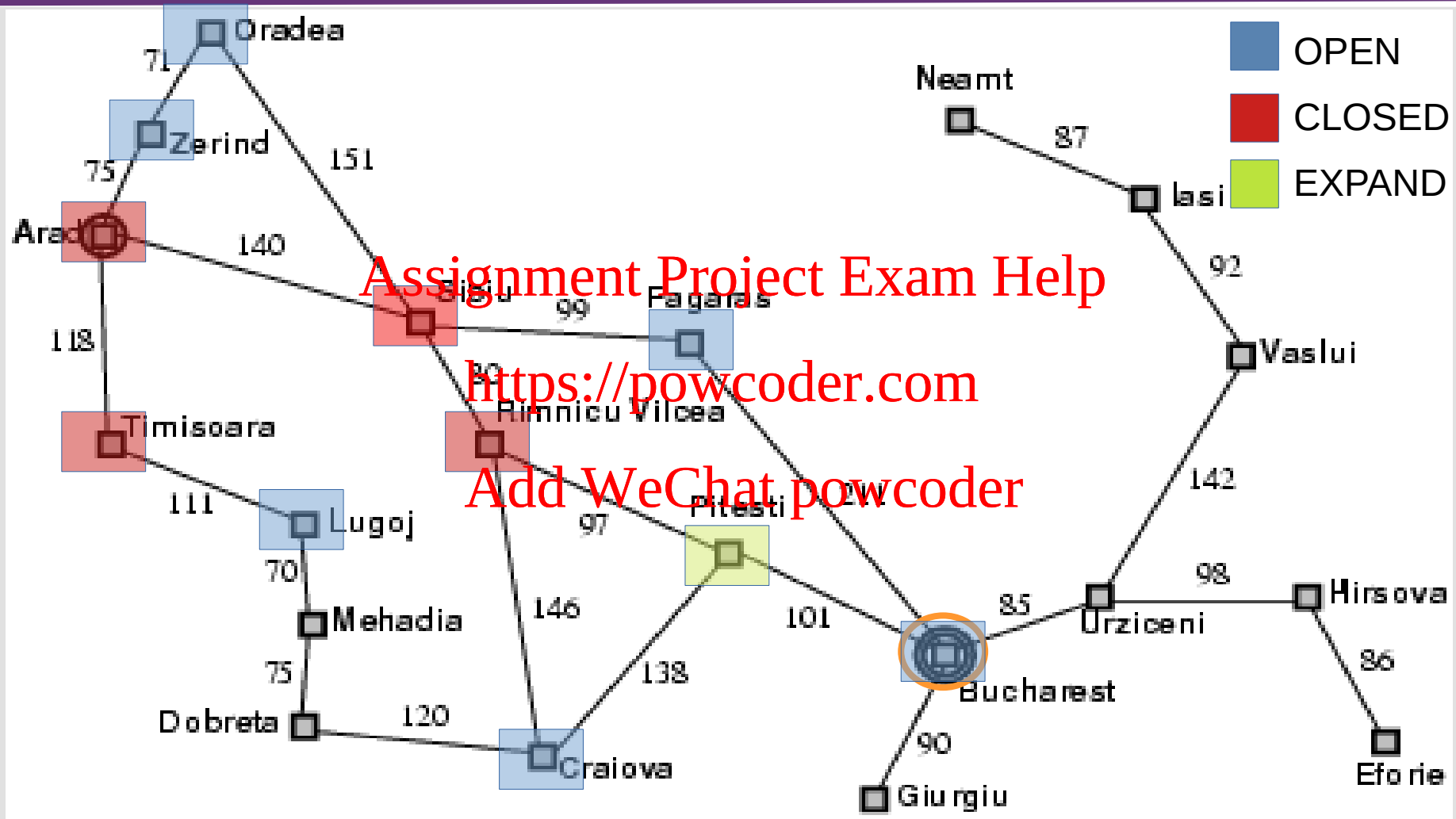
Graph Search Algorithm



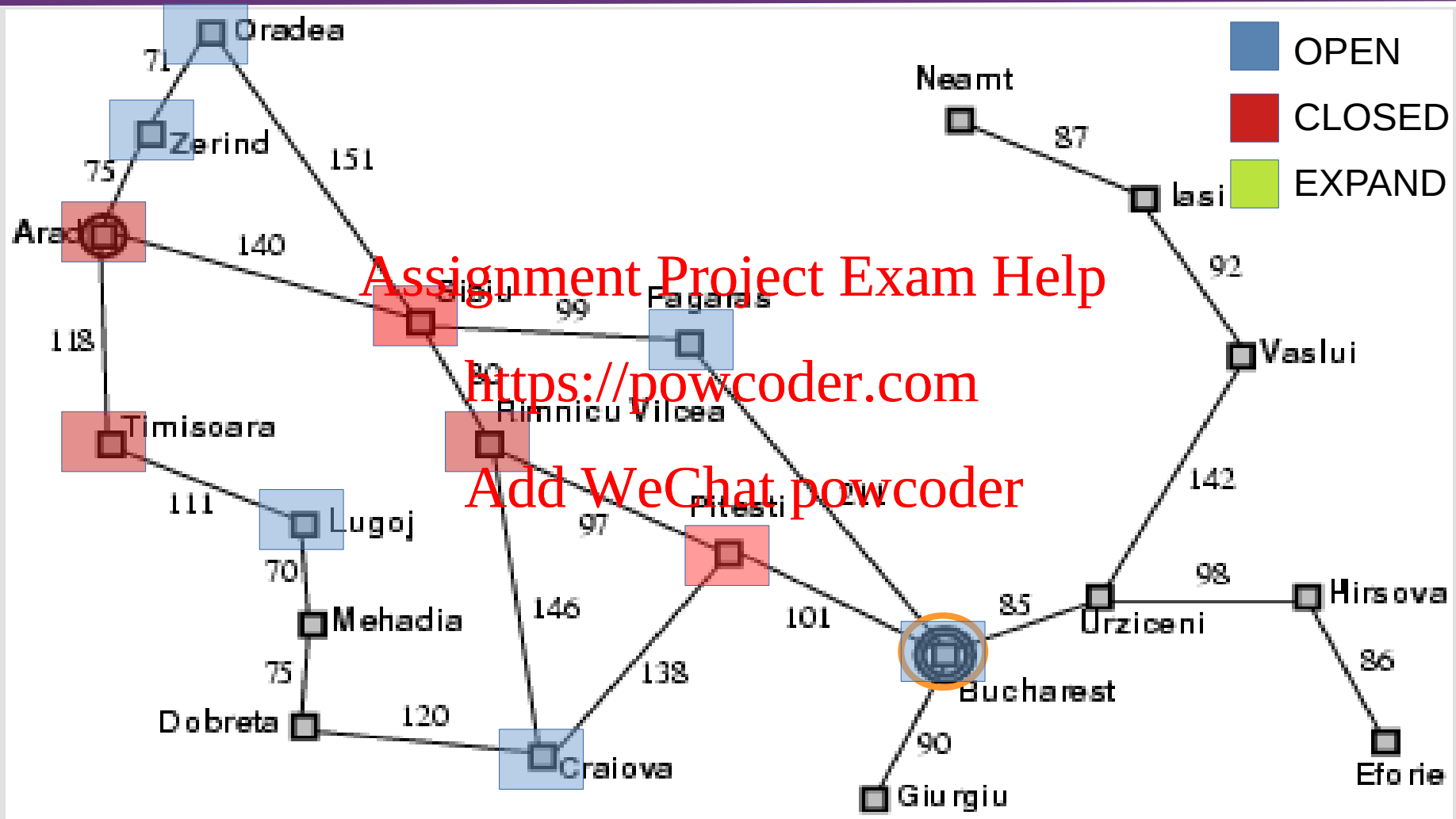
Graph Search Algorithm



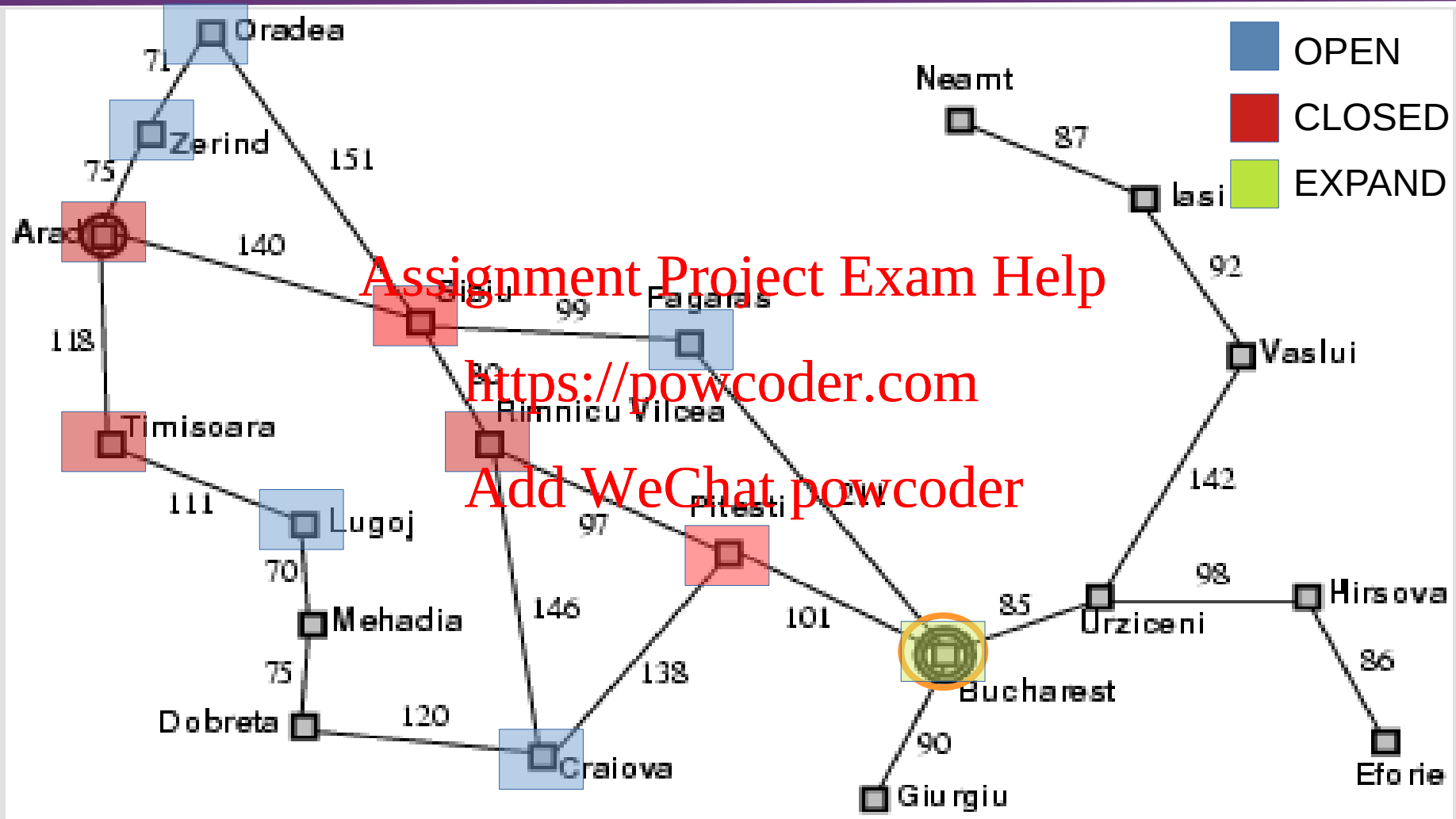
Graph Search Algorithm



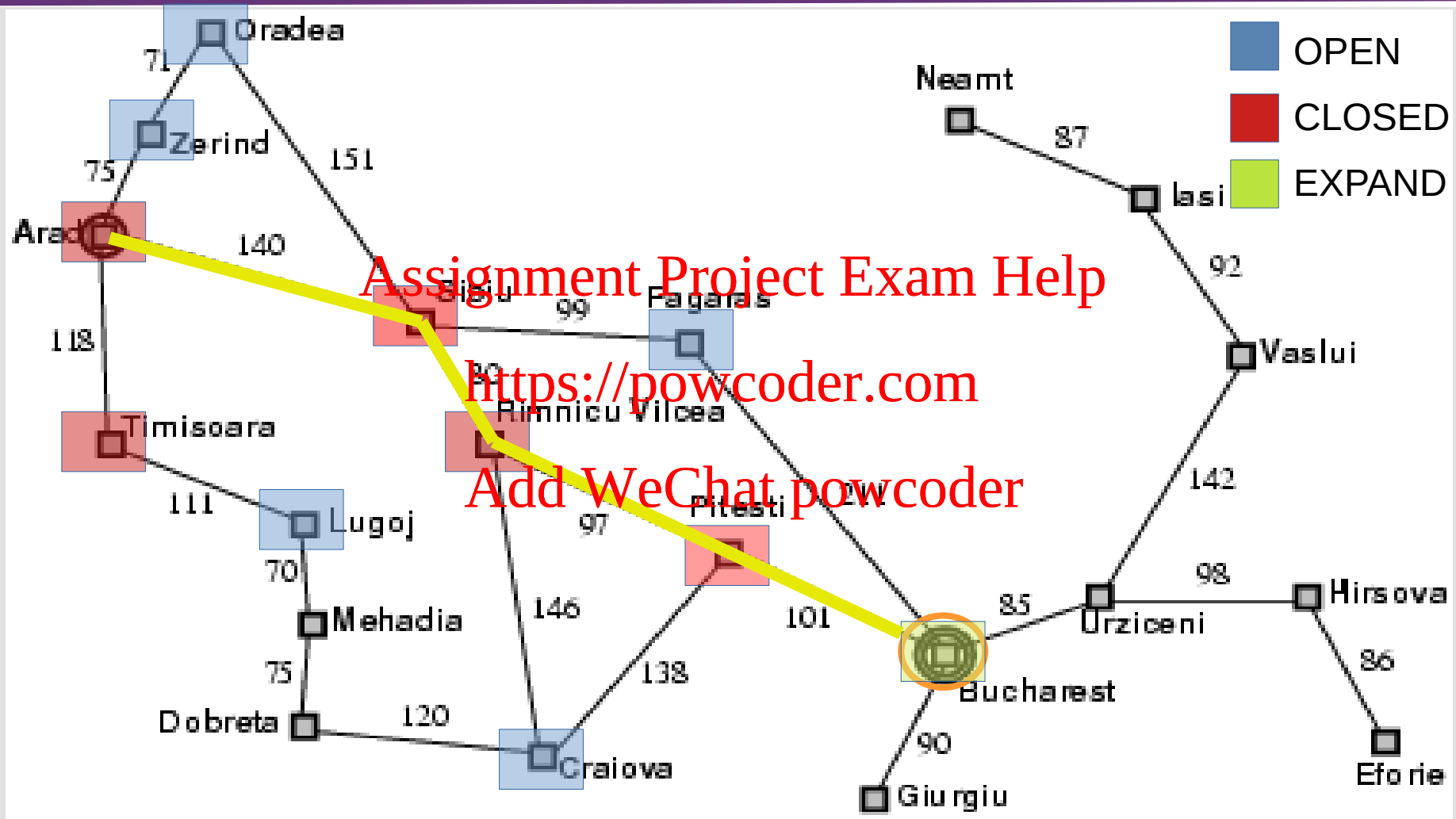
Graph Search Algorithm



Graph Search Algorithm



Graph Search Algorithm



Basic Search Algorithm: Key Issues

- Tree- or Graph- search?
  - Trees may be unbounded (search space is infinite) + suffer from repeated states
  - Graph may be prohibitively large (search space is huge)
- Return a path or a node?
- Repeated states
  - Failure to detect repeated states can increase the complexity of a problem
- How are the nodes ordered? ➔Search strategy
  - Is the graph weighted or unweighted?
  - How much is known about the “quality” of intermediate states?
  - Is the aim to find a ***minimal cost path*** or ***any path***
  - ***a.s.a.p. (as soon as possible)***?

MONASH University
Information Technology

FIT3080 – Artificial Intelligence

Search Strategies

Search Strategies

A search strategy is defined by picking the order of node expansion

- Can be categorised into two distinct types
  - Uninformed – decide based ***only*** on problem definition
  - Informed – use guidance on where to look for solutions
- Evaluated along several dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: maximum number of nodes generated
  - **space complexity**: maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?
- Some important metrics for comparing strategies:
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of any path in the state space (may be ∞)

MONASH University
Information Technology

FIT3080 – Artificial Intelligence

Uninformed Search Strategies

Uninformed Search Strategies

Uninformed search strategies use only the information available in the problem definition

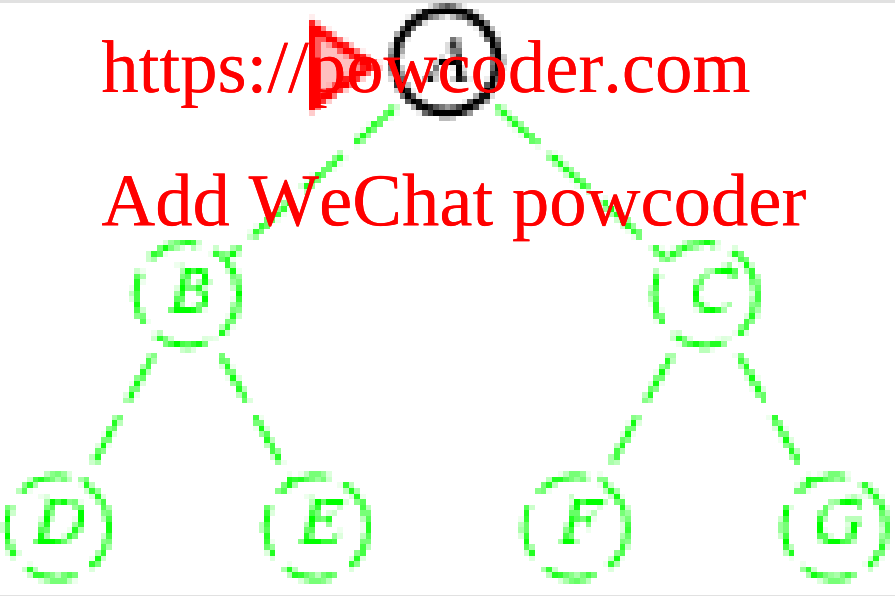
- action costs
- node depth
- etc

Some well known approaches:

- Breadth-first search (BFS)
- Uniform-cost search (UCS)
- Depth-first search (DFS)
- Depth-limited search (DLS)
- Iterative deepening search (IDS)

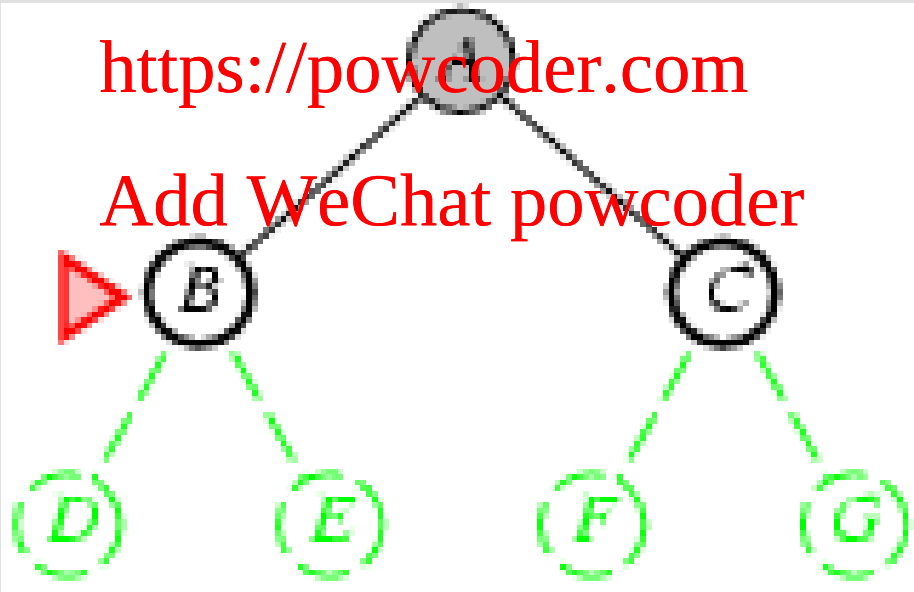
Breadth-first Search (I)

- Expand shallowest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: FIFO – put successors at end of queue



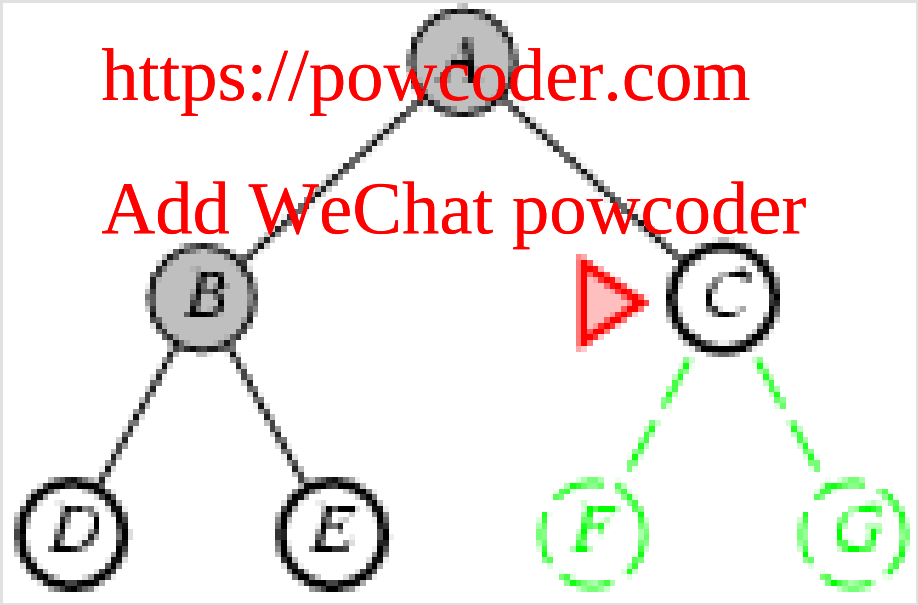
Breadth-first Search (II)

- Expand shallowest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: FIFO – put successors at end of queue



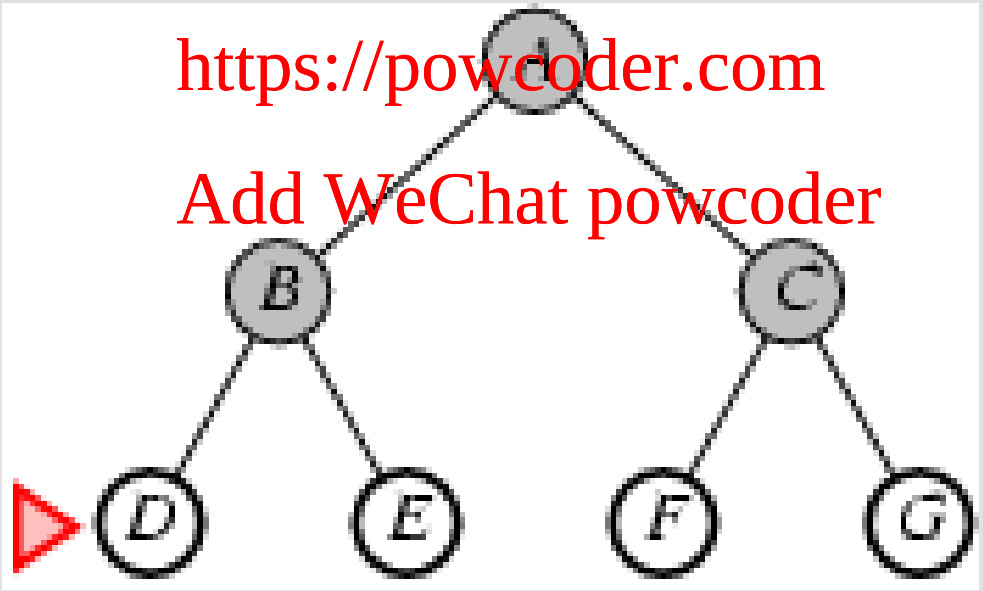
Breadth-first Search (III)

- Expand shallowest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: FIFO – put successors at end of queue

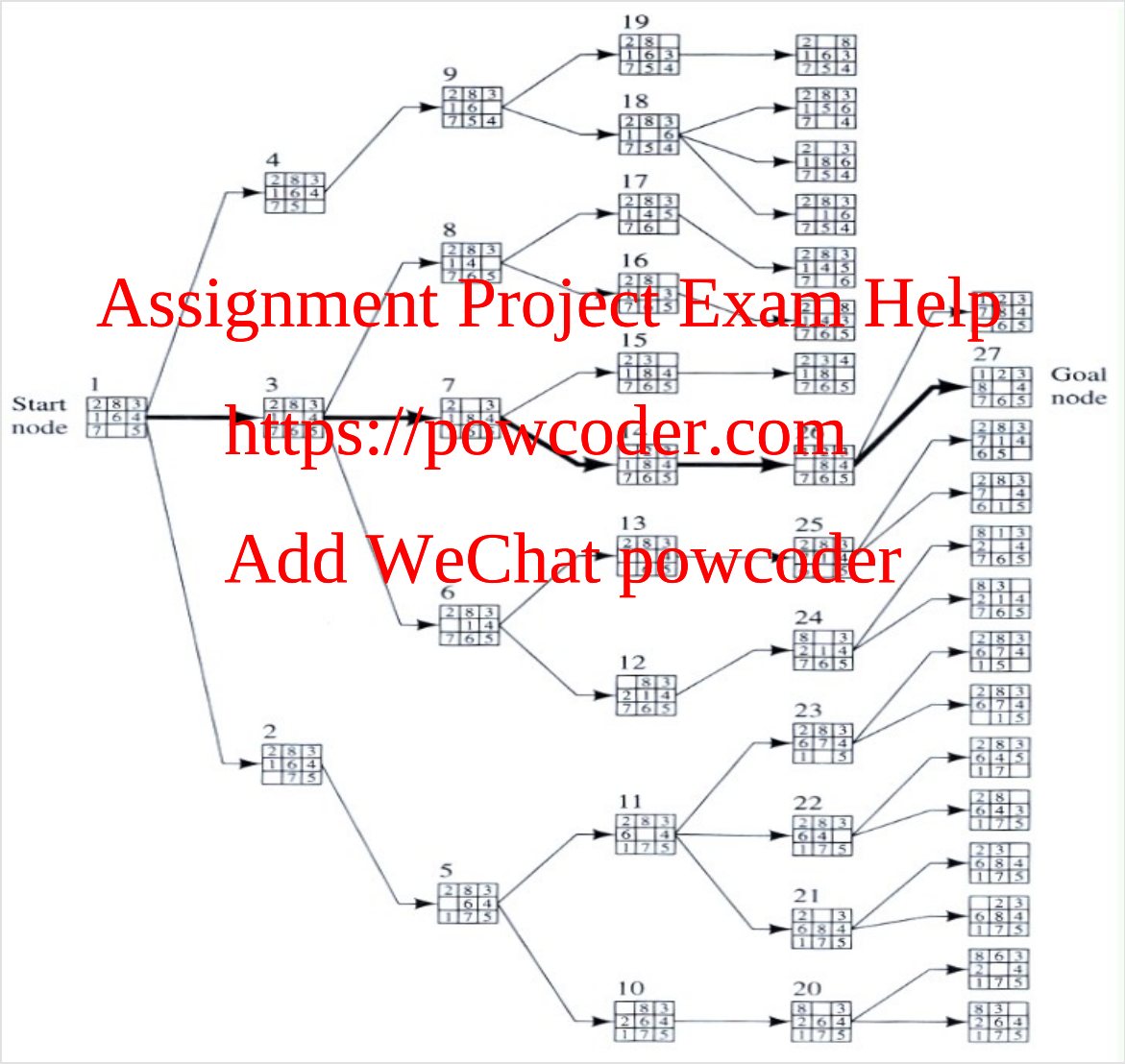


Breadth-first Search (IV)

- Expand shallowest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: FIFO – put successors at end of queue



BFS – Example



Properties of Breadth-First Search

- Complete? Yes (if b is finite)
- Time? $b + b^2 + b^3 + \ldots + b^d = b\frac{b^d - 1}{b-1} = O(b^d)$

also $= 1 + b + \ldots + b^d - 1 = \frac{b^{d+1} - 1}{b-1} - 1 = O(b^d)$

- Space? $O(b^d)$ (keeps every node in memory)

- Optimal? Yes (if all actions have the same cost)

Complexity of BFS

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

Assume: Branching factor b=10, 1 million nodes/second, 1000 bytes per node

Complexity of BFS

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

Assume: Branching factor b=10, 1 million nodes/second, 1000 bytes per node

Memory is the bigger problem

Uniform-Cost Strategy

- Expand node n with the smallest path cost (g-value)
  - g(n) = sum of action costs from the start node
- Update g-values when we find a smallest cost path to a frontier node.
- **QUEUEING-FN:** Priority Queue (e.g., Binary Heap)

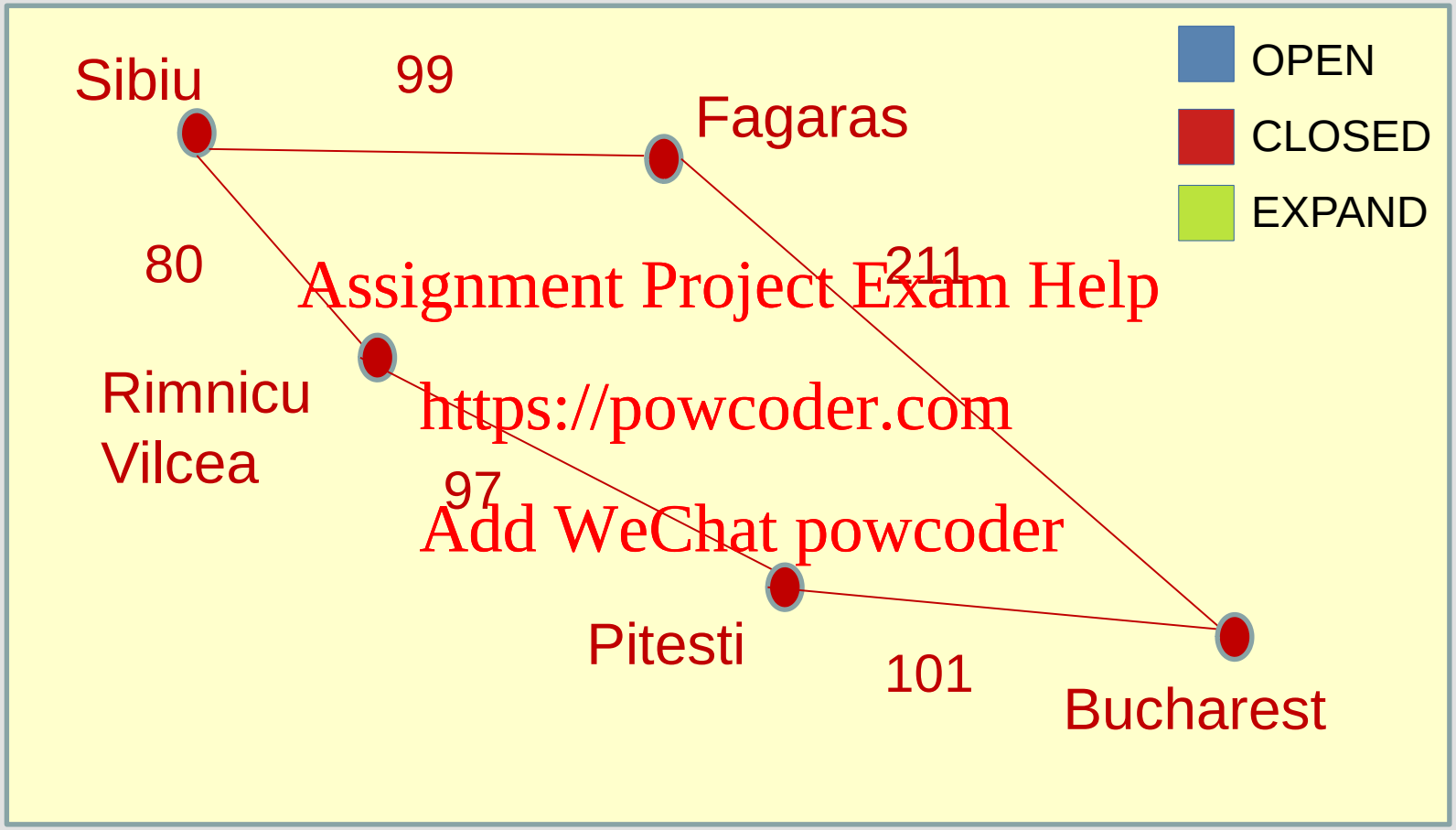
Uniform-Cost Search (Tree or Graph)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution or failure

- Initialize the frontier (OPEN list) using the initial state of *problem*
- **Loop**
  1. **if** OPEN is empty **then return** failure
  2. ***remove*** the lowest-cost node from OPEN
  3. **if** the node is a goal **then return** corresponding solution.
  4. ***add the chosen node to the CLOSED list.***
  5. ***expand*** the chosen node
     a. **if** the resulting nodes are in CLOSED **then discard** them
     b. **if** the resulting nodes are not in OPEN **then add** them
     c. **else if** the resulting nodes are in OPEN ***with higher path cost*** **then update** them and their re-compute their priority

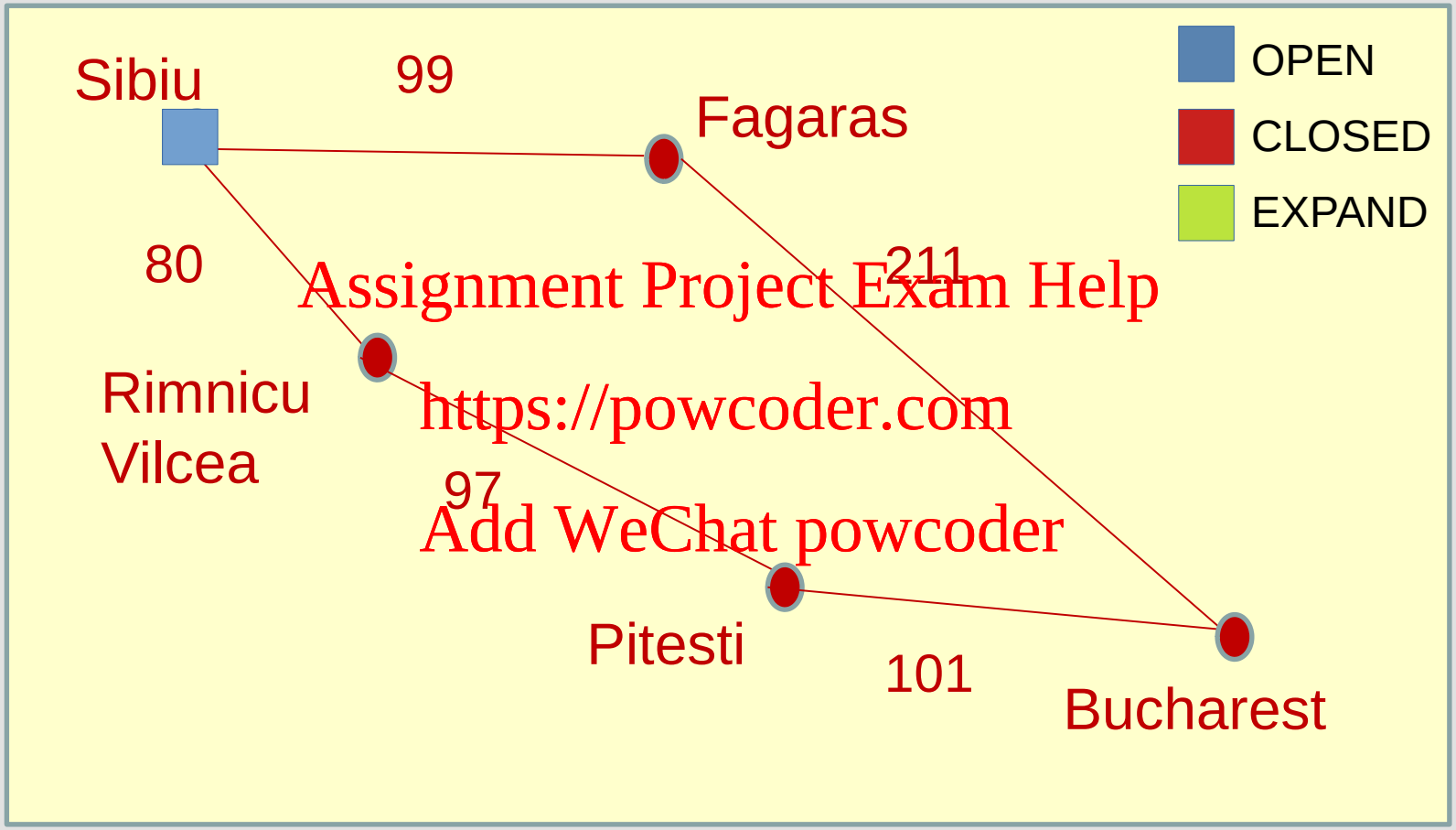
  end

Uniform-cost (Graph-) Search



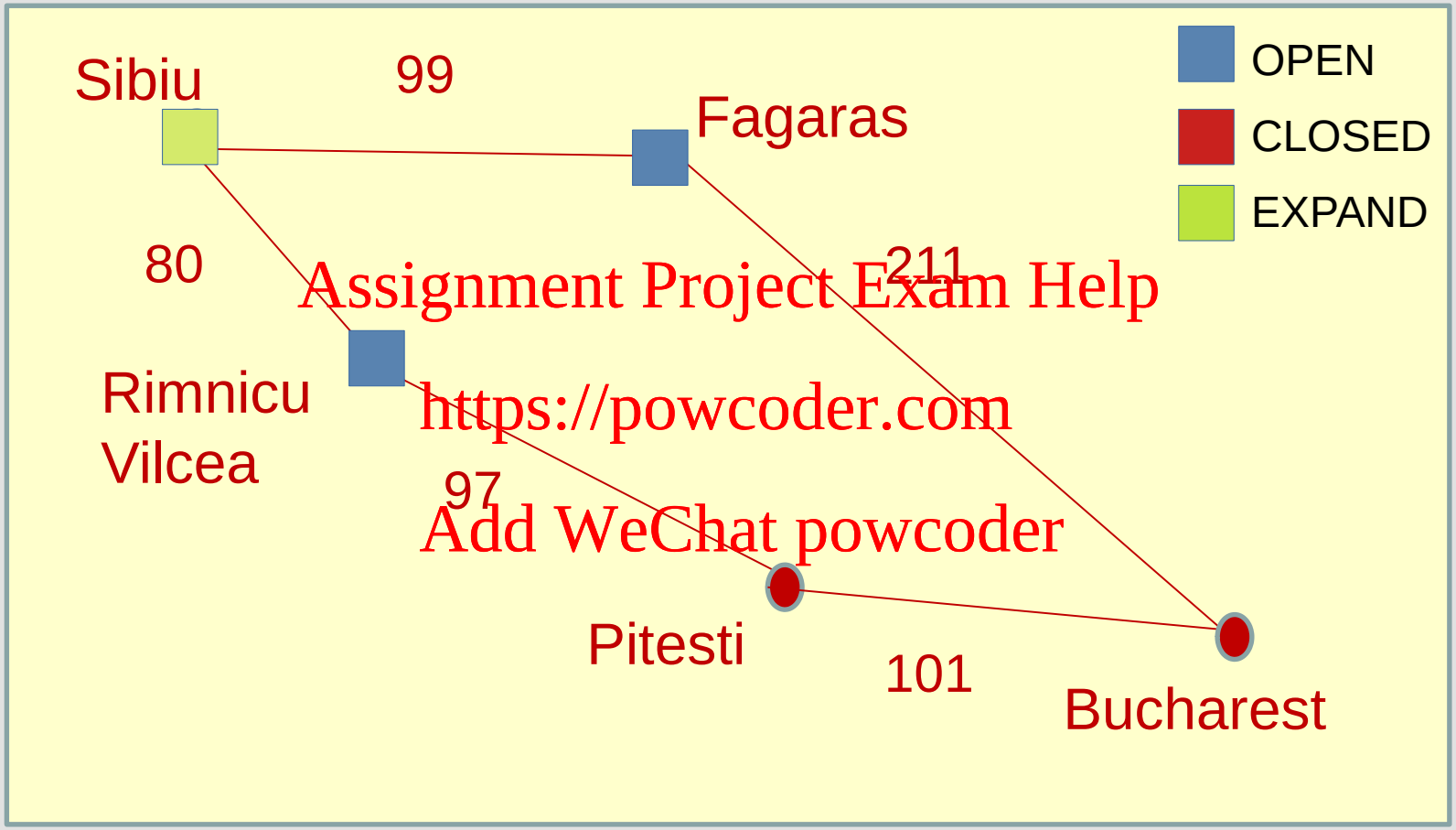
OPEN = { }
CLOSED = { }

Uniform-cost (Graph-) Search



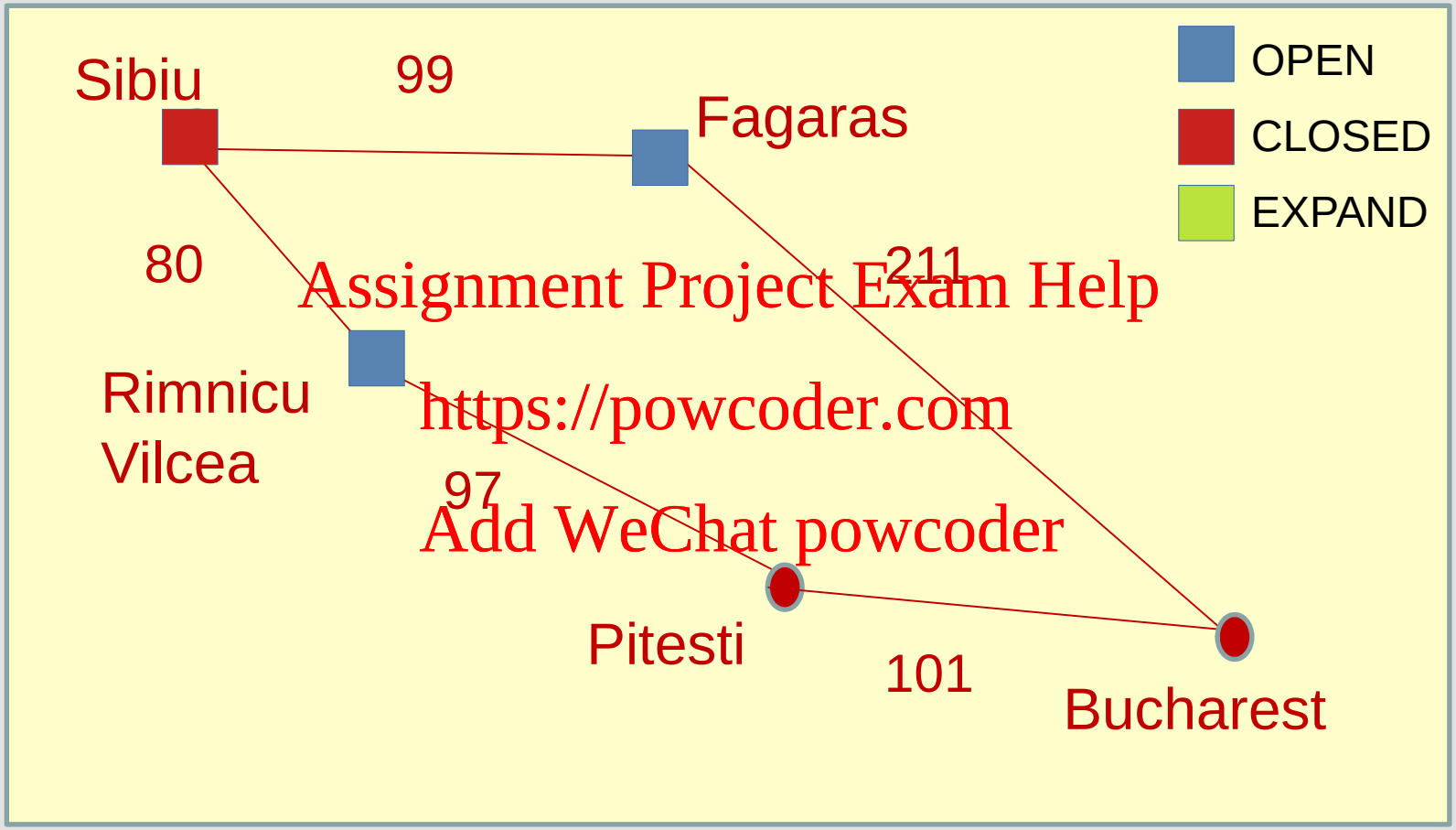
OPEN = { (Sibiu, 0) }
CLOSED = { }

Uniform-cost (Graph-) Search



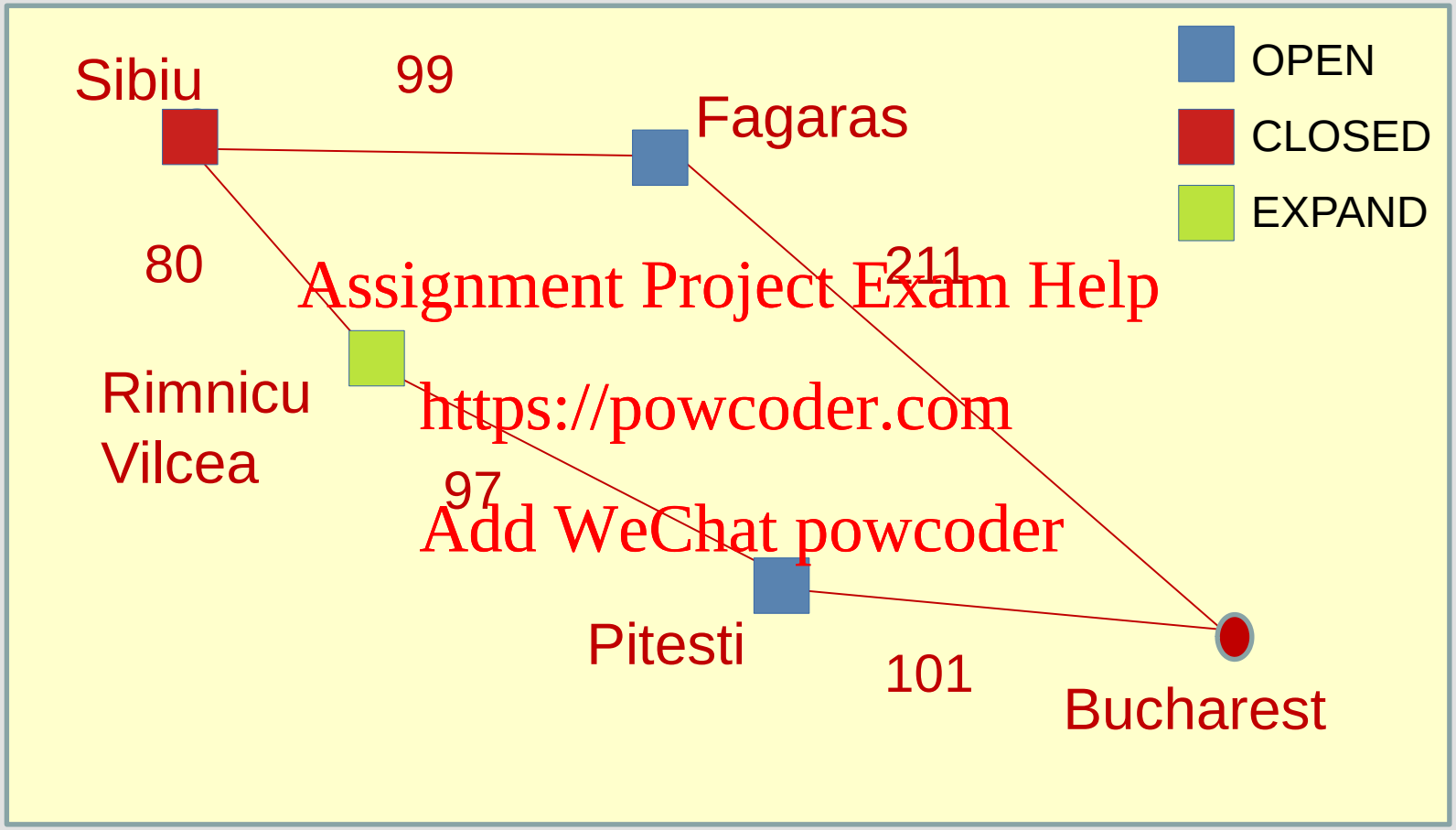
OPEN = { (Rimnicul Vilcea, 80), (Fagaras, 99) }
CLOSED = { }

Uniform-cost (Graph-) Search



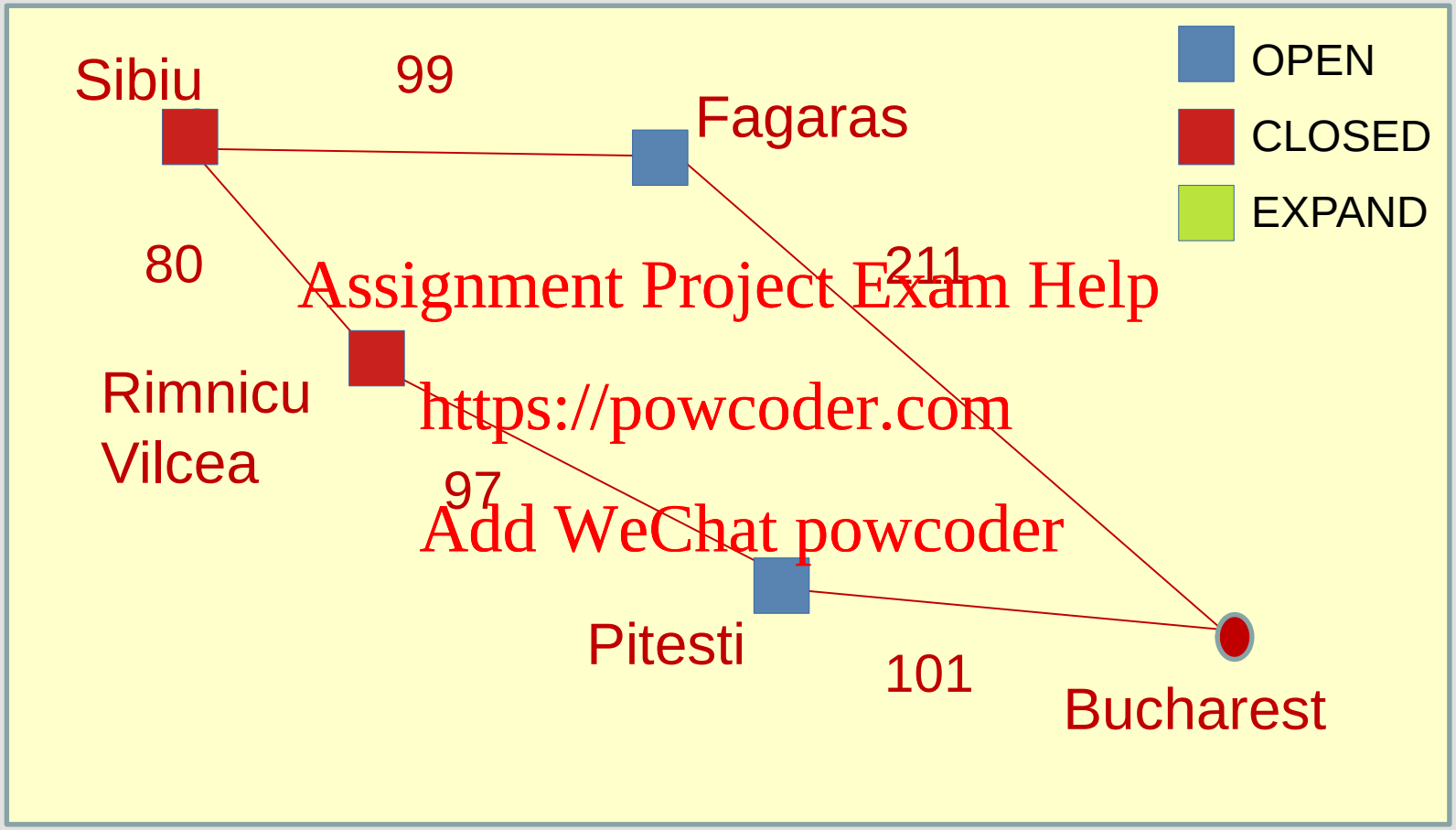
OPEN = { (Rimnicul Vilcea, 80), (Fagaras, 99) }
CLOSED = { (Sibiu, 0) }

Uniform-cost (Graph-) Search



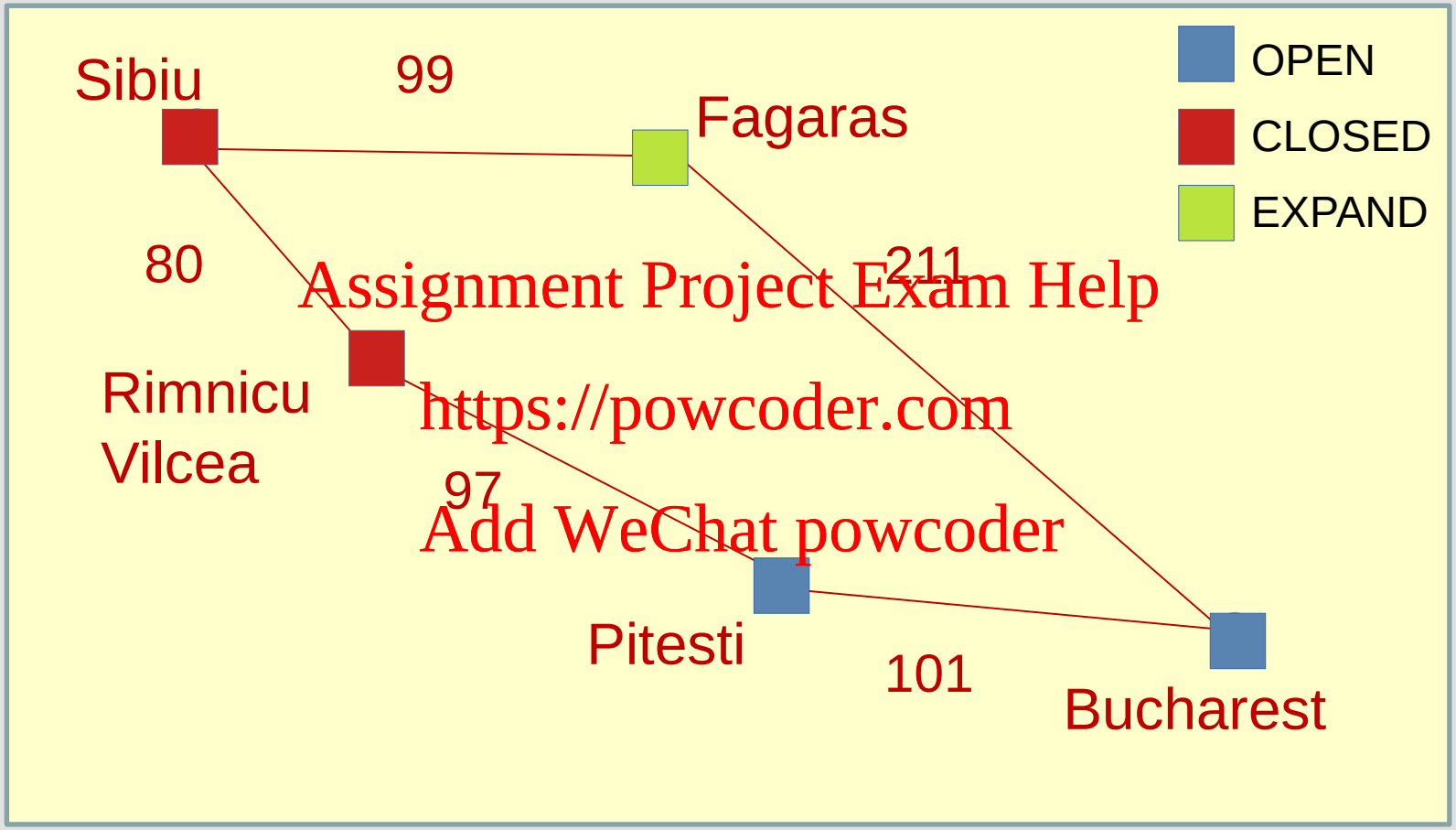
OPEN = { (Fagaras, 99), (Pitesti, 177) }
CLOSED = { (Sibiu, 0) }

Uniform-cost (Graph-) Search



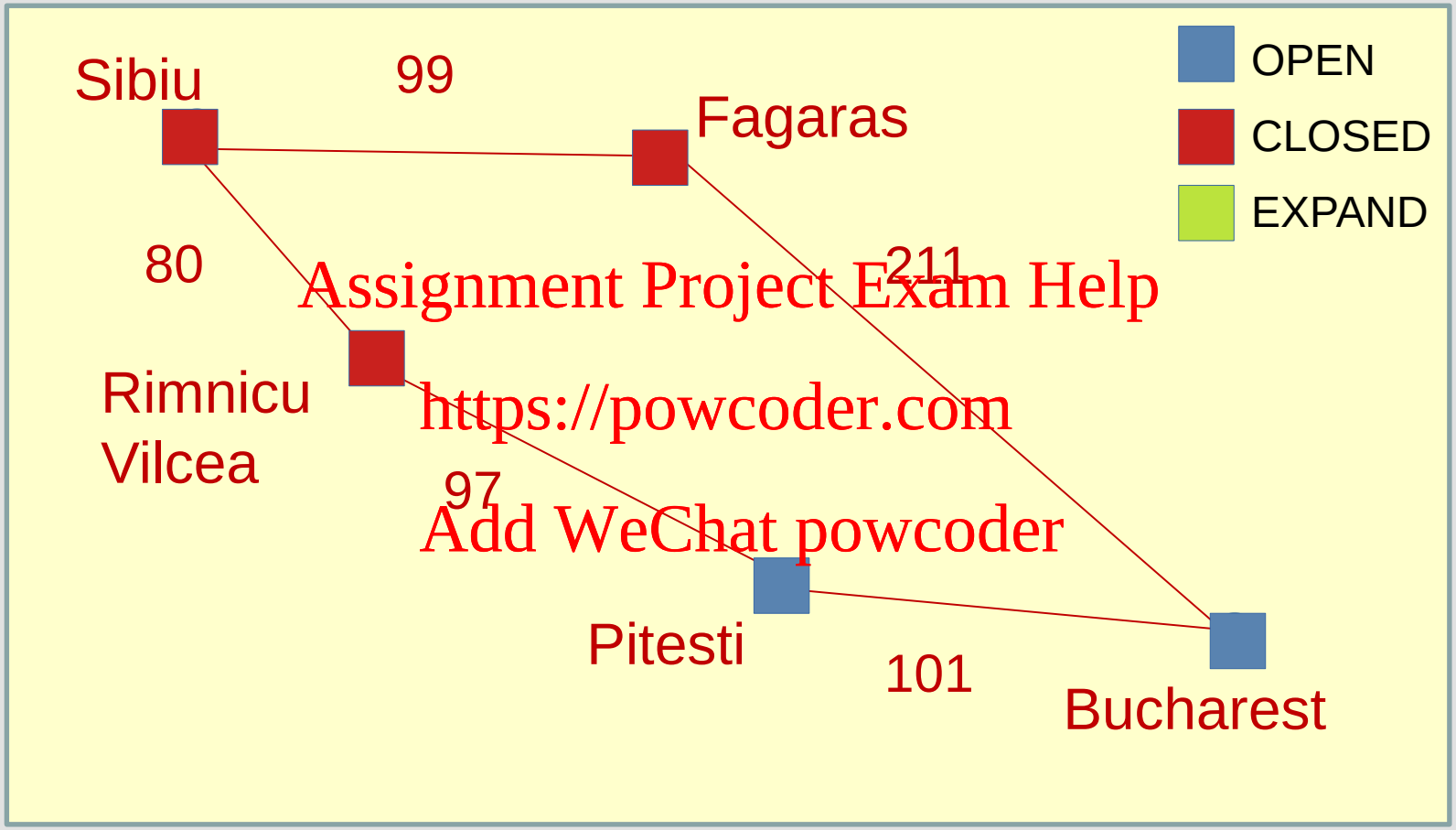
OPEN = { (Fagaras, 99), (Pitesti, 177) }
CLOSED = { (Sibiu, 0), (Rimnicul Vilcea, 80) }

Uniform-cost (Graph-) Search



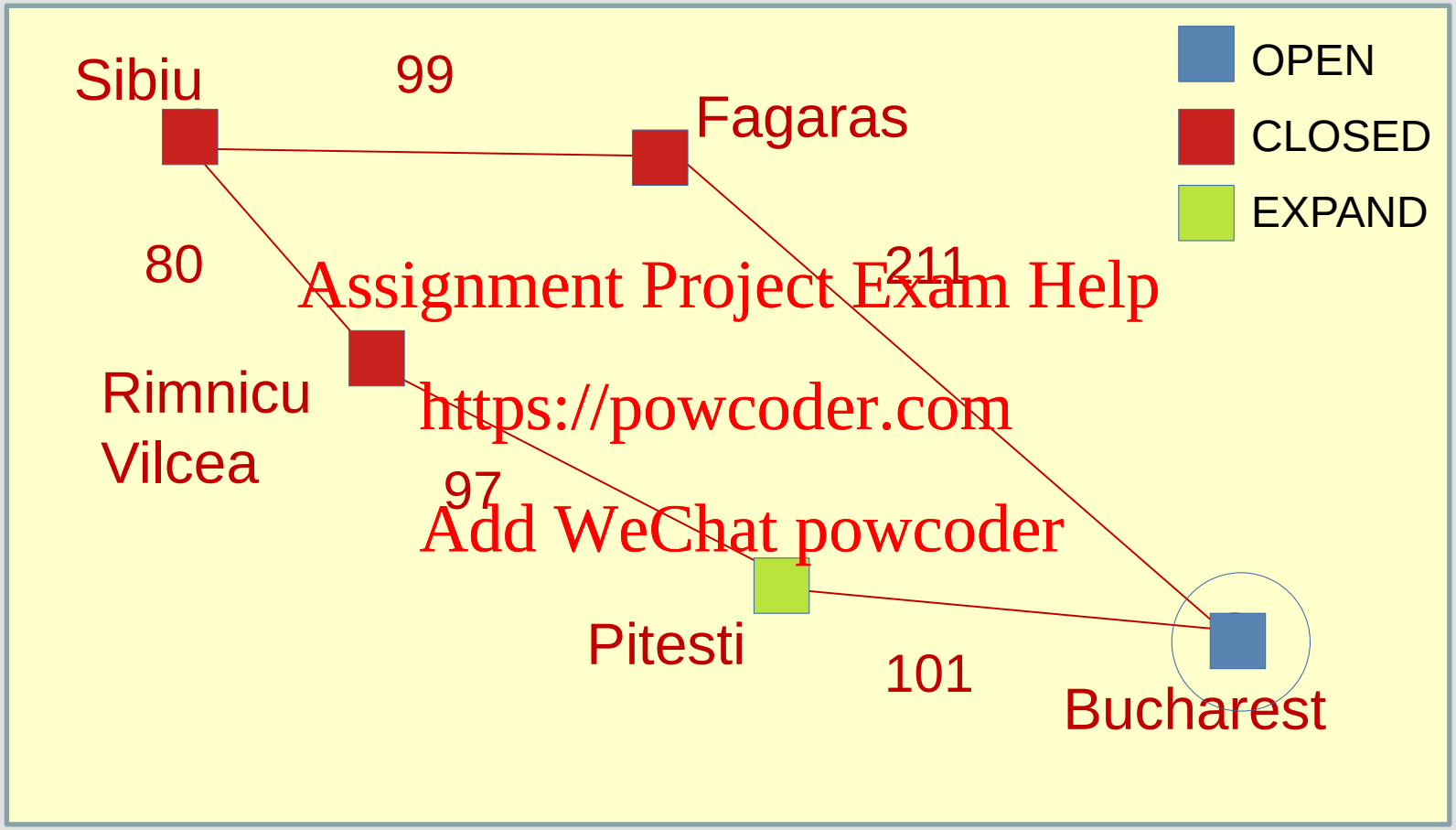
OPEN = { (Pitesti, 177), (Bucharest, 310) }
CLOSED = { (Sibiu, 0), (Rimnicul Vilcea, 80) }

Uniform-cost (Graph-) Search



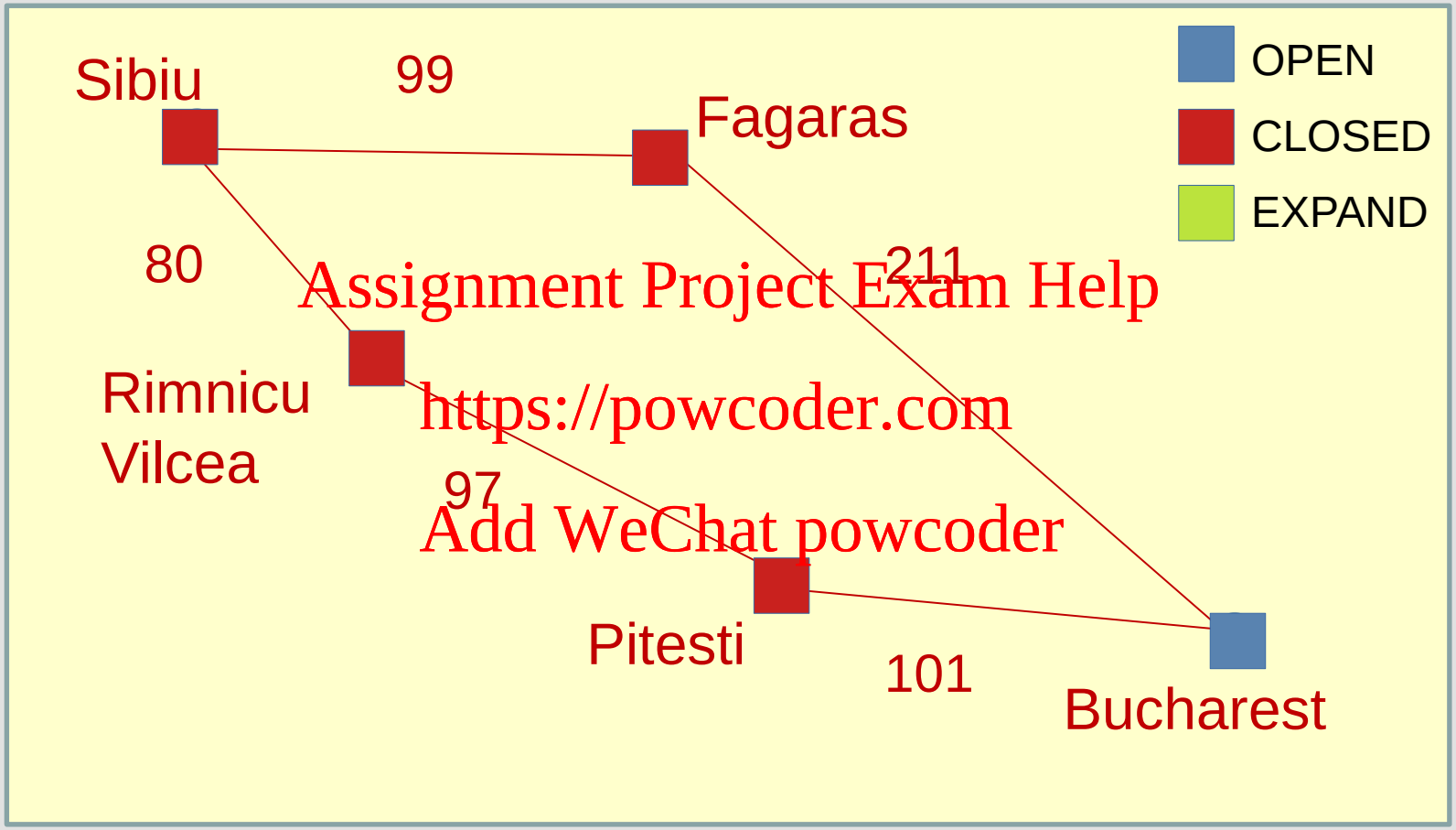
OPEN = { (Pitesti, 177), (Bucharest, 310) }
CLOSED = { (Sibiu, 0), (Rimnicul Vilcea, 80), (Fagaras, 99) }

Uniform-cost (Graph-) Search



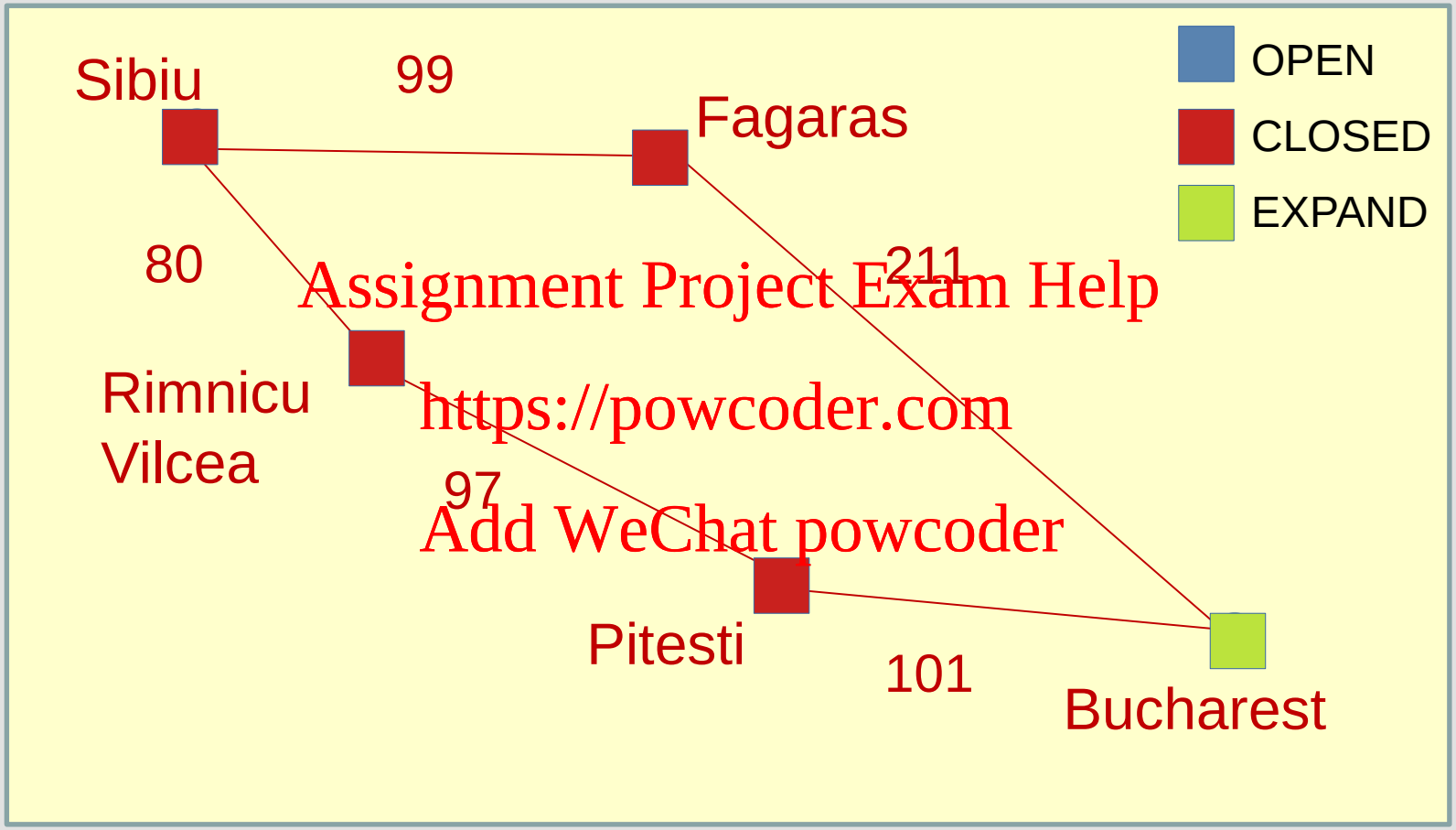
OPEN = { **(Bucharest, 278)** }
CLOSED = { (Sibiu, 0), (Rimnicul Vilcea, 80), (Fagaras, 99) }

Uniform-cost (Graph-) Search



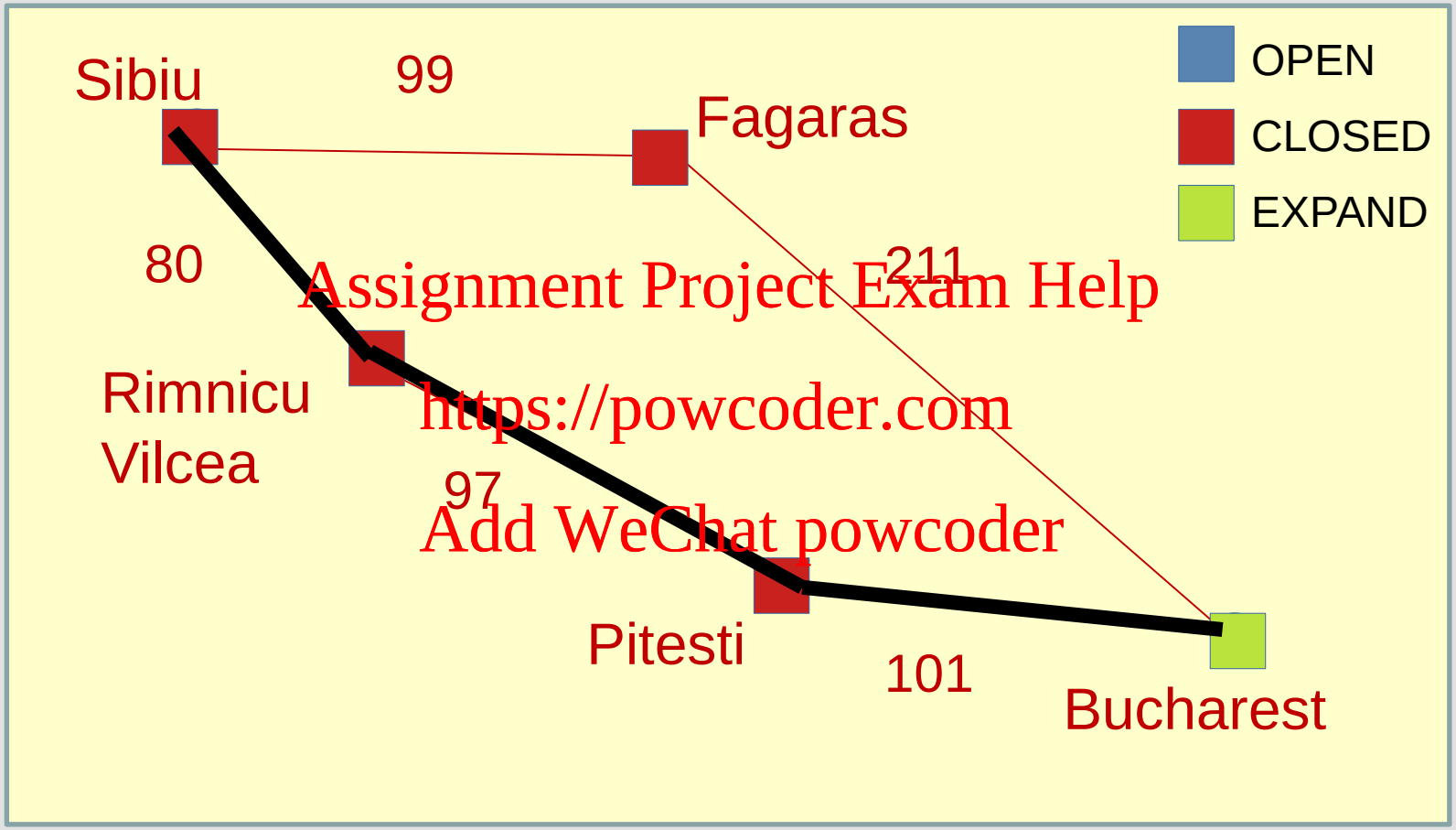
OPEN = { (Bucharest, 278) }
CLOSED = { (Sibiu, 0), (Rimnicul Vilcea, 80), (Fagaras, 99), (Pitesti, 177) }

Uniform-cost (Graph-) Search



OPEN = { (Bucharest, 278) }
CLOSED = { (Sibiu, 0), (Rimnicul Vilcea, 80), (Fagaras, 99), (Pitesti, 177) }

Uniform-cost (Graph-) Search



OPEN = { (Bucharest, 278) }
CLOSED = { (Sibiu, 0), (Rimnicul Vilcea, 80), (Fagaras, 99), (Pitesti, 177) }

Uniform Cost (Tree-) Search

Very similar, with some small changes:

- Selection strategy (minimum g-value) unchanged
- Cost updates are unchanged
- There is no CLOSED list!
- We can try some super simple duplicate detection
  - Never generate a parent
  - Never generate any ancestor

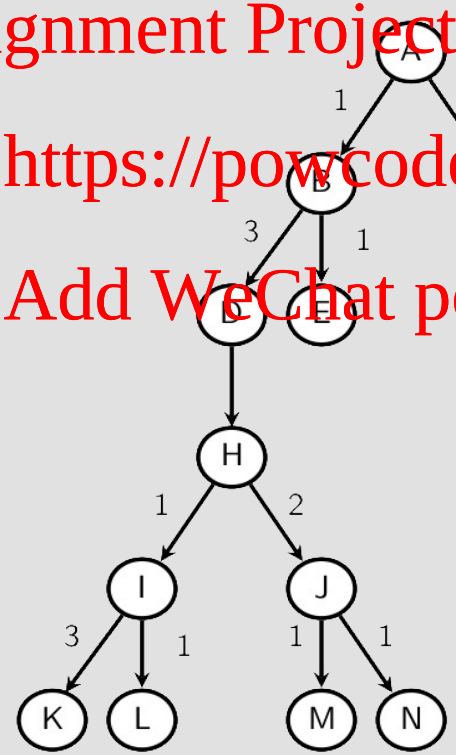
Properties of Uniform-cost Search

- **Complete?** Yes, if step cost ≥ ε
- **Time?** $O(b^{1+floor(C^*/\varepsilon)})$
  - where C^* is the cost of the optimal solution
- **Space?** $O(b^{1+floor(C^*/\varepsilon)})$
- **Optimal?** Yes – nodes expanded in increasing order of *g(n)* = cost of path to node *n*

Almost equivalent to BFS if step costs all equal!

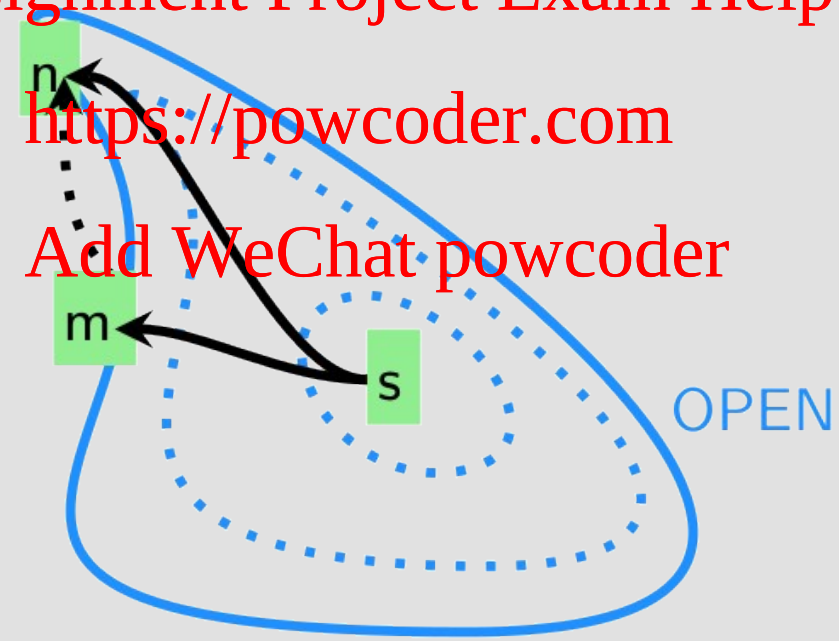
Properties of Uniform-cost Search

UCS may expand more nodes than BFS in practice. Why?



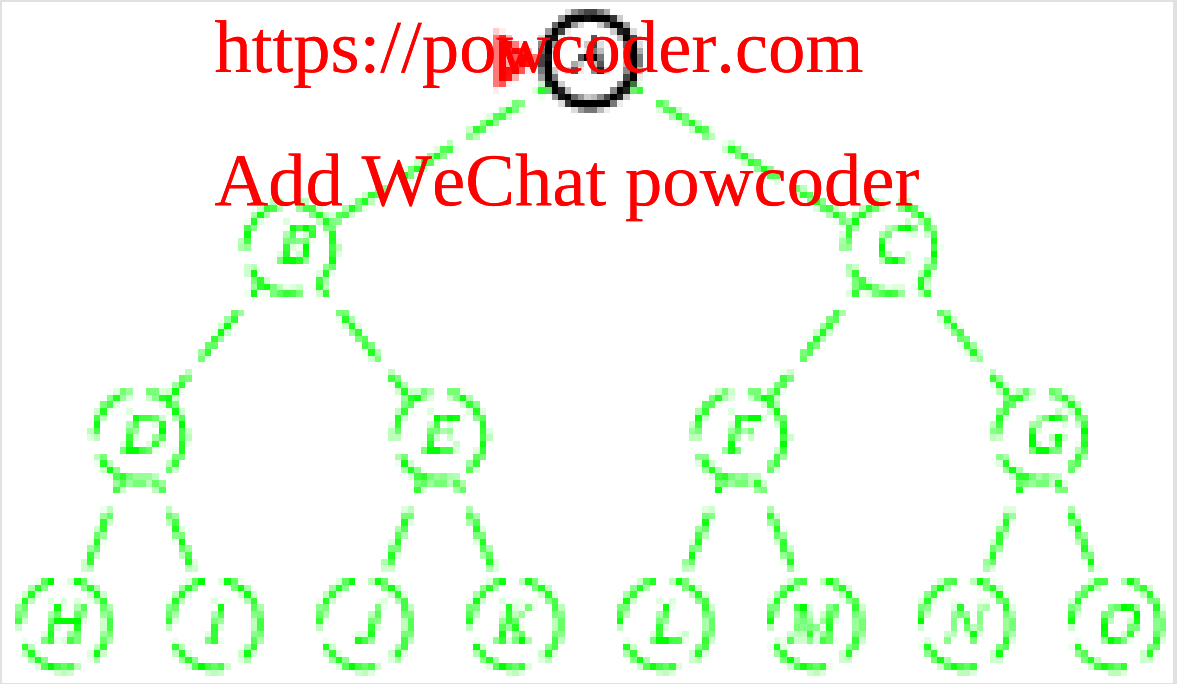
Properties of Uniform-cost Search

When UCS selects a node for expansion, the optimal path to that node has been found. Why?



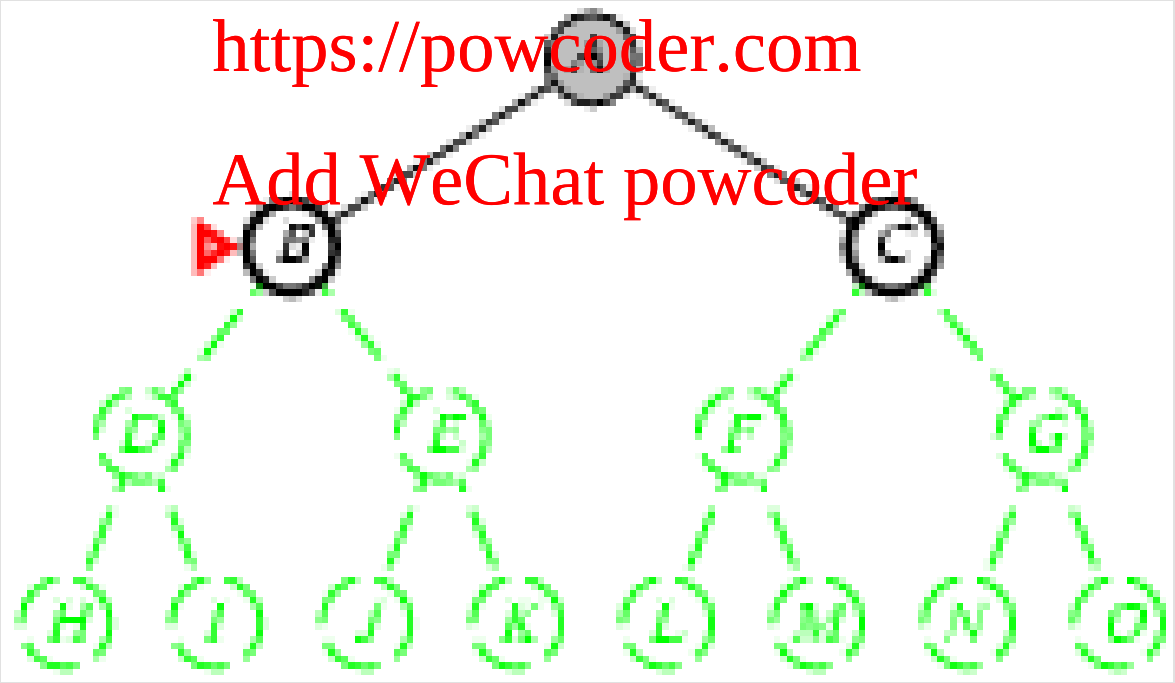
Depth-first Search (I)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



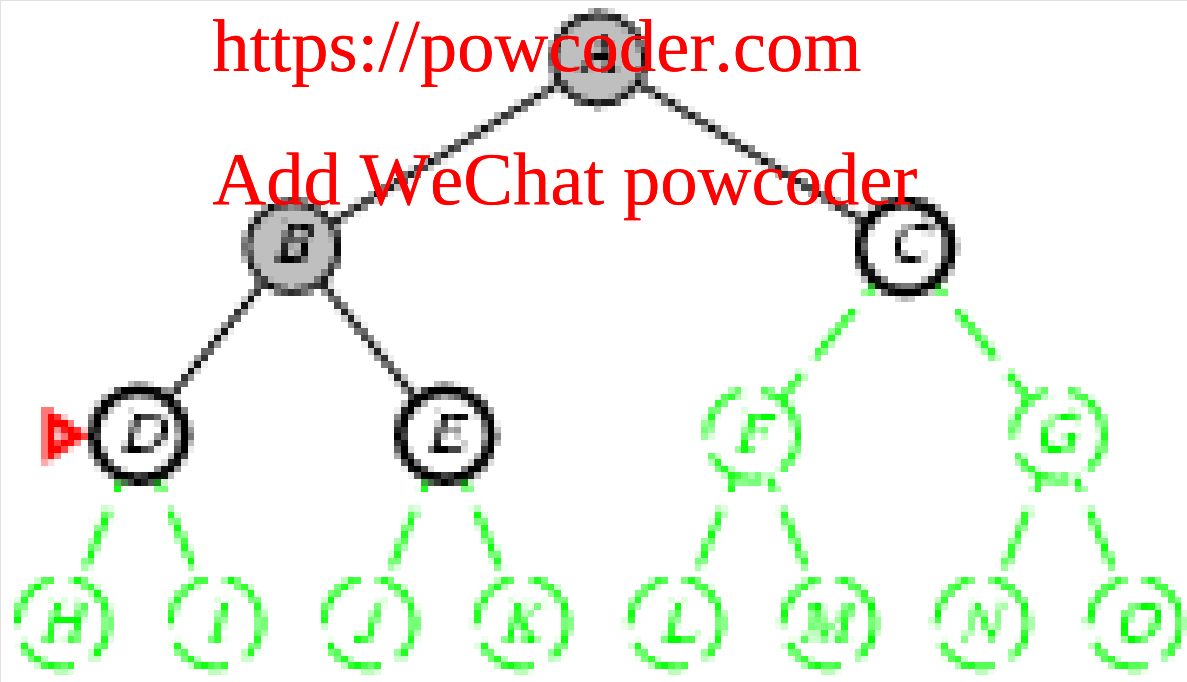
Depth-first Search (II)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



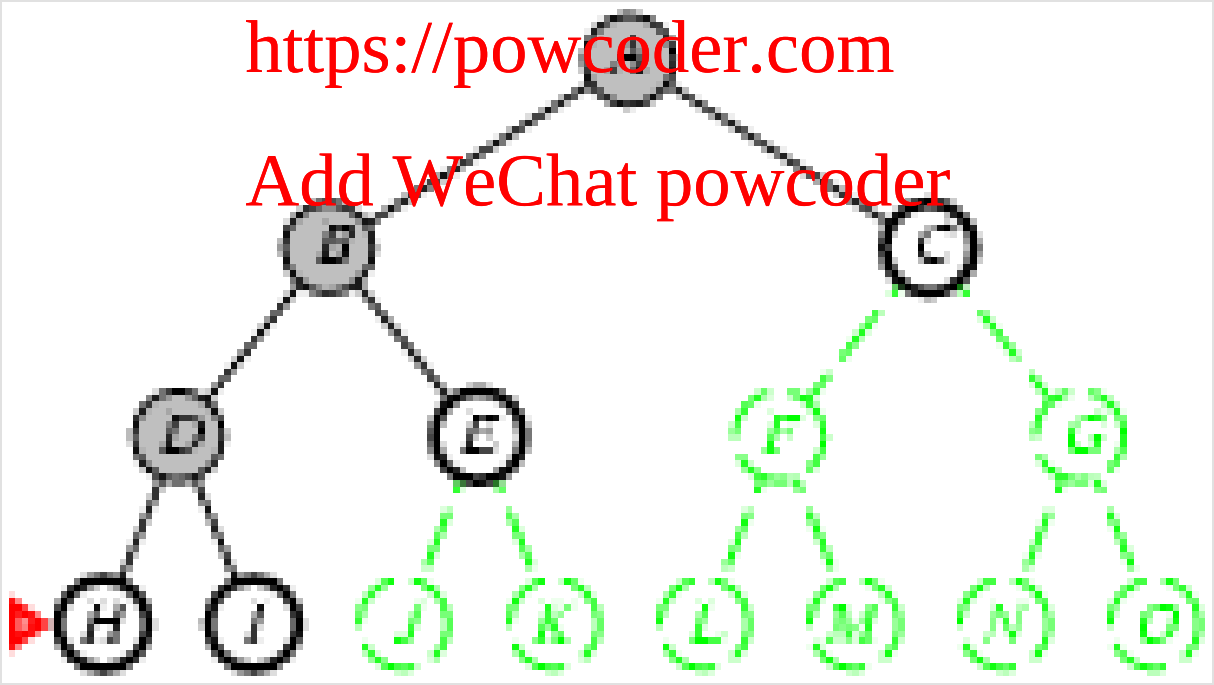
Depth-first Search (III)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



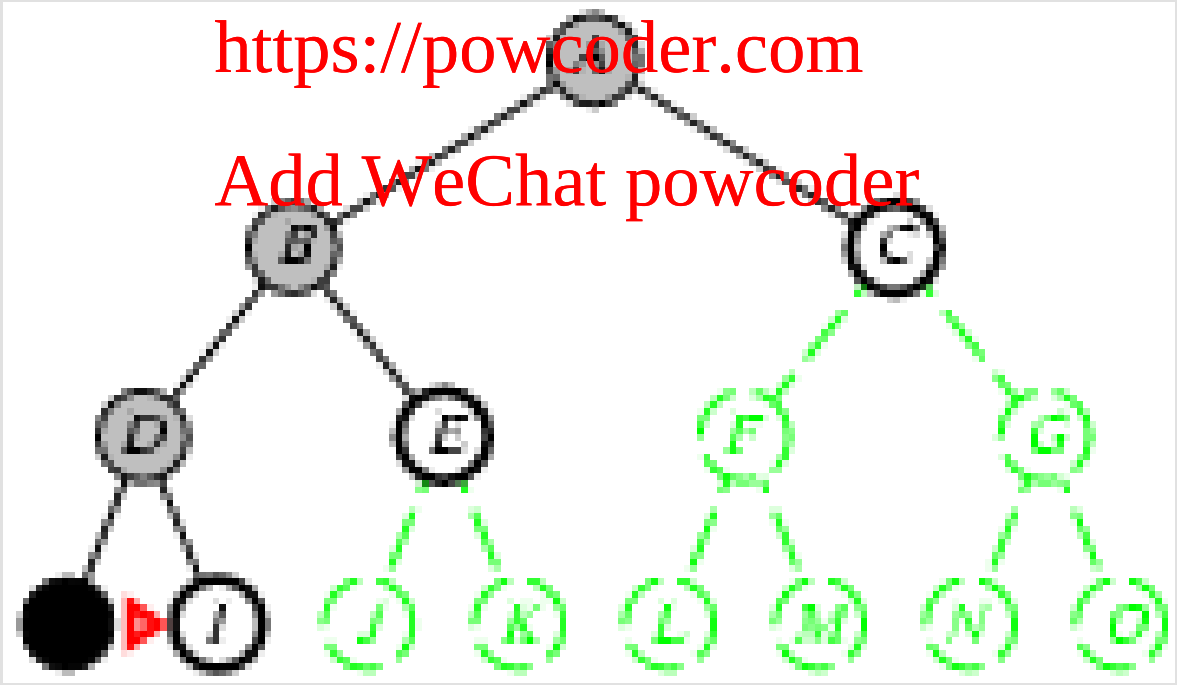
Depth-first Search (IV)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



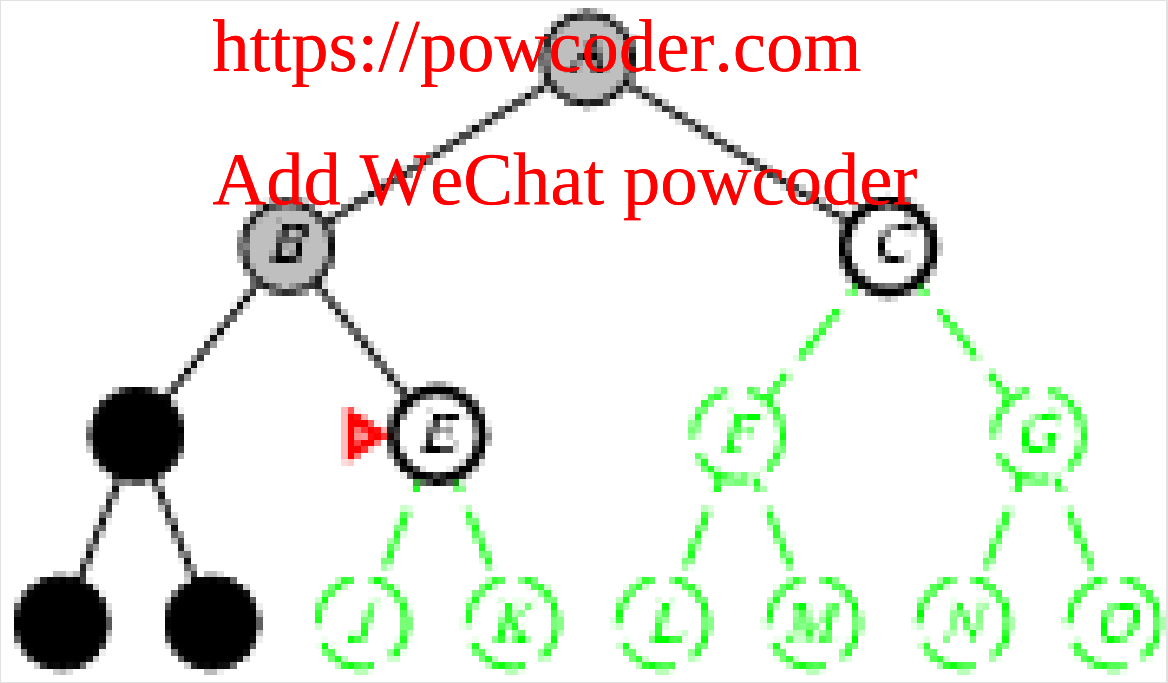
Depth-first Search (V)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



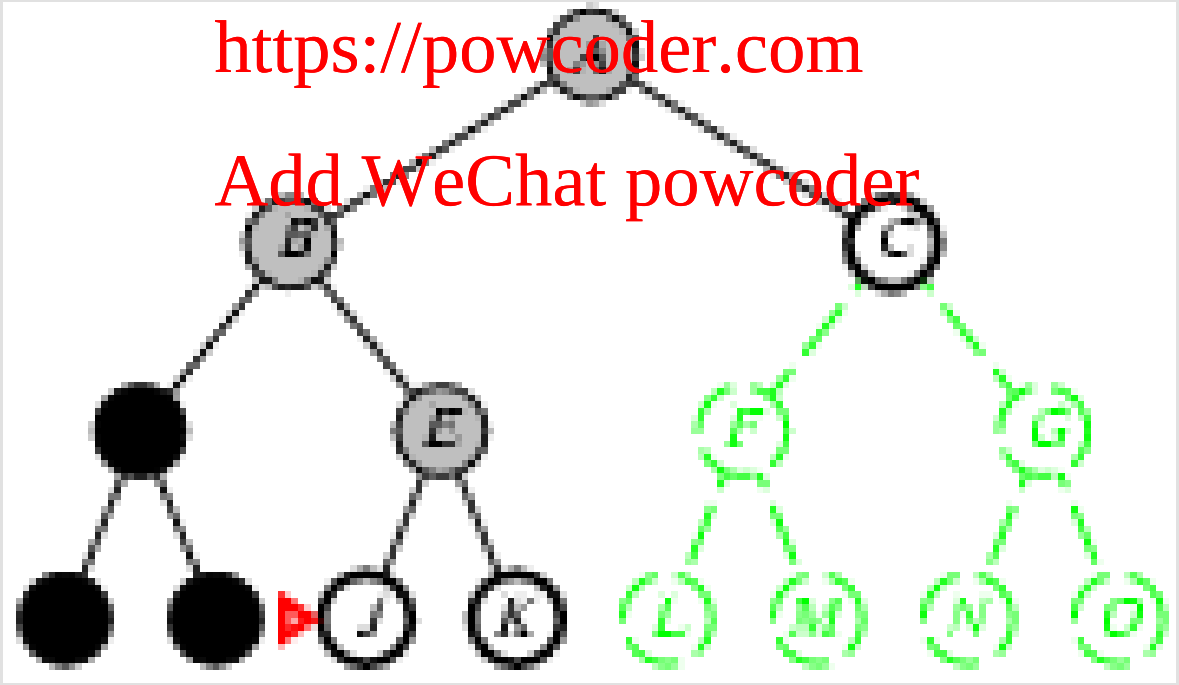
Depth-first Search (VI)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



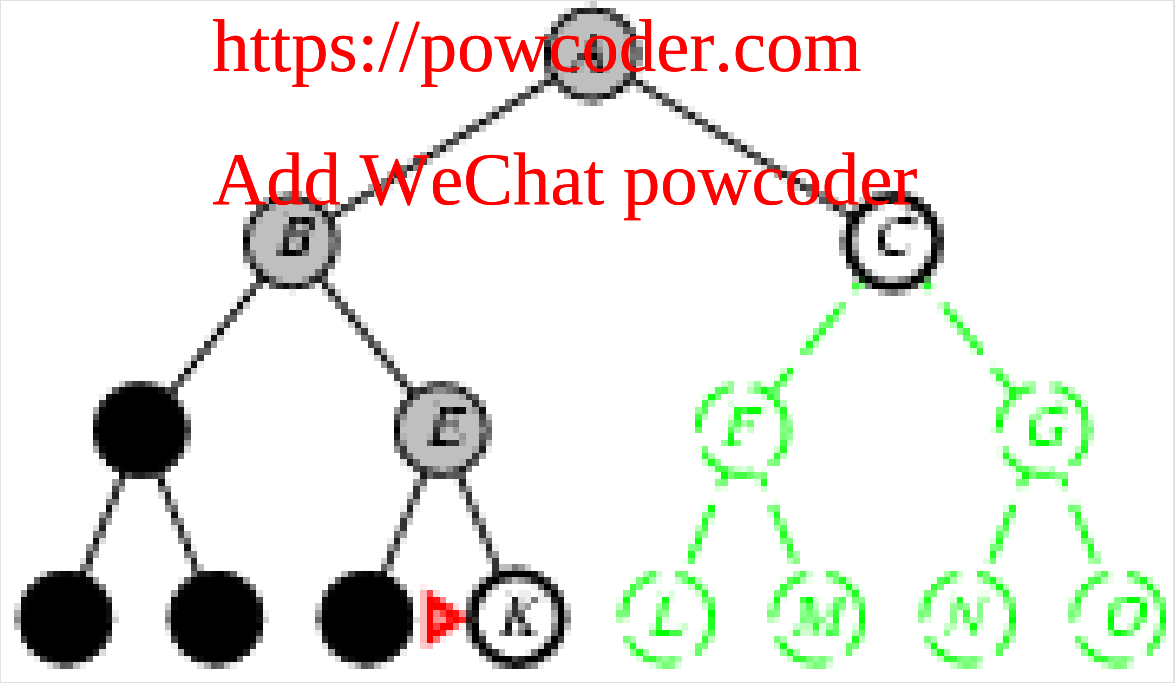
Depth-first Search (VII)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



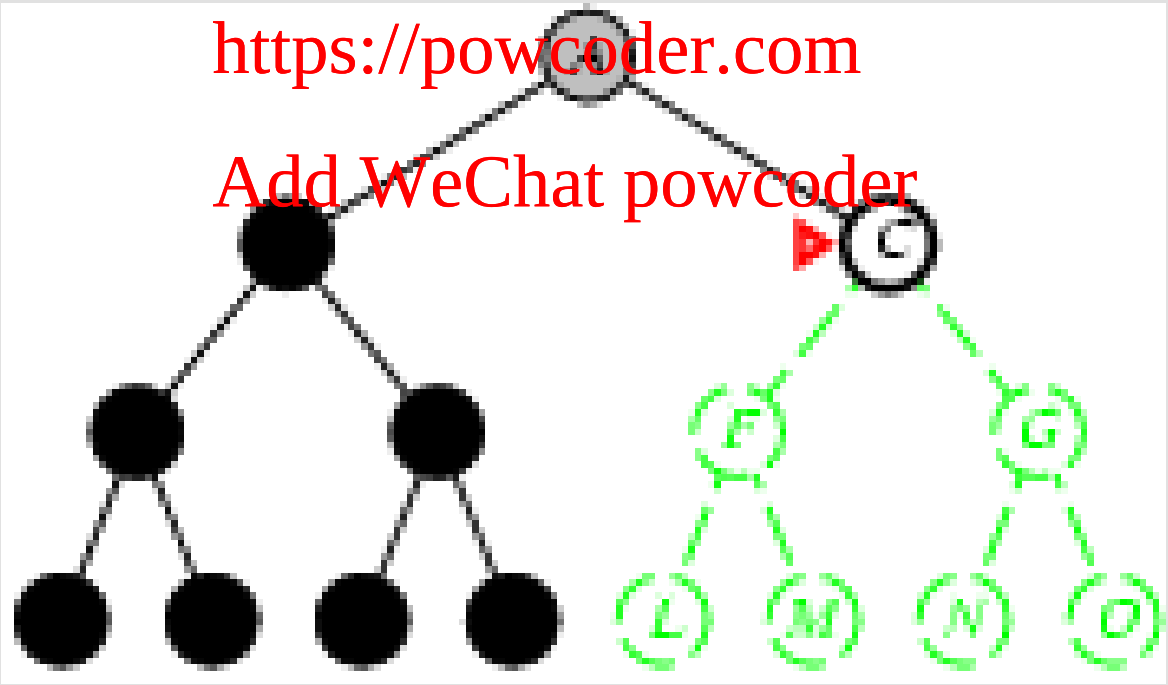
Depth-first Search (VIII)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



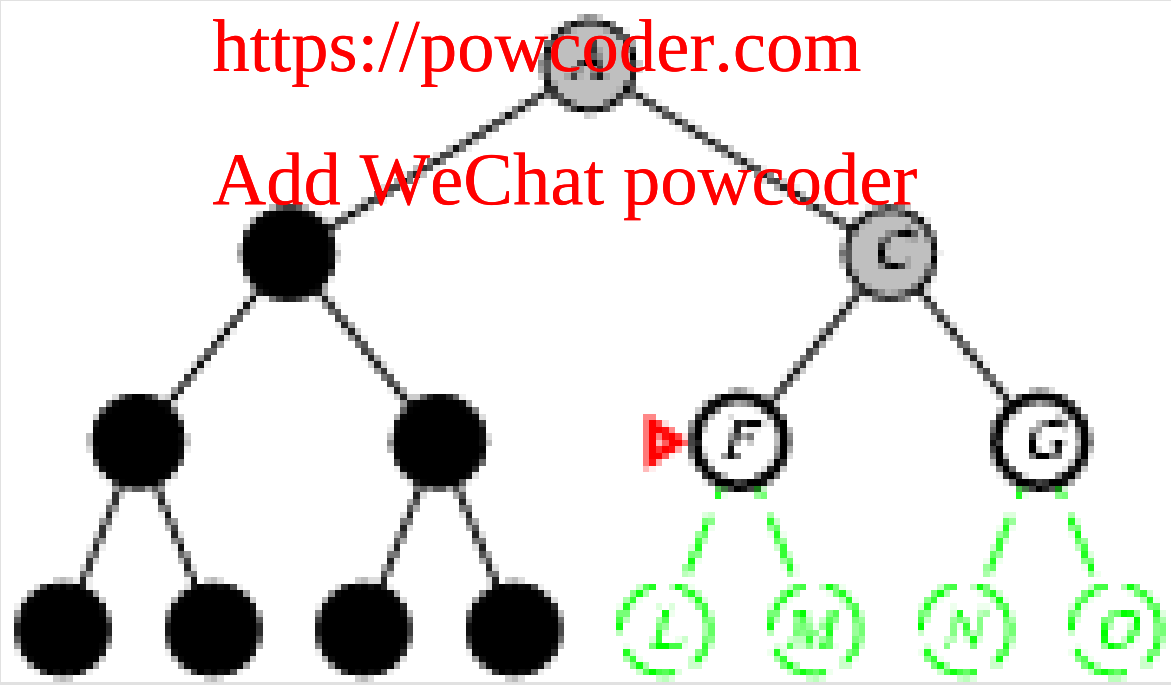
Depth-first Search (IX)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



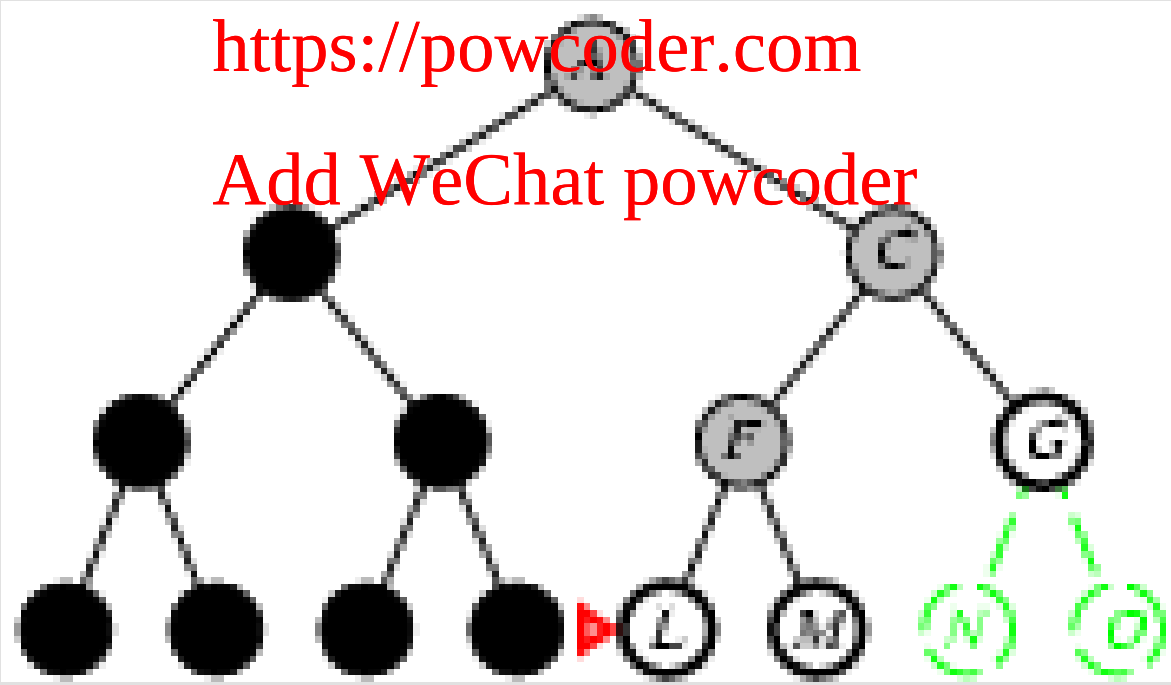
Depth-first Search (X)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



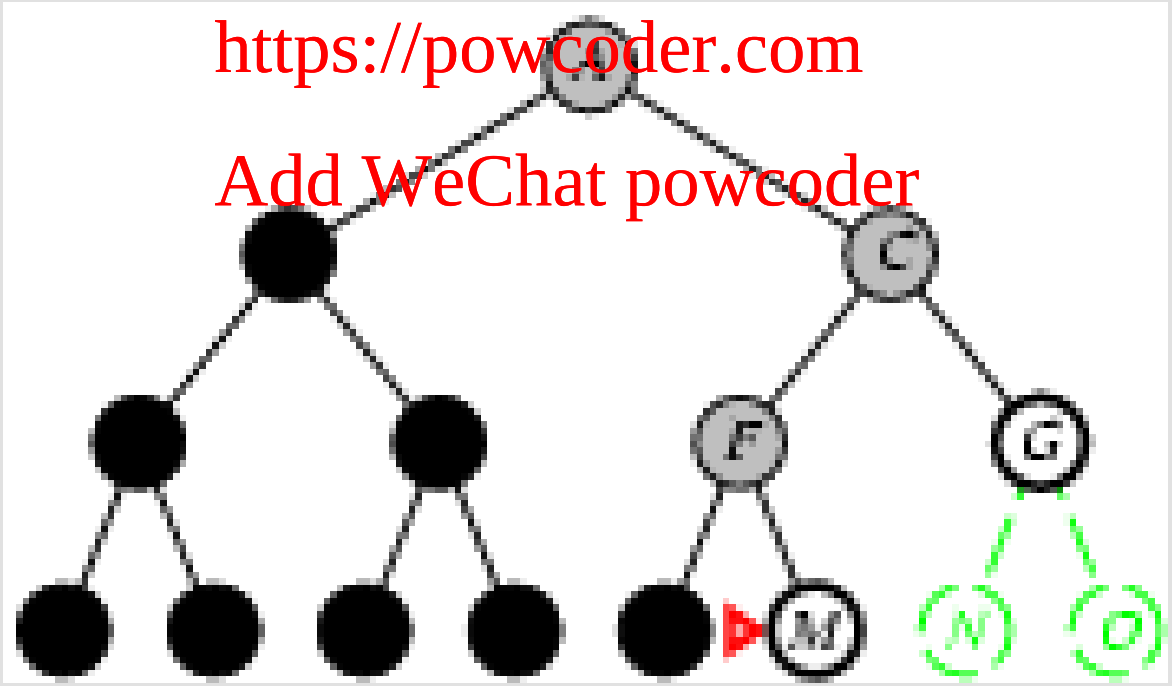
Depth-first Search (XI)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



Depth-first Search (XII)

- Expand the deepest unexpanded node
- Implementation: managing the frontier
  - QUEUEING-FN: LIFO – insert successors in front of queue



Properties of Depth-first Search

- **Complete?**
  - Infinite-state spaces: No
  - Finite-state spaces: Yes, if we check for ancestors
- **Time?** $O(b^m)$ (terrible if m is much larger than d!)
- **Space?** $O(bm)$, i.e., linear space
- **Optimal?** No

When all step costs are the same, will DFS find the optimal path?

Depth-limited Search

- Depth-first search with depth limit L
  - i.e., nodes at depth L have no successors
- **Complete?** No if $d > L$
- **Time?** $b + b^2 + b^3 + \ldots + b^L = b\frac{b^L - 1}{b-1} = O(b^L)$

also $= 1 + b + \ldots + b^L - 1 = \frac{b^{L+1} - 1}{b-1} - 1 = O(b^L)$

- **Space?** $O(bL)$
- **Optimal?** No

When all step costs are the same, will DLS find the optimal path?

Depth First Iterative Deepening

function DFID-SEARCH(*problem*) **returns** a solution or failure

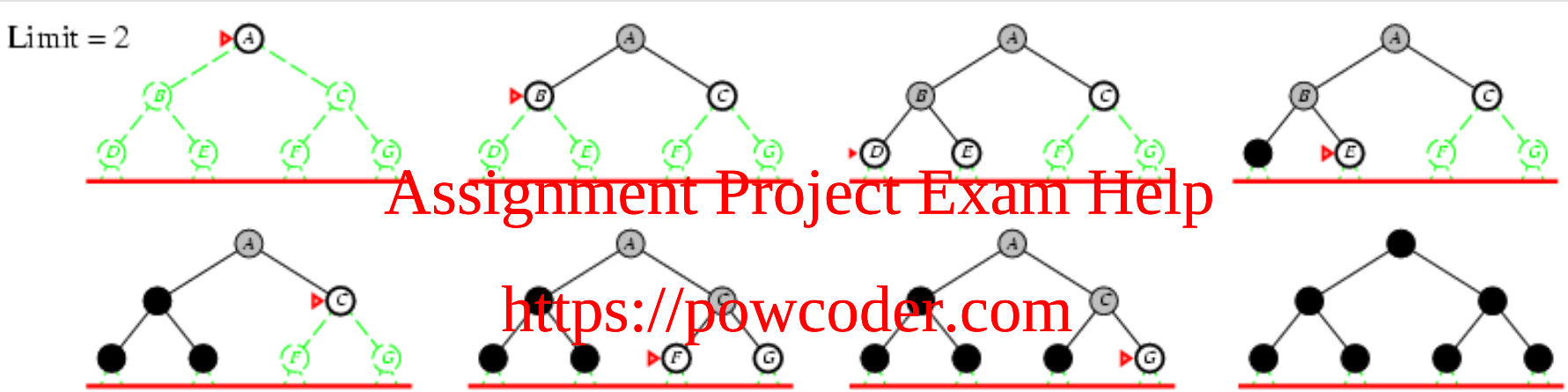
- Initialize the frontier using the initial state of *problem*
- **For** *depth* ← 0 **to** ∞
  - *result* ← DEPTH-LIMITED-SEARCH(*problem*,*depth*)
  - **if** *result* ≠ *cut-off* **then return** *result*
- **end**

indicates failure

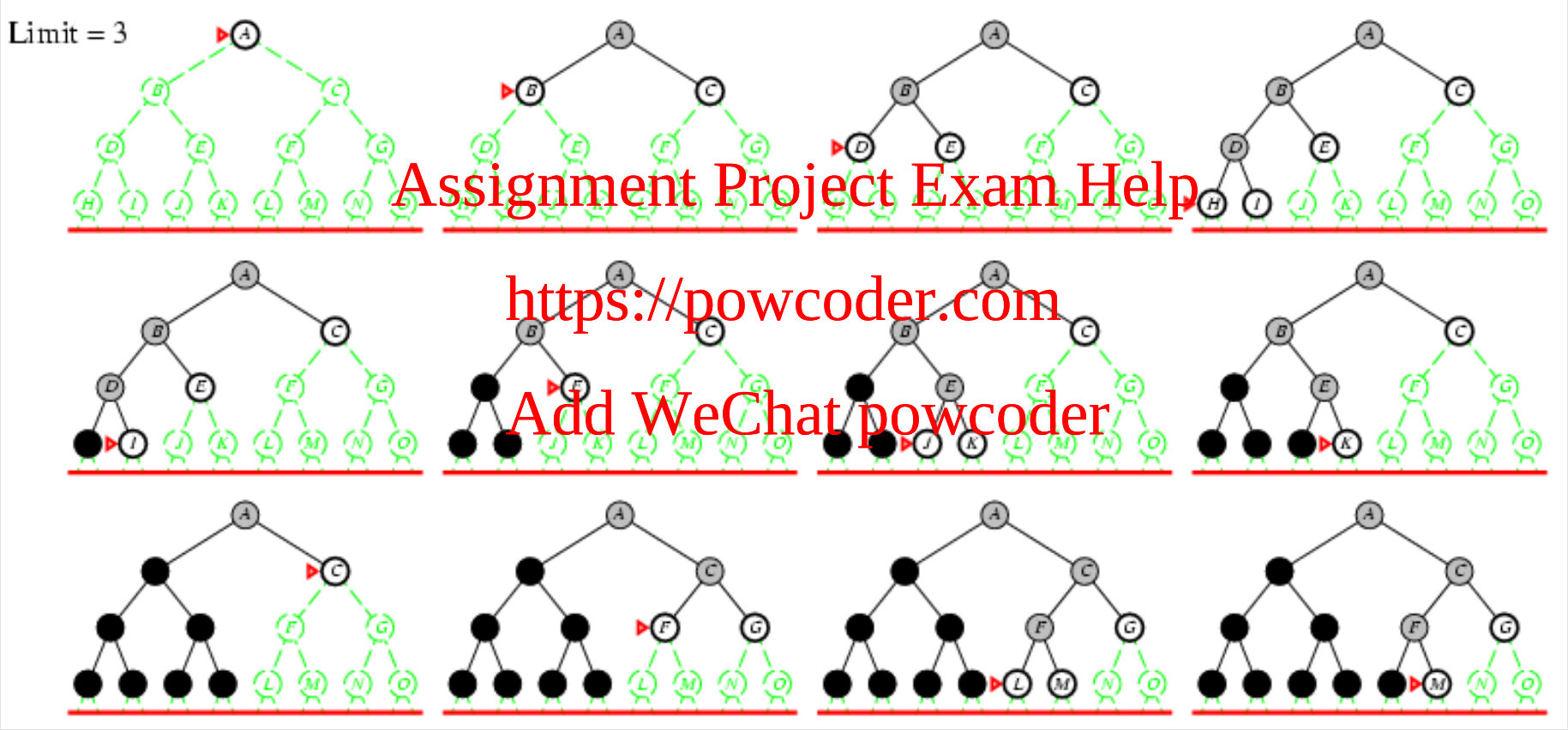
Example: DFID

Example: DFID

Example: DFID



Example: DFID



DFID Generated Nodes

- Number of nodes generated in a depth-limited search to depth d with branching factor b:

$$N_{DLS} = b + b^2 + b^3 + \ldots + b^d = b\frac{b^d - 1}{b-1} = O(b^d)$$

- Number of nodes generated in an iterative deepening search to depth d with branching factor b:

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

because we go to level 1 for $(d-1)$ occasions

- Example: For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
  - Overhead = $\frac{123{,}456 - 111{,}111}{111{,}111}$ = 11% in this case

Properties of DFID

- **Complete?** Yes
- **Time?**
  $db + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d = O(b^d)$
- **Space?** $O(bd)$
- **Optimal?** Yes, if step costs are identical

Demo Time

Which algorithm is faster in practice?

- BFS, DFS or DFID?

Let's race them!

https://www.movingai.com/SAS/EXP/

Re-cap: Problem Solving with Search

We have considered:

- Basic framework of a problem-solving agent
- Problem formulation
- Basic tree search over state spaces
- Graph search and pruning
- Uninformed search algorithms
  - > Breadth-first search (BFS)
  - > Uniform-cost search (UCS)
  - > Depth-first search (DFS)
  - > Depth-limited search (DLS)
  - > Depth-First Iterative Deepening (DFID)

A Family of Search Algorithms

- Tentativeness
  - Irrevocable – no reconsideration
  - Tentative – with reconsideration
- Informedness
  - Uninformed – decide based *only* on problem definition
  - Informed – use guidance on where to look for solutions

	Irrevocable	**Tentative**
Uninformed	Random Walk	Tree and Graph - Search (BFS, DFS, DLS, IDS, UCS)
Informed	Hill climbing, Local beam search, Simulated annealing, Genetic algorithms	Greedy best-first search, A, A*

Coming up week

- **Informed search**
  - Using heuristics on path cost to direct search
- **Reading for next week (week 3):**
  - Russell and Norvig (4th edn), chapter 3.5 and 3.6
- **Reminder:**
  - Labs start this week (week 2), unit hurdle requirements, academic integrity policies (and practice quizzes) and potential consequences, special consideration process