Search 01

- Problem Formulation
- Uninformed Search
- Informed Search

PRAKARN UNACHAK

Outline

- Problem Formulation
  - Prepare problem for search
  - Goal, state, actions, transition model, cost
- Tree Search
- Uninformed Search
  - Only use information in problem definition
- Informed Search
  - Use information beyond problem definition
  - Approximate how “good” a state is

Problem Formulation

Overview of Searching

We want to find solution(s) from initial state (starting point) to goal state

- At a state, we can perform some actions, each will take us to another (or the same) state
- Some problems want *path*: sequence of actions taken – best solution is solution with least path cost
- Some problems only want *configuration* (final state) that fit the description of goal

Problem Solving by Searching

- Goal-based or Utility-based Problem Solving
  - Either to reach goal, or state with better utility
- Problem definition provides description of
  - Where we start at
  - Which actions can be done, and when
  - (If important) Costs associated with actions
  - What is considered goal of this problem

Problem Solving by Searching

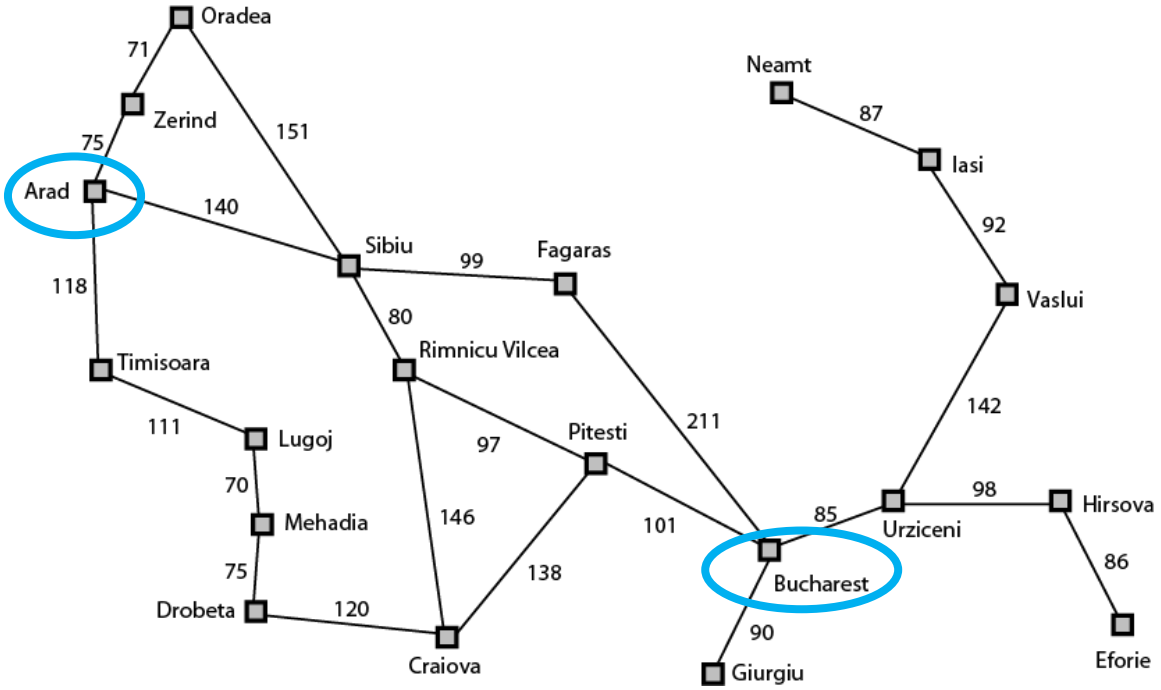
- Need to do:
  - Goal Formulation – What is consider a goal?
    - Define the meaning of the solution: what type of answer do we want?
    - Define what *goal state* should look like. Can be more than one
  - Problem Formulation – States and Actions
    - States: which information should a state contains?
    - Actions: Possible actions in each state. What will happen when an action is performed in a particular state?

Problem Type

- In this slide, we will consider *Single-state Problem*: deterministic and fully-observable, with discrete actions
- Deterministic: when an action is performed in a particular state, the resulting *successor state* will always be the same
- Fully-observable: there is always enough information at current time to know current state
- Discrete actions have limited number of possible actions

Example Problem - Romania

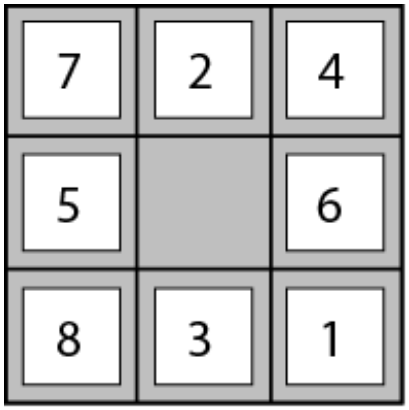
- Find a path with least distance from Arad to Bucharest



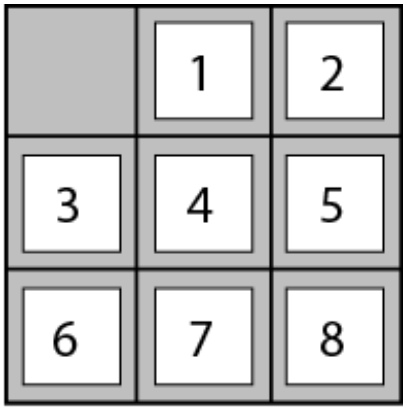
Credit: Artificial Intelligence: A Modern Approach

Example Problem – 8-Puzzle

- 8 numbered tiles on 3 × 3 board, with one blank space
- Can slide adjacent tile into blank space (swapping places)
- Find shortest sequence of moves from Start State to Goal State



Start State

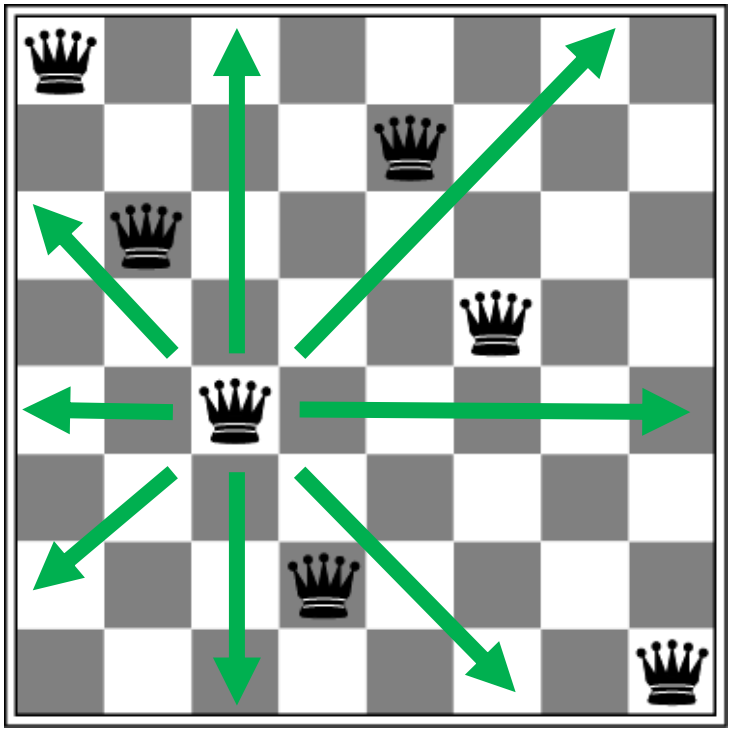


Goal State

Credit: Artificial Intelligence: A Modern Approach

Example Problem – n-Queen

- A queen can attack any piece on the same column, same row, or diagonal to it
- Find a way to place n queens on $n \times n$ board without any piece being able to attack another, does not care about path
- Two way to formulate:
  - **Incremental Formulation** – Place 1 queen at a time
  - **Complete-state Formulation** – Rearrange n queens already on the board



Credit: Artificial Intelligence: A Modern Approach

Selecting a State Space

- Sometimes, we don't need all details provided by the problem statement, especially real-world problems
- An *abstraction* approach is needed to reduce the details to the level we want (state and action)
- A good abstraction must be
  - Valid: we can expand any abstract solution into a solution in the more detailed world
  - Useful: it is easier to carry out each of the actions in the solution than the original problem

Single-state Problem Formulation

After we know what states look like

- Initial State – the starting point
- (Available) Actions – what can be done in each state
- Transition Model – what happen when an action is performed
  - If we can perform an action at a state → need to define successor state of that (state, action) pair
  - *State Space* is set of all states reachable from the initial states by any sequence of (allowed) actions
  - Solutions will (often) be in the form of *paths*: sequences of states linked by actions

Single-state Problem Formulation (cont.)

- Goal Test – a way to check whether a state is a goal state, can be a particular state, or a set of conditions to be met
- Path Cost Function – $g(n)$ – way to assign cost to path, usually we use sum of *step costs*: costs of actions at certain states

Example Problem Formulation - Romania

- **State**: At certain city on the map
- **Initial State**: “At Arad” or “Arad”
- **(Available) Actions**: Travel to adjacent city on the map
- **Transition Model**
  - We can show transition model in the form of Successor Function *S*(*x*) which is a set of pairs of <*action*, *successor state*>
  - For example, $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \langle Arad \rightarrow Sibiu, Sibiu\rangle, \langle Arad \rightarrow Timisoara, Timisoara\rangle\}$
- **Goal Test**: “At Bucharest”
- **Path Cost**: Sum of distance traveled

Example Problem Formulation – 8-Puzzle

- **States**: positions of tiles
- **Actions**: swap blank space with the adjacent tile: *Left*, *Right*, *Up* and *Down,* as possible
- **Transition model**: the swapping of the blank space and the tile
- **Goal test**: the given goal state
- **Path cost**: sum of step cost, which is 1 per move
  - **Possible States**: 9!/2 = 181,440 States

Example Problem Formulation – n-Queen

Incremental Formulation

- **States**: Board with 0 – *n* queen
- **Actions**: Place a queen on the position without one
- **Transition model**: Current board + one queen at designated position
- **Goal test:** 8 queens placed on the board without any piece being able to attack another
- **Path cost**: Does not care

Example Problem Formulation – n-Queen (cont.)

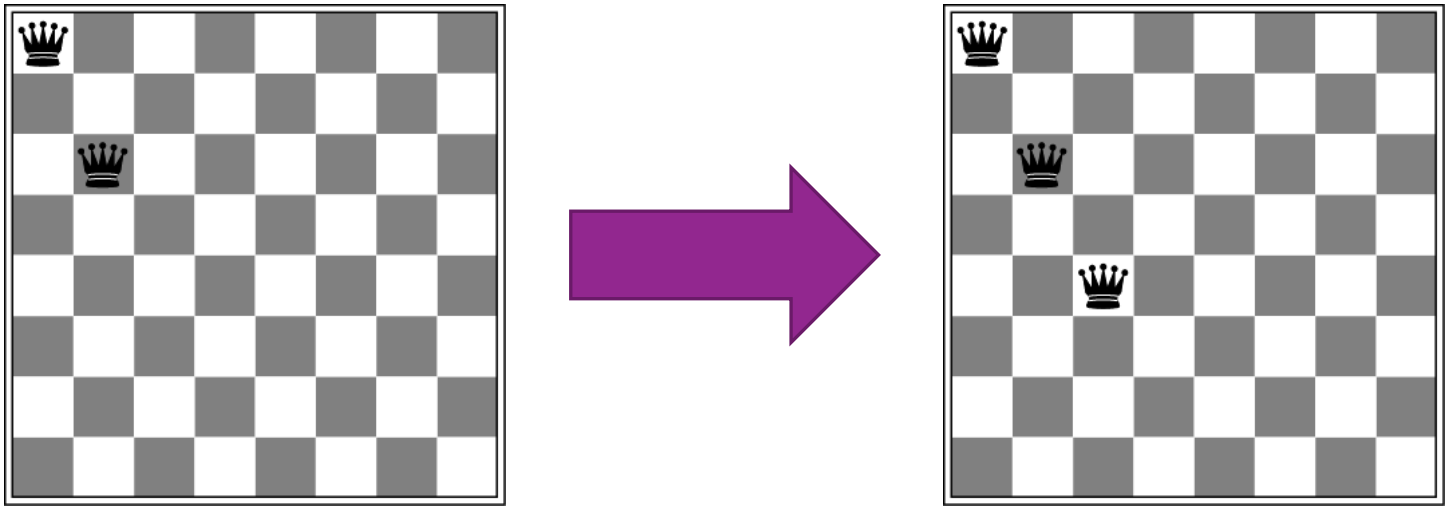
- For 8-Queen, if you can place queen anywhere, there are

$$64 \cdot 63 \cdot 62 \cdot 61 \cdot 60 \cdot 59 \cdot 58 \cdot 57 \approx 1.8 \times 10^{14}$$

possible sequences

- We can limit number of viable positions for each queen, one per column
  - **States**: Board with 0 – *n* queens, one on each leftmost column
  - **Actions**: Place a queen onto the leftmost unoccupied column, where it cannot attack any piece already placed

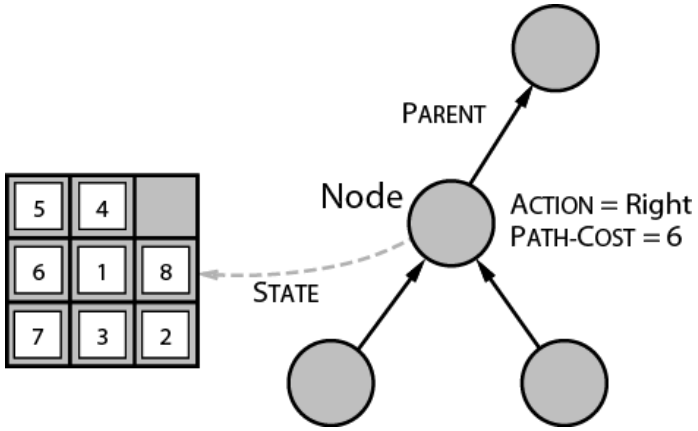
Example Problem Formulation – n-Queen (cont.)



- State space reduces from 1.8×10^{14} to 2,057

Tree (& Graph) Search

Tree Search



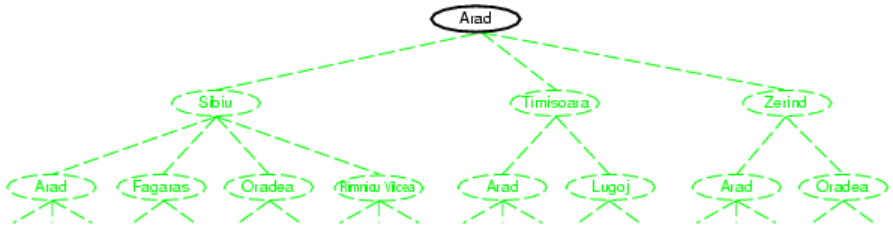
- Explore search space with search tree
  - Create nodes containing successor states (children) from already-explored state (parent) – *expanding* state
  - State Vs. Node
    - State represent status of the solution
    - Node is a data structure that is part of search tree
    - A node will contain: state, parent node, action, path cost, and depth (level)

Tree Search (cont.)

Performing Tree Search — Creating Search Tree

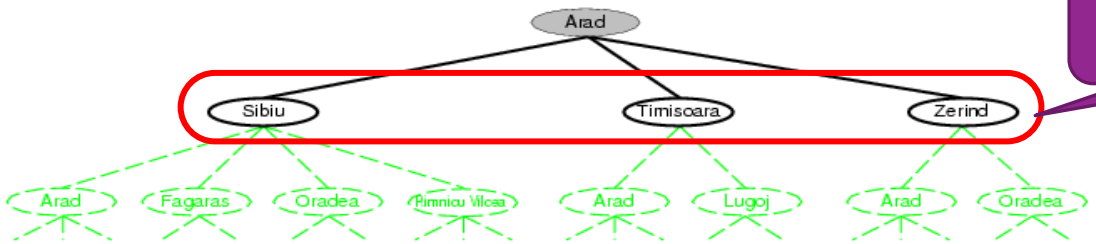
- Start at *root node*, containing initial state
- At each step, we pick a node to expand, based on *search strategy*
  - Perform actions that can be performed on the selected node, creating child nodes containing successors of this node
  - Repeat until a node pass the goal test
- Nodes without child node are called *leaf node*
- Set of unexpanded leaf node is called *frontier* (or *fringe* or *open list*)
  - We will pick node in this set to expand next
  - Frontier can be a particular data structure based on the search strategy

Tree Search Example – Romania



Initial State: At Arad

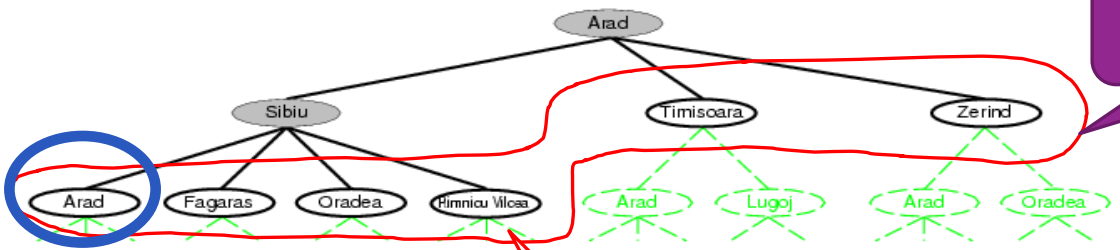
Tree Search Example – Romania (cont.)



Frontier Set

- Mark Arad as explored
- Expand Arad, Then expand a successor (Sibiu)

Tree Search Example – Romania (cont.)



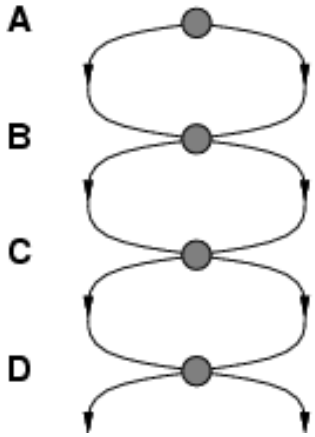
Frontier Set

- Mark Sibiu as explored
- Expand Sibiu, if not goal, expand a node in frontier set

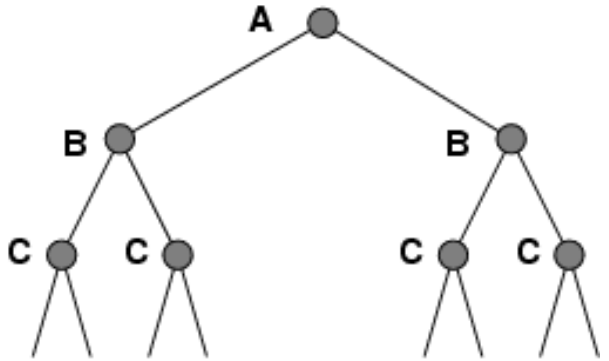
Nodes whose parent is explored will be added into frontier (fringe) to be chosen later

Repeated States and Graph Search

- In tree search: we cannot check nodes with the same state, can result in infinite loop or search tree growing exponentially
- Fix by using extra memory to remember visited state in *explored set* – performing *graph search*



Graph Search



Tree Search

Tree Search Algorithm

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 Strategies

- Search strategy will order the priority of nodes in frontier to be expanded
- To measure the effectiveness of search strategy, we use the following properties:
  - Completeness: If a solution exists, will it always be found?
  - Time Complexity: Number of nodes created
  - Space Complexity: Number of nodes kept in memory
  - Optimality: Is least-cost (best) solution guaranteed?
- Time and space complexity can be measured in:
  - b – Maximum Branching Factor of Search Tree
  - d – Depth of Least-cost Solution
  - m – Maximum Depth of Search Space (can be ∞)

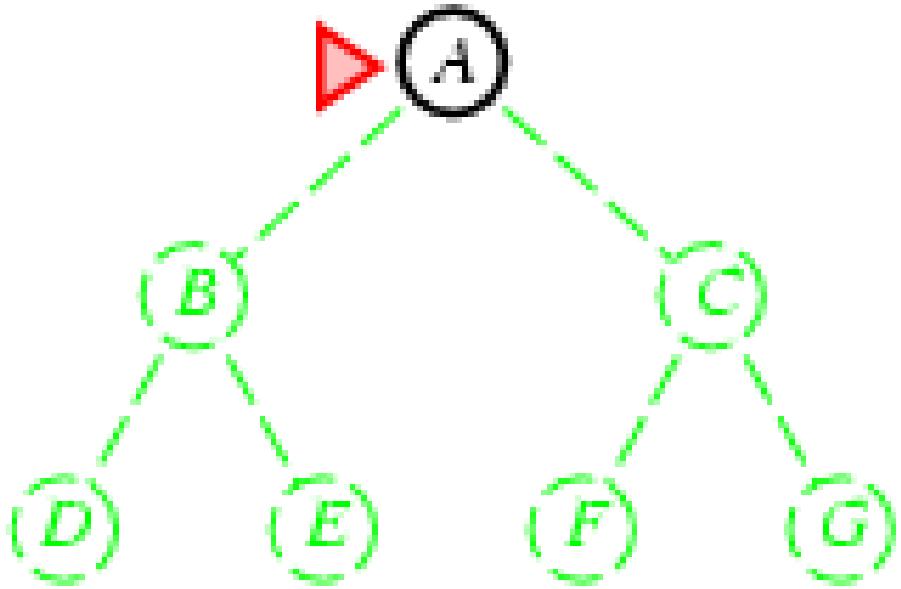
Searching
– Uninformed Search

Uninformed Search Strategies

- Uninformed search (blind search) strategy only use information in problem definition
  - Beside path cost, cannot tell how “promising” a non-goal state is
- Examples:
  - Breadth-first Search
  - Depth-first Search
  - Depth-limited Search
  - Iterative Deepening Search
  - Bidirectional Search
  - Uniform-cost Search

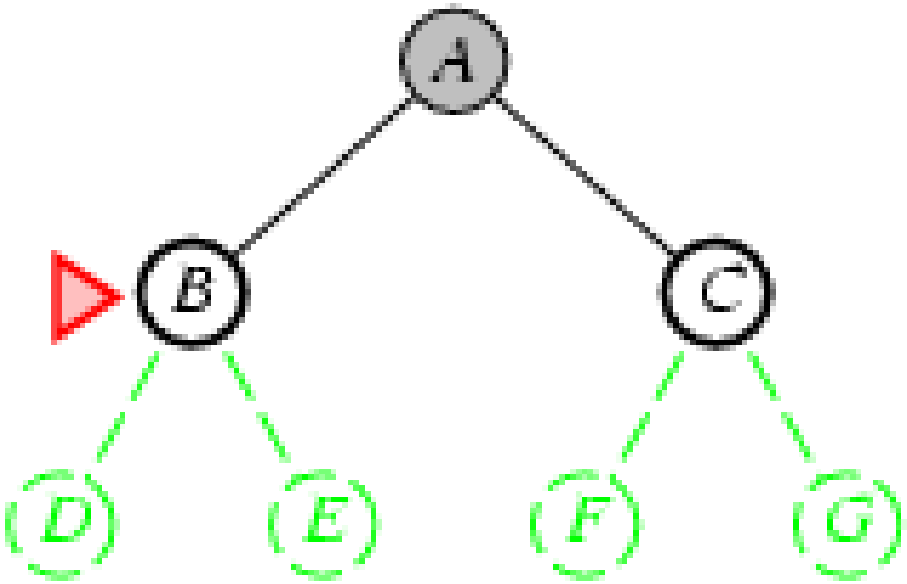
Breadth-first Search (BFS)

- Expand shallowest node (least depth)
- Implementation
  - Frontier is FIFO queue, new node will be inserted in the back of the queue



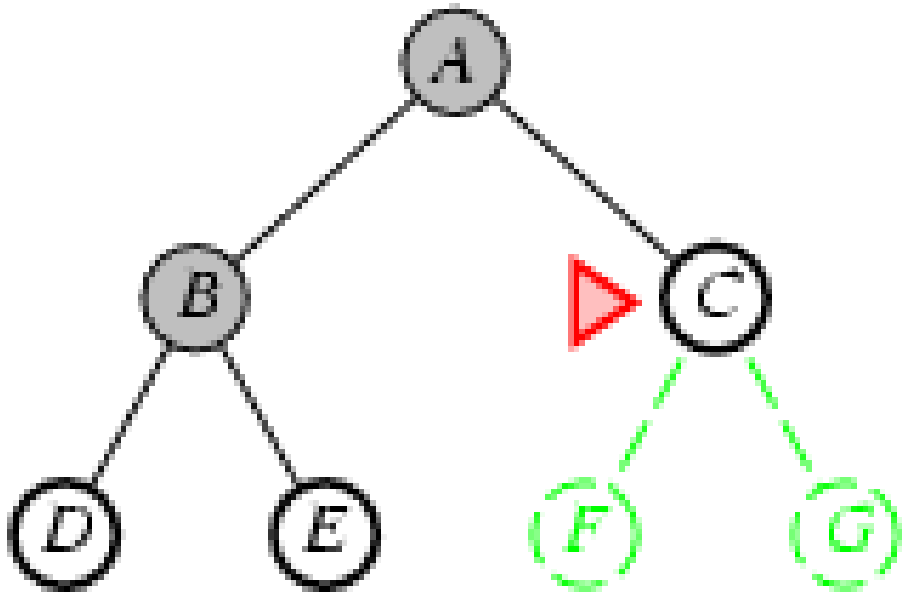
Breadth-first Search

- Expand shallowest node (least depth)
- Implementation
  - Frontier is FIFO queue, new node will be inserted in the back of the queue



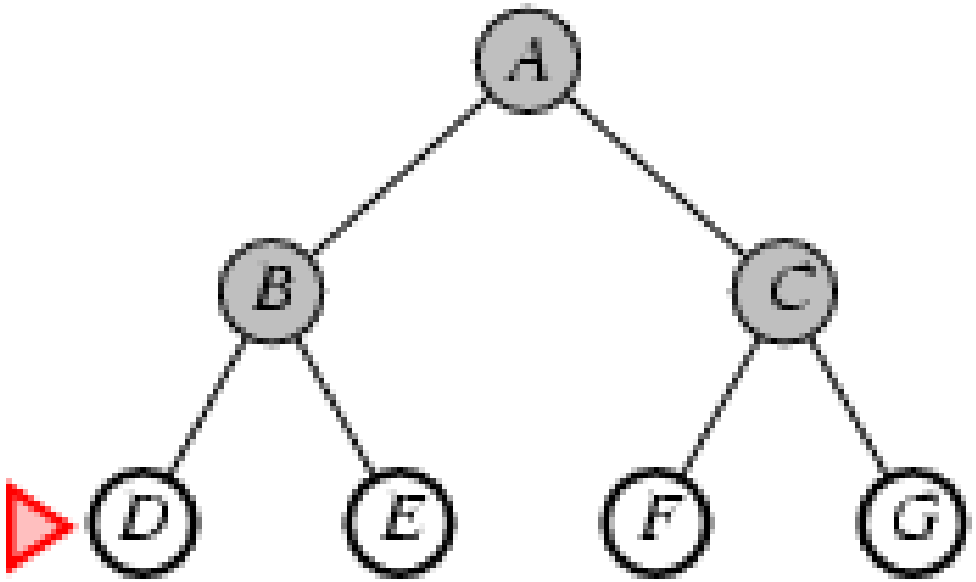
Breadth-first Search

- Expand shallowest node (least depth)
- Implementation
  - Frontier is FIFO queue, new node will be inserted in the back of the queue



Breadth-first Search

- Expand shallowest node (least depth)
- Implementation
  - Frontier is FIFO queue, new node will be inserted in the back of the queue



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)
```

Properties of BFS

- Complete? Yes (If b is limited)
- Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (Need to keep most/all nodes in the memory)
  - $O(b^{d-1})$ in explored set and $O(b^d)$ in frontier
- Optimal? Yes (if cost = 1 per step)
- Space is the main issue

Characteristics of BFS (cont.)

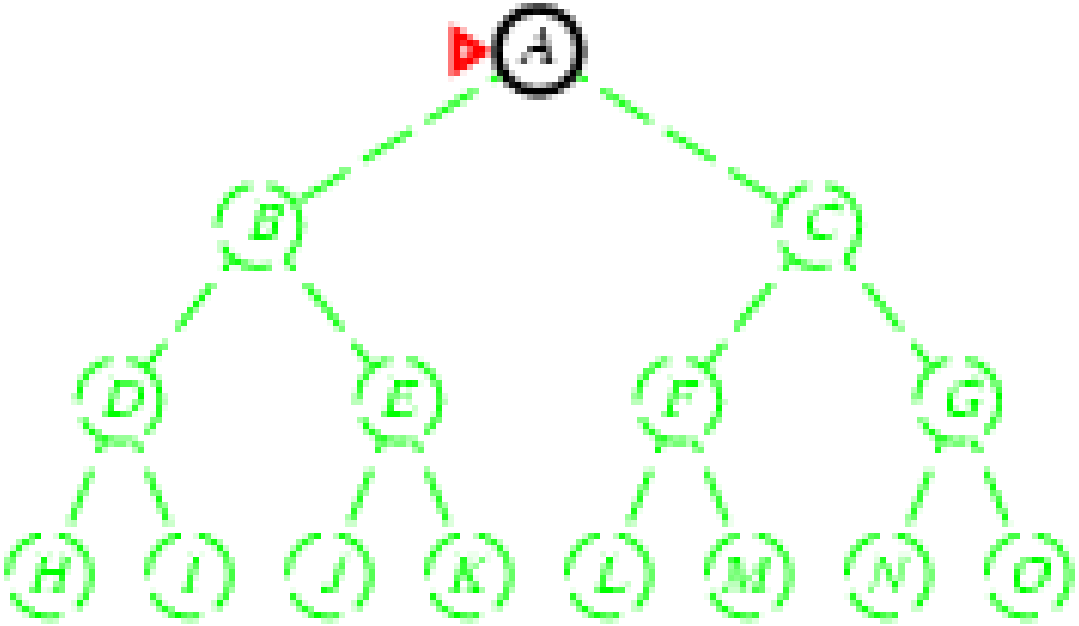
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Credit: Artificial Intelligence: A Modern Approach

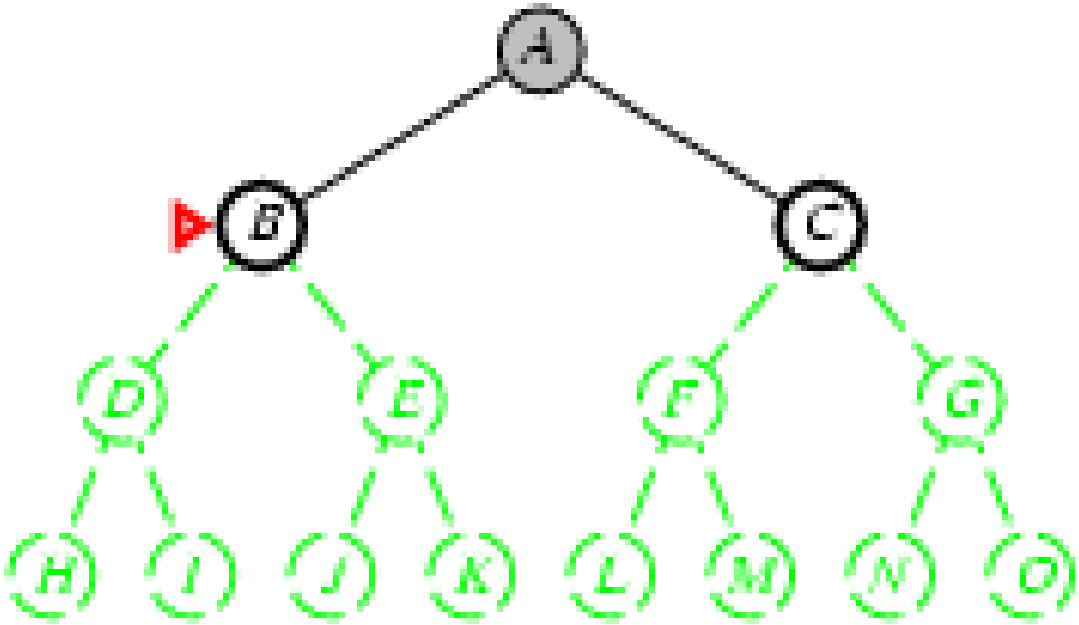
Depth-first Search (DFS)

- Expand the deepest node
- Implementation
  - Frontier = LIFO stack, successor will be place on top of the stack



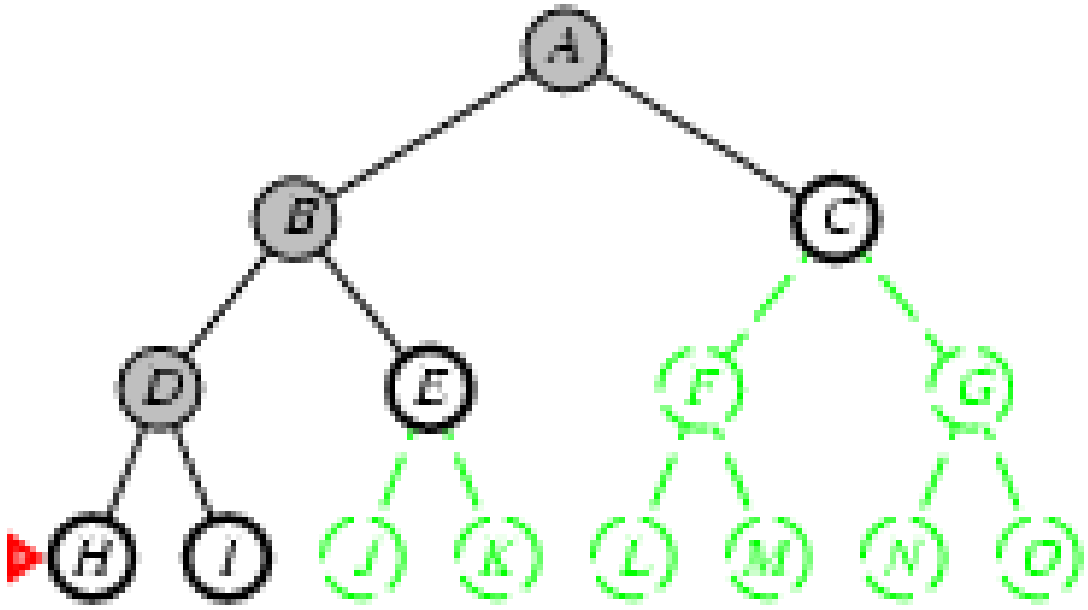
Depth-first Search

- Expand the deepest node
- Implementation
  - Frontier = LIFO stack, successor will be place on top of the stack



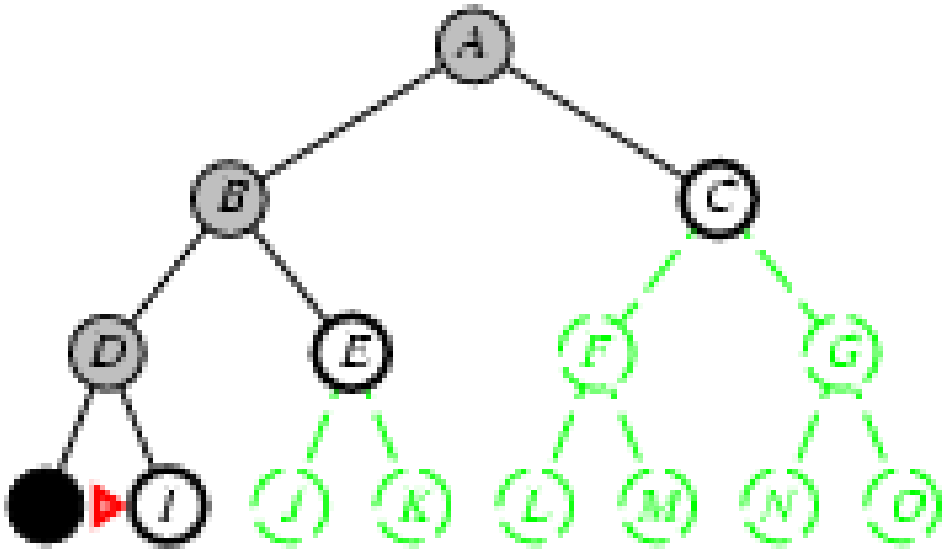
Depth-first Search

- Expand the deepest node
- Implementation
  - Frontier = LIFO stack, successor will be place on top of the stack



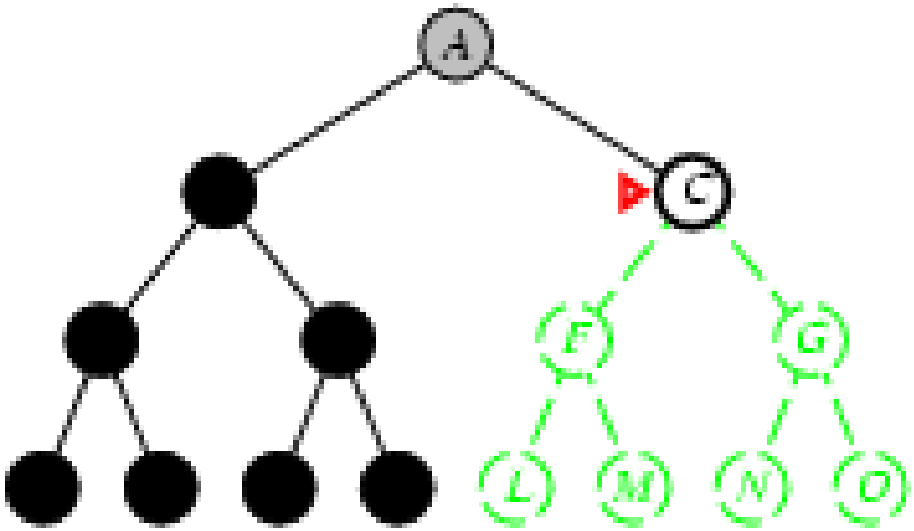
Depth-first Search

- จะ Expand Node ที่ลึกที่สุด
- Implementation
  - Frontier = LIFO Queue (Stack), Successor จะถูกวางไว้บนสุด



Depth-first Search

- จะ Expand Node ที่ลึกที่สุด
- Implementation
  - Frontier = LIFO Queue (Stack), Successor จะถูกวางไว้บนสุด



Characteristics of DFS

- Complete? No: Fails if the search space is infinite-depth or contain loops
  - If can be set to avoid repeated states in path → Complete In finite spaces
- Time? $O(b^m)$: Problem when m is much greater than d
  - Might be faster than BFS if solutions are dense
- Space? $O(bm)$ → Linear Space!
  - Keep only path and (much smaller) frontier
  - Expanded subtree can be delete
- Optimal? No

Depth-limited Search

- DFS that limit search depth to *limit*
  - Treat node at depth *limit* as having no children
  - No infinite loop, but does not guarantee finding a solution
- 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
```

Iterative Deepening Search (IDS)

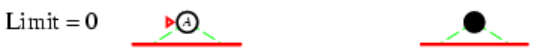
- Use depth-limited search, with iteratively increasing depth (*limit*)

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure

  inputs: *problem*, a problem

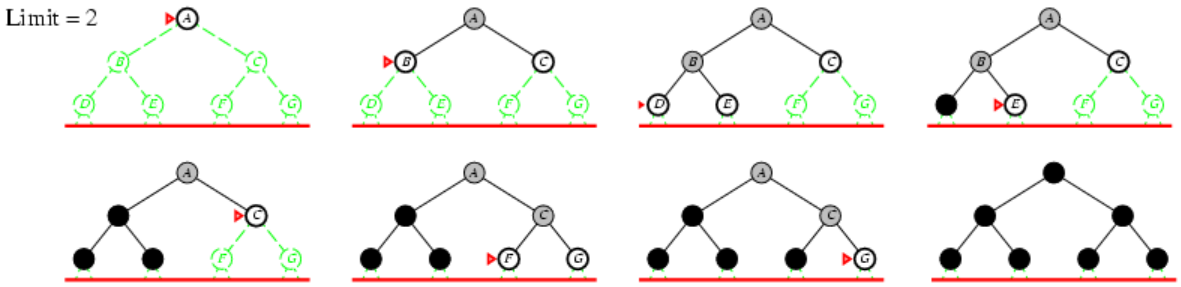
  for *depth* ← 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ cutoff **then return** *result*

Iterative deepening search *limit* =0

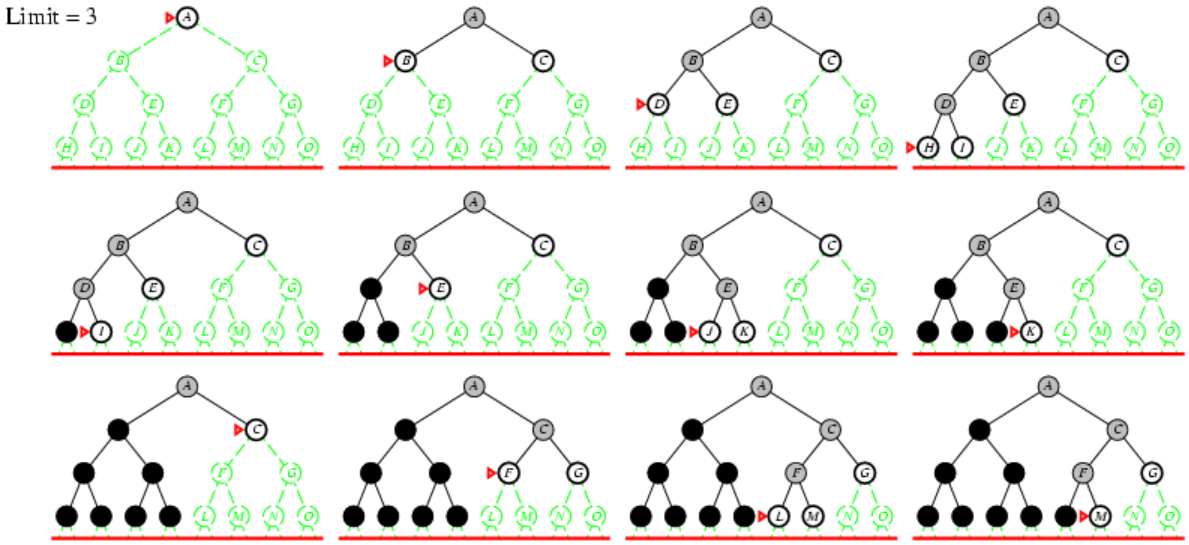


Iterative deepening search *limit* =1

Iterative deepening search *limit* =2



Iterative deepening search *limit* =3



Iterative Deepening Search (cont.)

Time-complexities

- Number of nodes created by BFS at depth *d* and branching factor *b*:

$$N_{BFS} = b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

  - Where each term is number of nodes created on each level (depth)

- Number of nodes created by IDS with limit at depth *d* and branching factor *b* :

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

  - Nodes at depth 1 are created *d* times. Nodes at depth 2 are created *d-1* times…

Iterative Deepening Search (cont.)

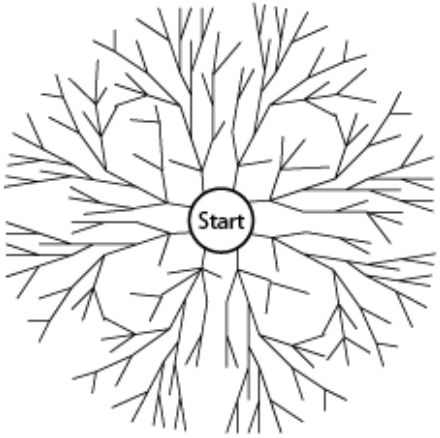
- Let $b = 10$, $d = 5$,
  - $N_{BFS} = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$
- Overhead = (123,456 - 111,111)/111,111 = 11%
- Because most created nodes are at the deepest level

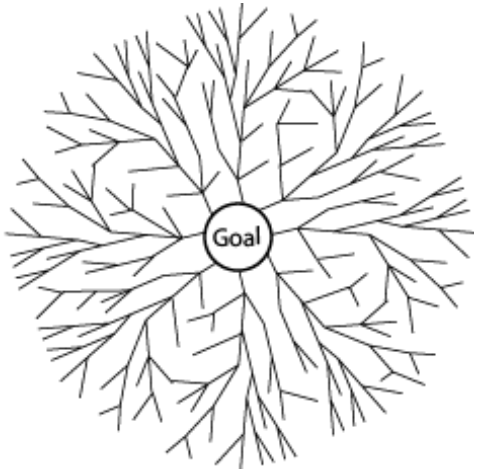
Properties of iterative deepening search

- Complete? Yes
- Time? $d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, ถ้า Step Cost = 1

- IDS uses linear space and not much more time than other uninformed search

Bidirectional Search

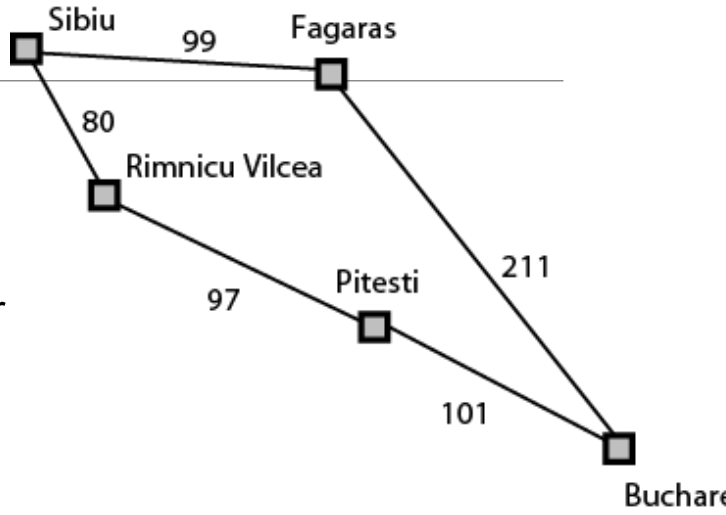




Two-way Search

- Forward search from initial state
- Backward search from goal state
- Number of created nodes $b^{\frac{d}{2}} + b^{\frac{d}{2}} \ll b^d$
- When both ways find the same state $\rightarrow$ Solution
  - But might not be optimal solution
- Some questions
  - Can backward search be done?
  - What if there are more than one goal states?

Uniform-cost Search



- When actions have different cost
- Expand Node with least path cost
- Implementation: priority queue by path cost as frontier
- = BFS if step costs are all equal
- Complete?? Yes, if step cost $\geq \mathcal{E} > 0$
- Time?? $O\left(b^{\lceil C^*/\varepsilon \rceil}\right)$ Expand only nodes with path cost less than optimal solution, when C^* is path cost of optimal solution
  - $b^{\lceil C^*/\varepsilon \rceil}$ might be much higher than b^d
- Space?? $O\left(b^{\lceil C^*/\varepsilon \rceil}\right)$ Keep nodes with path cost less than optimal solution,
- Optimal?? Yes—nodes Expanded In Increasing Order Of Cost, G(n)

Review: Uniform-cost Search

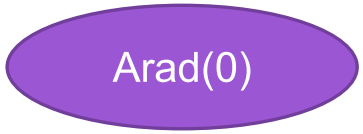
```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ⟵ a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ⟵ a priority queue ordered by PATH-COST, with node as the only element
    explored ⟵ an empty set
    loop do
        if EMPTY?(frontier) then return 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
```

Report solution when goal state is expanded **not when being put in the frontier**

Example of Uniform-cost Search

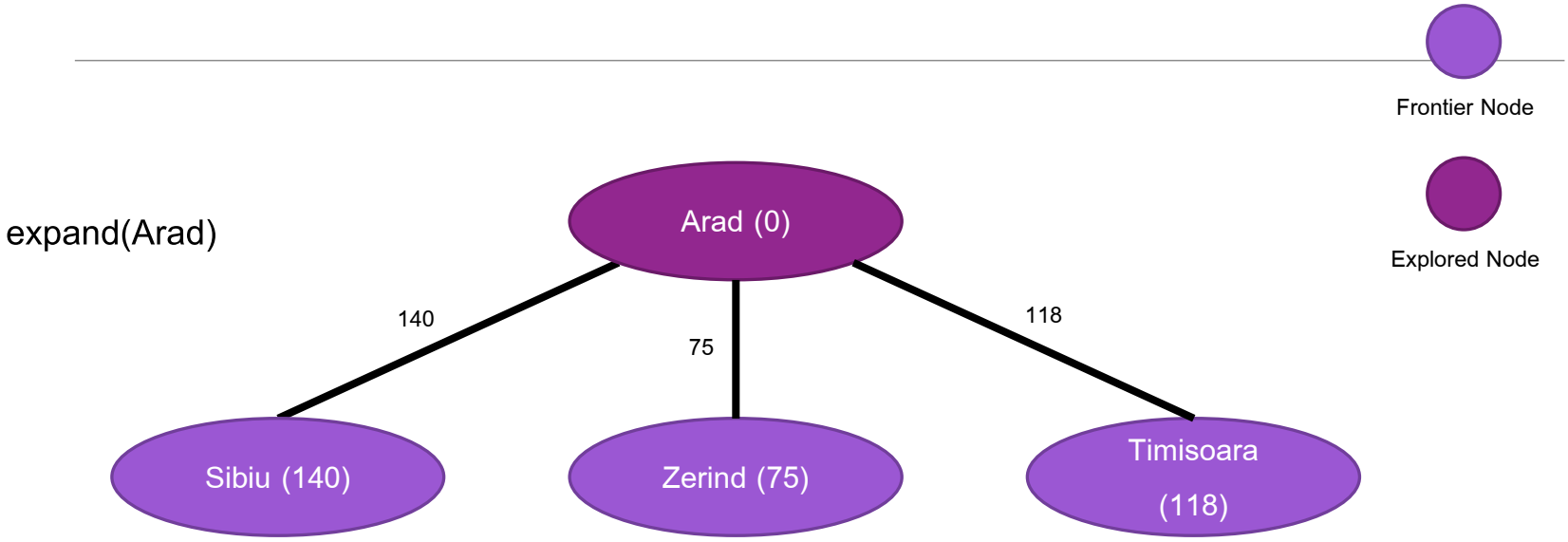
Frontier Node

Explored Node



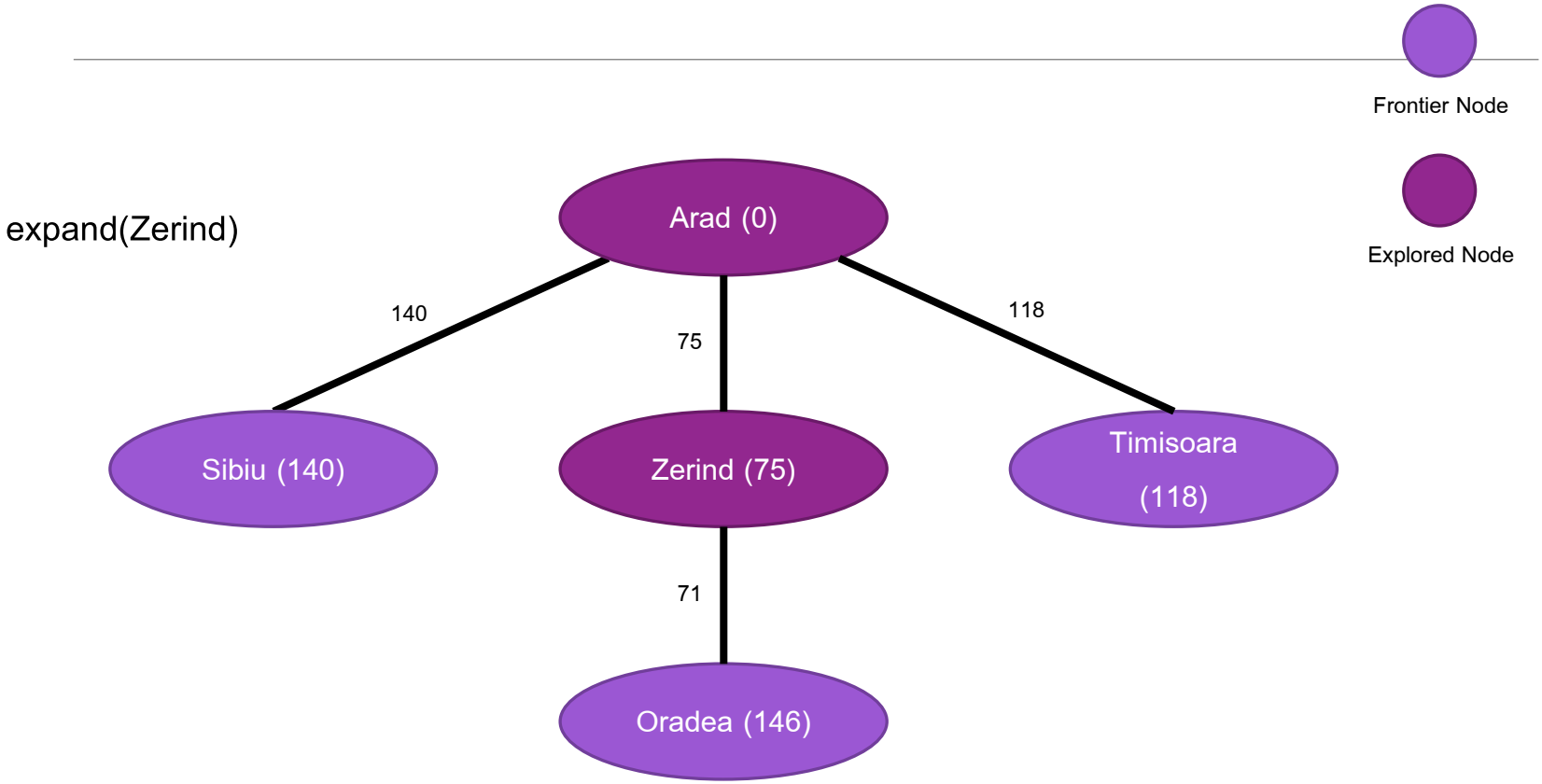
frontier = {Arad(0)}

Example of Uniform-cost Search



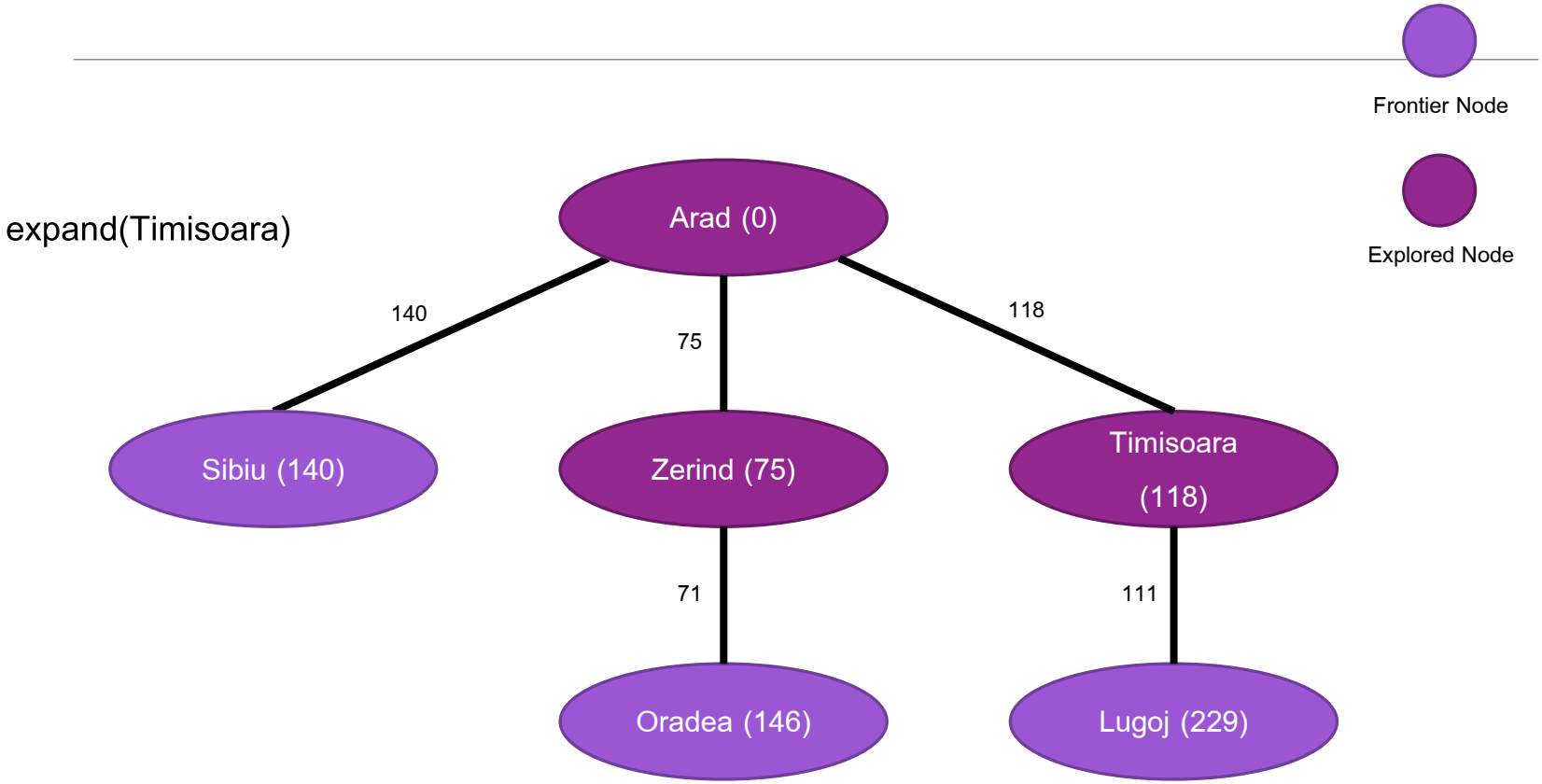
frontier = {Zerind(75), Timisoara(118), Sibiu(140)}

Example of Uniform-cost Search



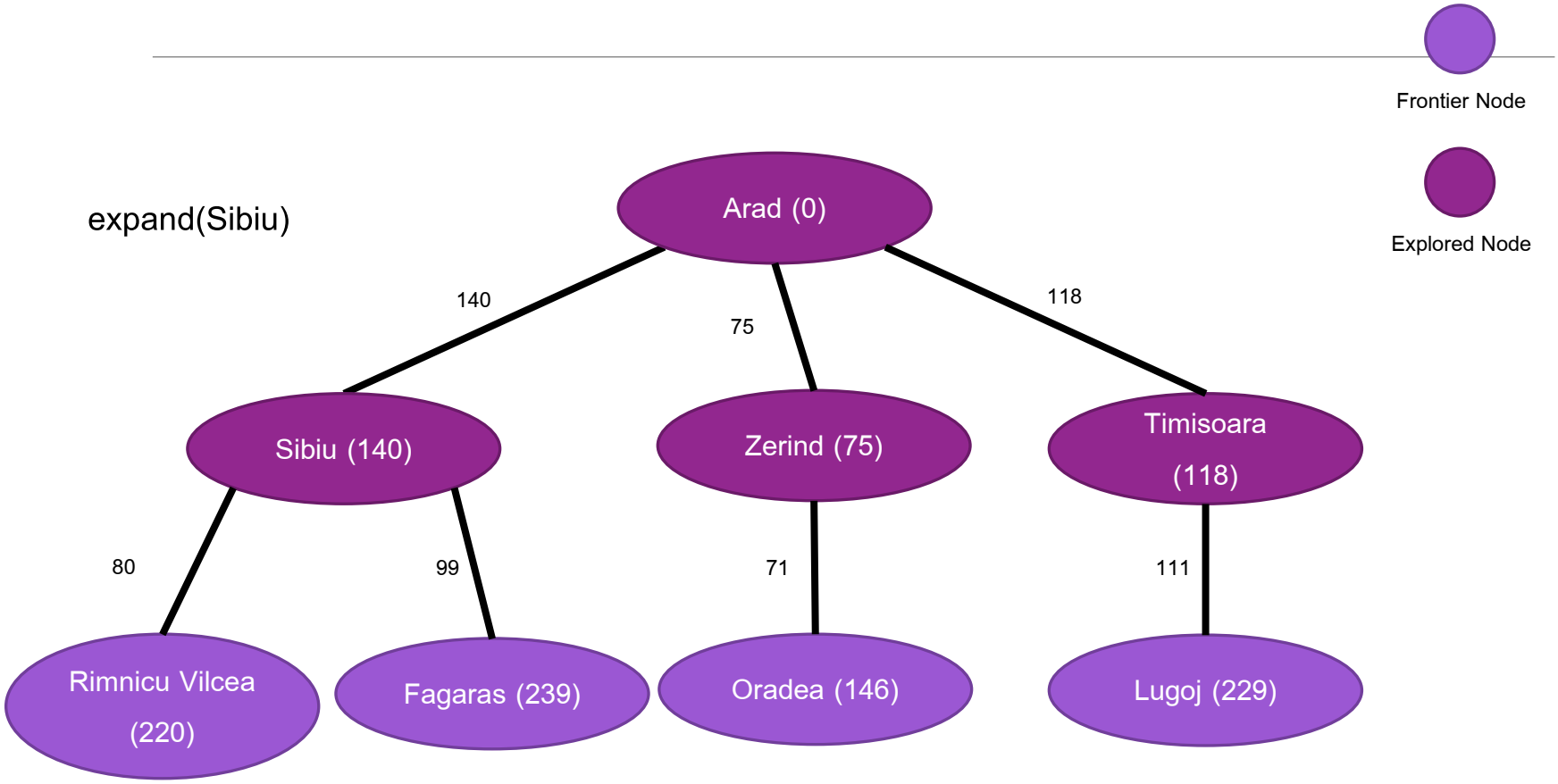
frontier = {Timisoara(118), Sibiu(140), Oradea(146)}

Example of Uniform-cost Search



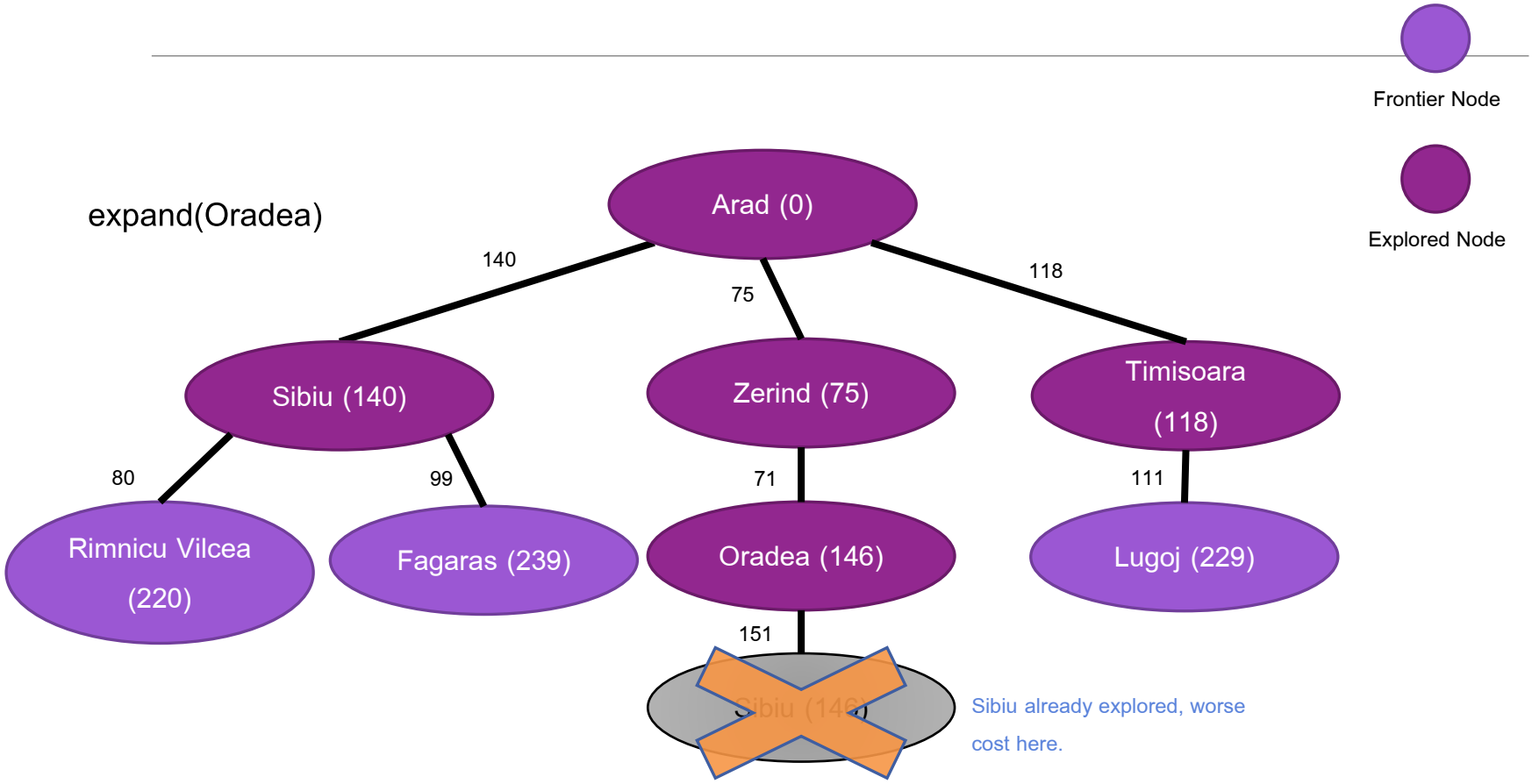
frontier = {Sibiu(140), Oradea(146), Lugoj (229)}

Example of Uniform-cost Search



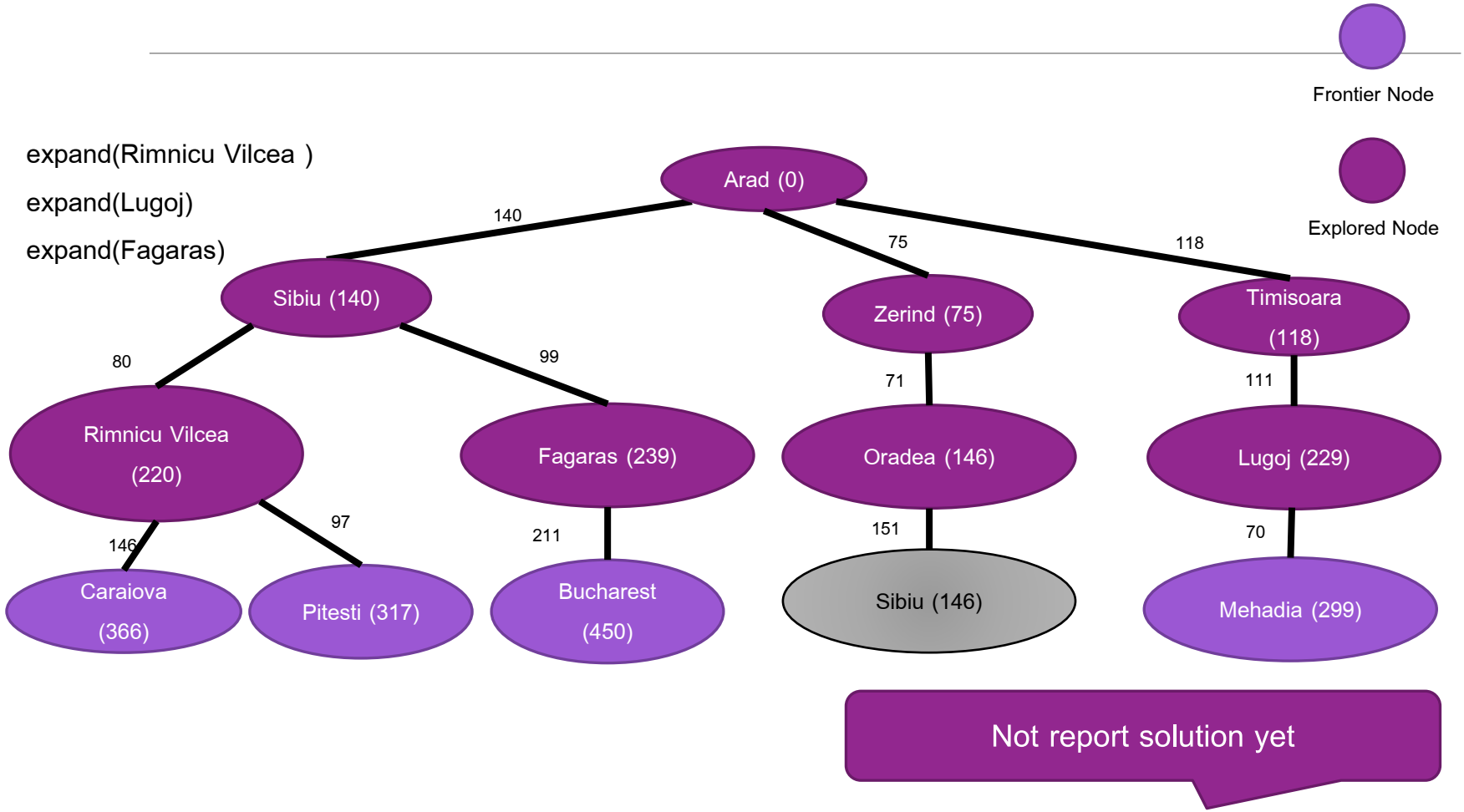
frontier = {Oradea(146), Rimnicu Vilcea (220), Lugoj (229), Fagaras (239)}

Example of Uniform-cost Search



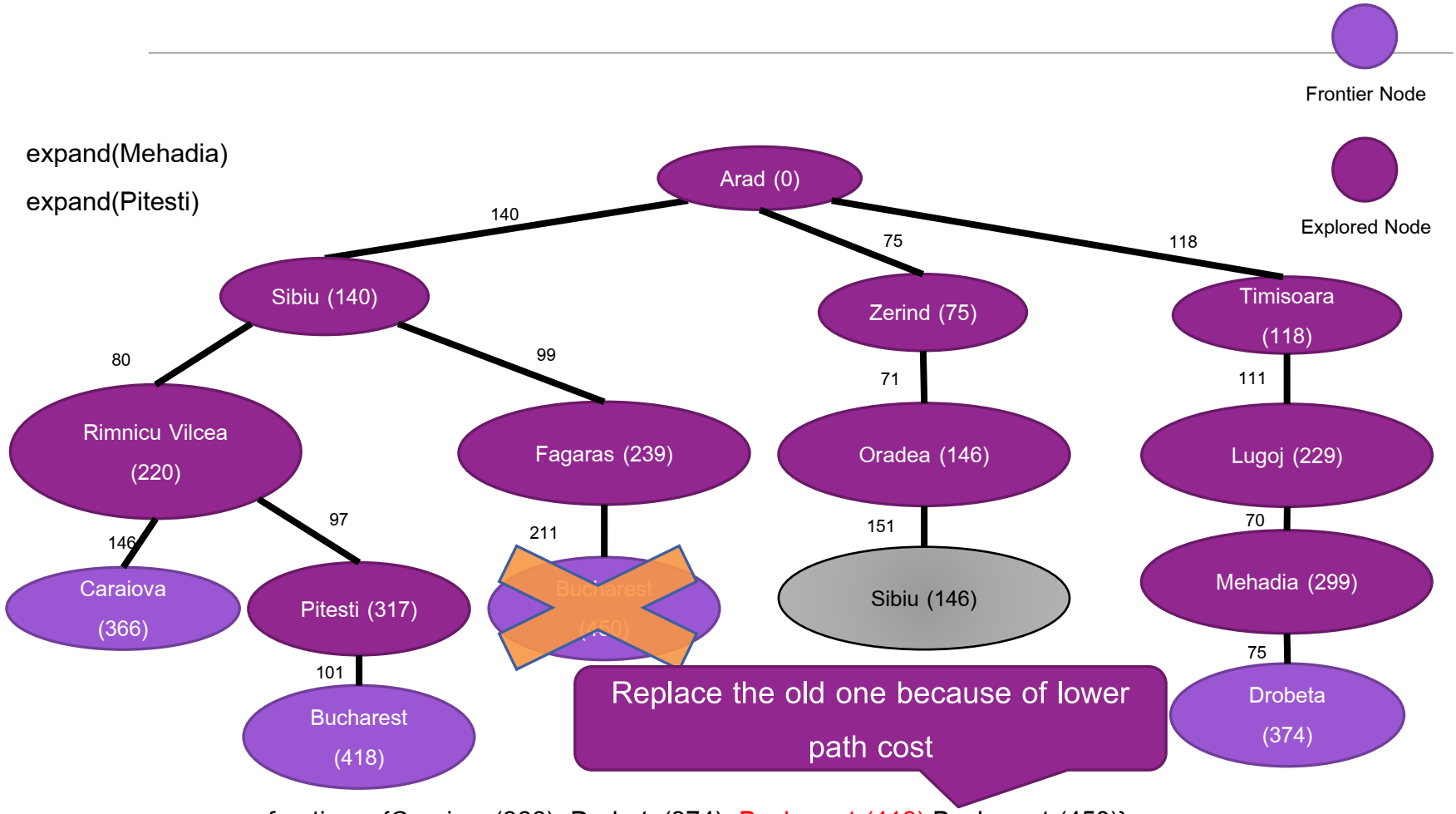
frontier = {Rimnicu Vilcea (220), Lugoj (229), Fagaras (239)}

Example of Uniform-cost Search



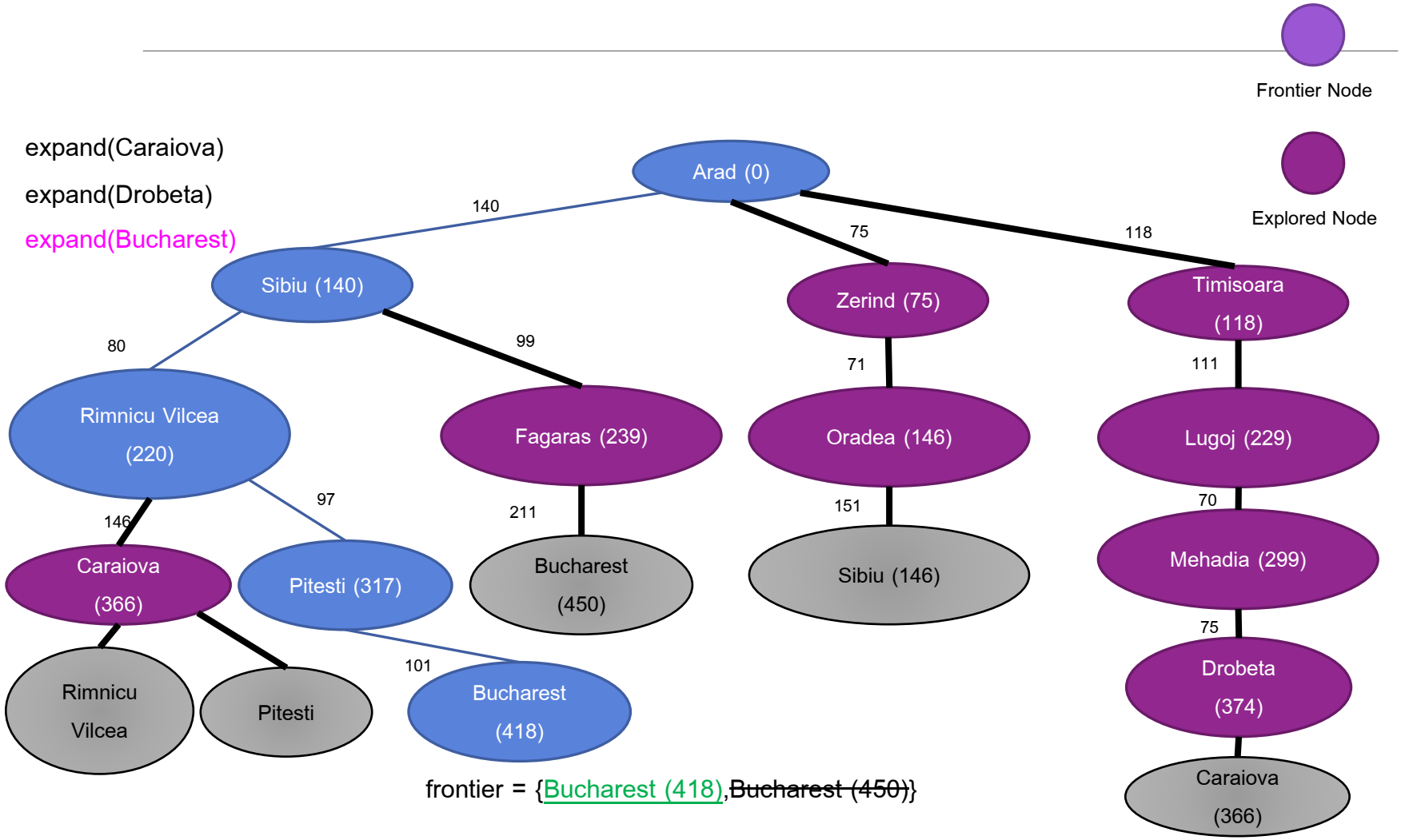
frontier = {Mehadia(299), Pitesti(317), Caraiova(366), Bucharest (450)}

Example of Uniform-cost Search



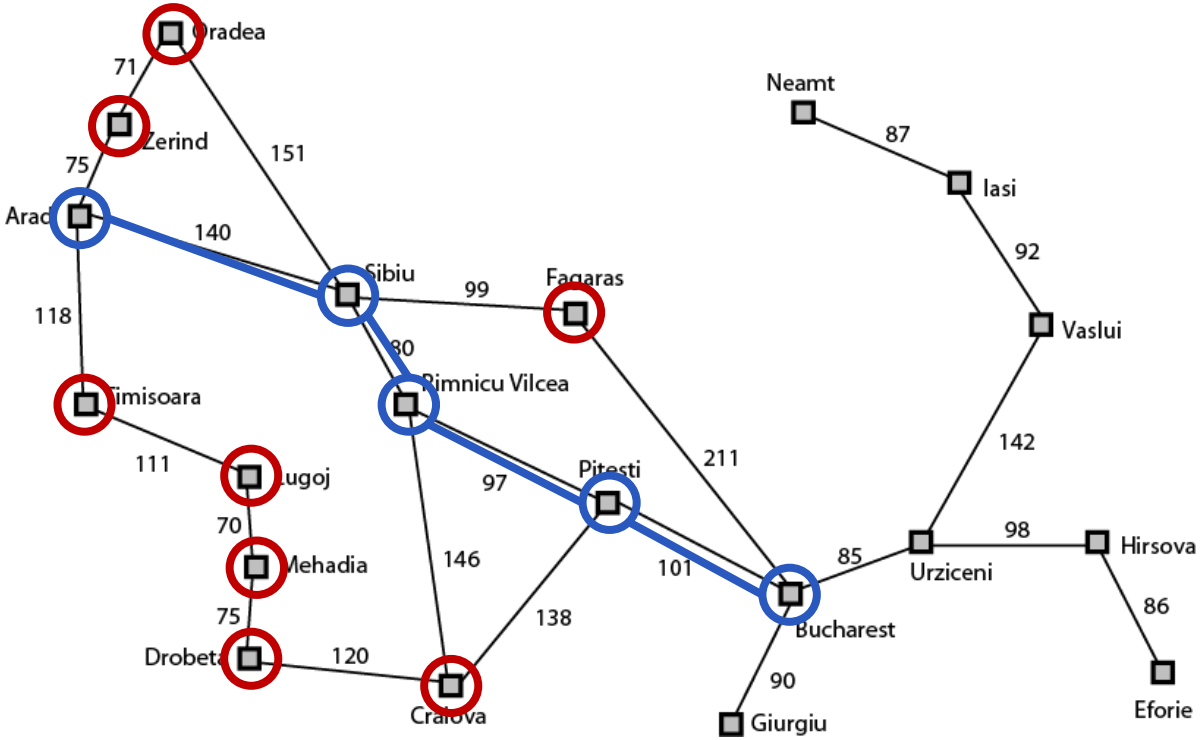
frontier = {Caraiova(366), Drobeta(374), Bucharest (418),~~Bucharest (450)~~}

Uniform-cost Search: ตัวอย่างการทำงาน



Uniform-cost Search

Expanded Nodes



Uniform-cost Search: Main Ideas

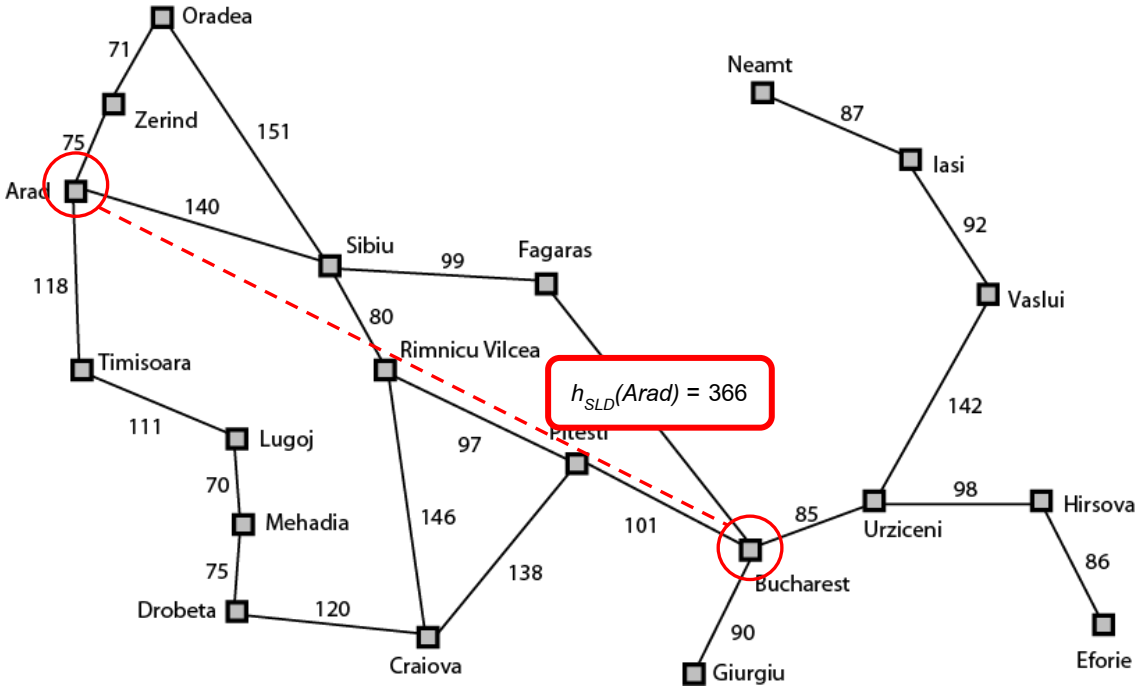
- If nodes with the same state appear in frontier set, pick the one with **least path cost**
- Won't report solution until node with goal state is chosen to be expanded
- Path Cost will only consider existing (past) nodes **and nothing in the future**

Searching
– Informed Search

Informed Search

- Use information beyond problem definition to pick a node in the frontier set
- To approximate the path cost from current node to goal state, *heuristic function*, $h(n)$ is used
  - Depend **only** on the state of the node
- Best-first search
  - Use Evaluation Function, $f(n)$ as cost estimate, which may contain both past path cost $g(n)$ and heuristic $h(n)$
  - Expand node in frontier by $f(n)$, how "interesting" a node is
  - Example: greedy best-first search, A* search

Heuristic Example: Romania



$h_{SLD}(n)$

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

- $h_{SLD}(n)$ *is* straight-line distance from any city to Bucharest

Greedy Best-first Search

- Evaluation Function $f(n) = h(n)$
  - Only use heuristic
  - For example, $h_{SLD}(n)$ straight-line distance from any city to Bucharest
- Greedy best-first search will expand node that seems to be nearest to goal

Example of Greedy Best-first Search

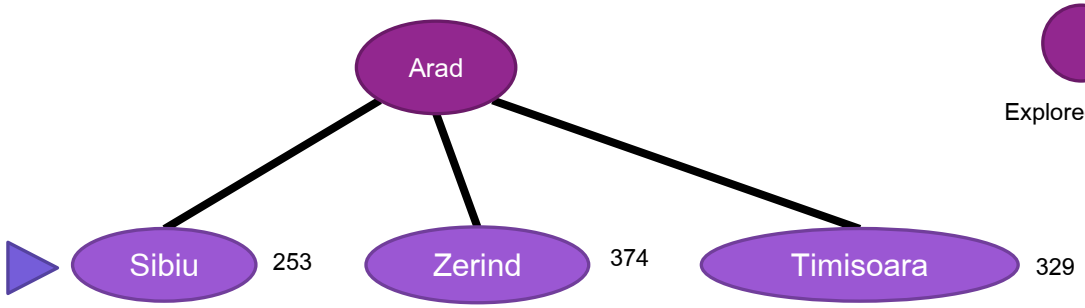
Frontier Node

Explored Node

Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example of Greedy Best-first Search

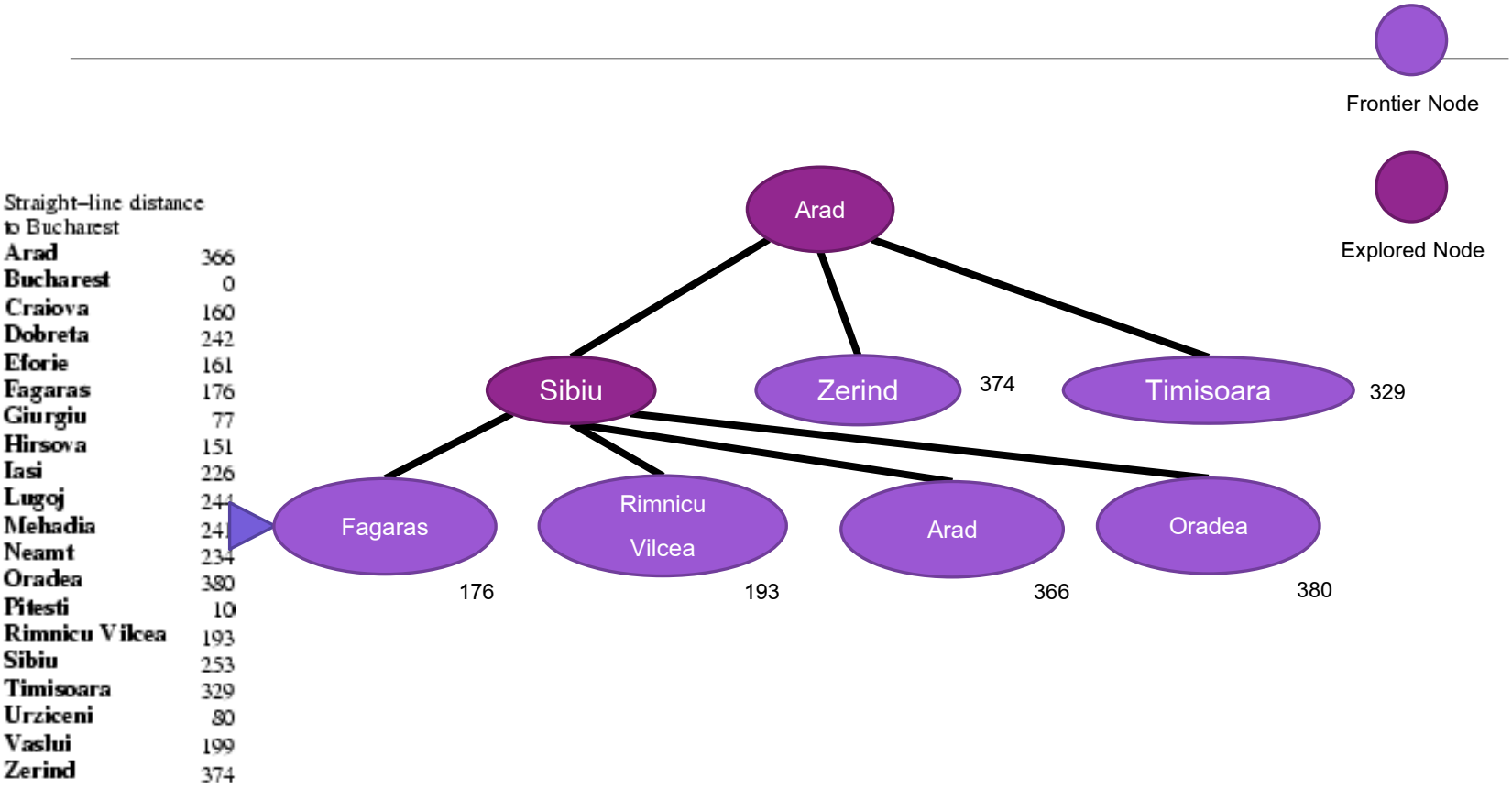
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Frontier Node

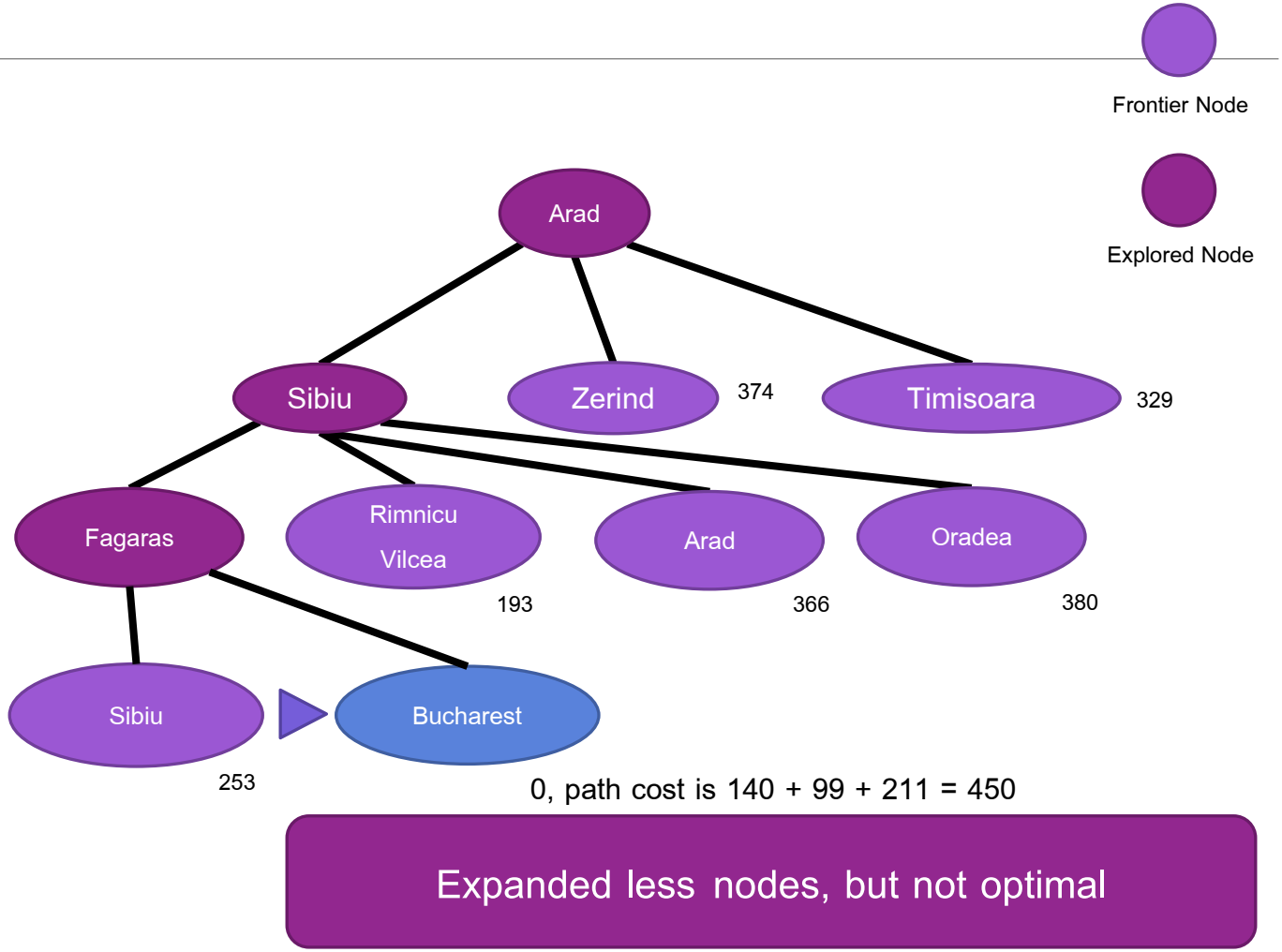
Explored Node

Example of Greedy Best-first Search



Example of Greedy Best-first Search

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Properties of Greedy Best-first Search

- Complete?
  - No – can be infinite loop for tree search
    - If you want to go from Iasi to Fagaras: Iasi → Neamt → Iasi → Neamt → ∞
- Time?
  - $O(b^m)$, but a good heuristic can reduce number of nodes to be create by a lot
- Space?
  - $O(b^m)$ – Like BFS, need to keep most nodes in memory
- Optimal?
  - No

A* ("A-star") Search

- Avoid expanding path with high (approximated) cost
  - Using both past path cost and heuristic
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ is path cost from initial state to node n (known)
  - $h(n)$ is estimated cost from n to goal
  - $f(n)$ is estimated cost from initial state, passing n, to goal
- A* need to use *admissible* heuristic
  - $h(n) \leq h^*(n)$ where $h^*(n)$ is **actual** path cost from n to goal
  - Never overestimate

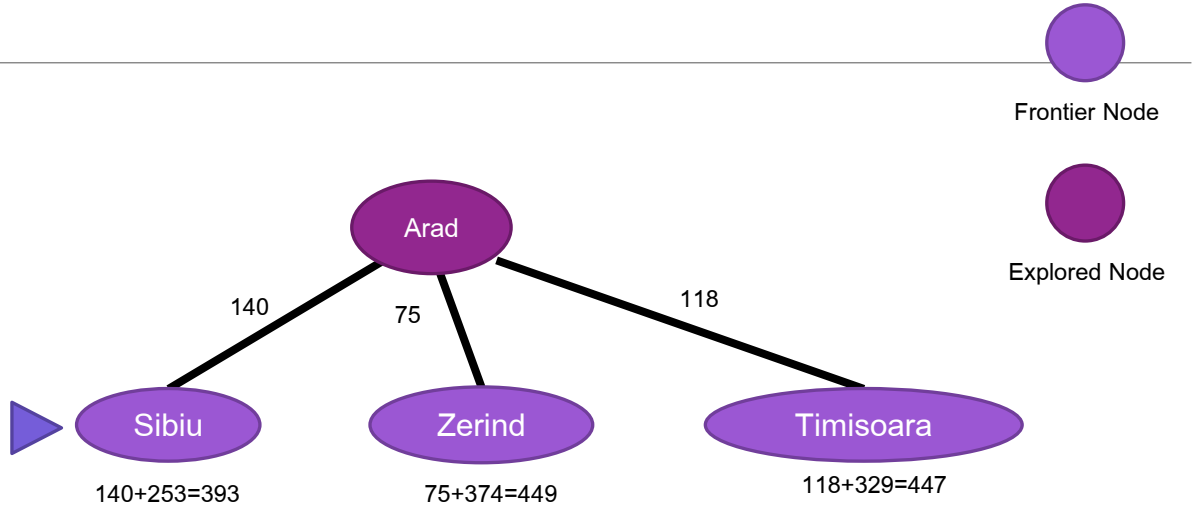
A* search example

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

0 + 366 = 366

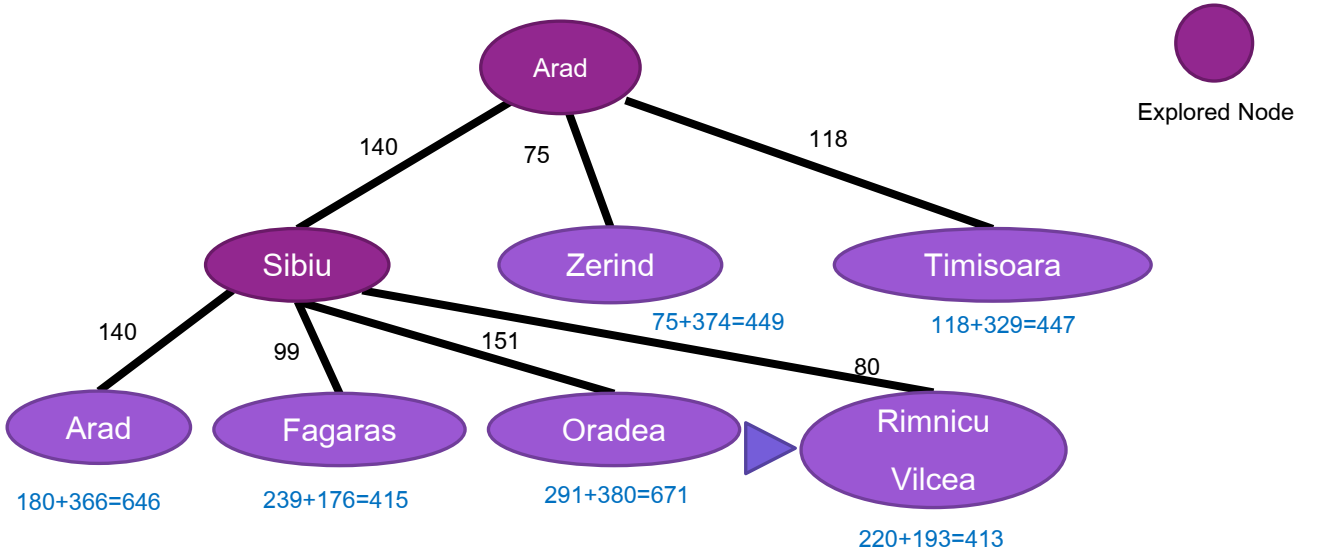
A* search Example

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



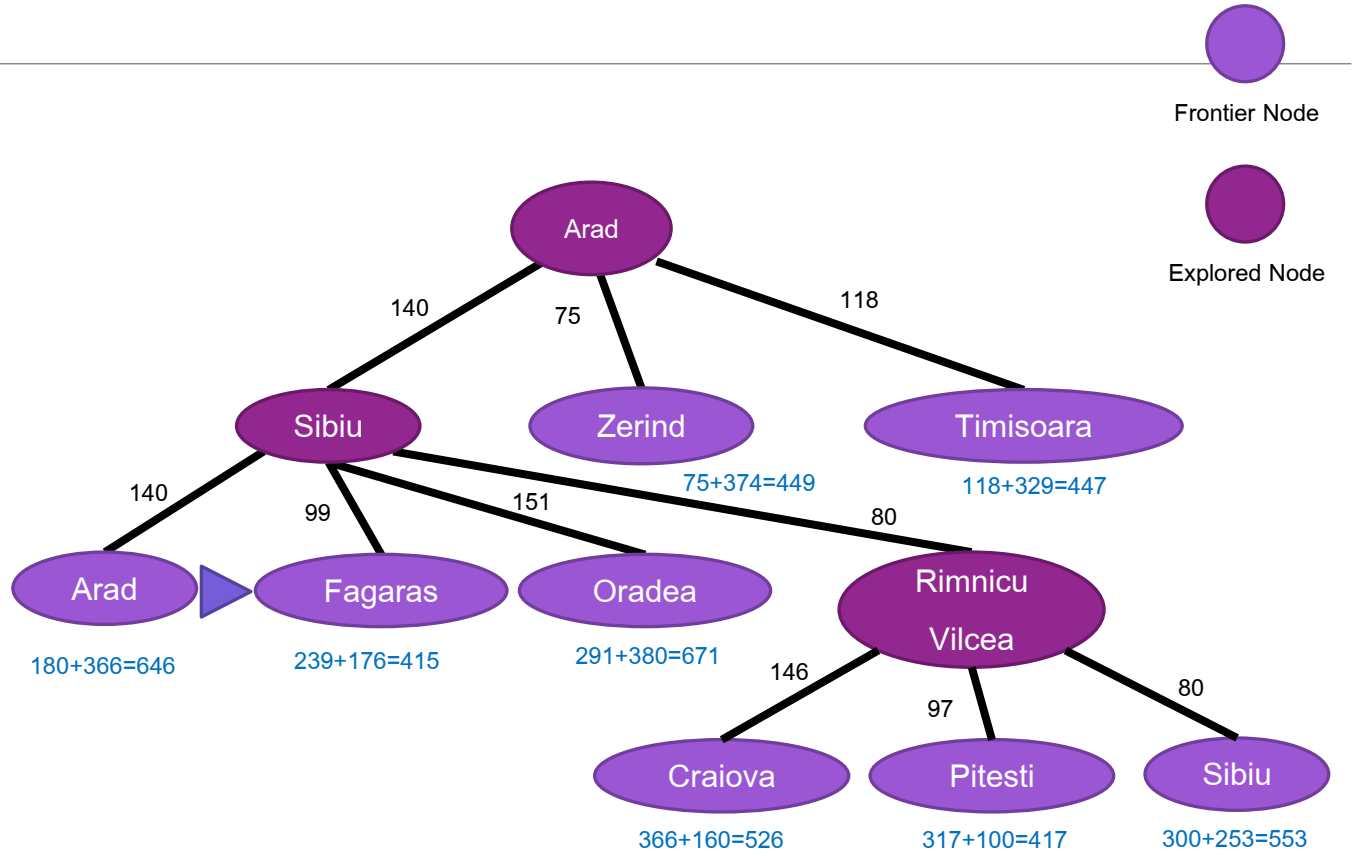
A* search Example

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



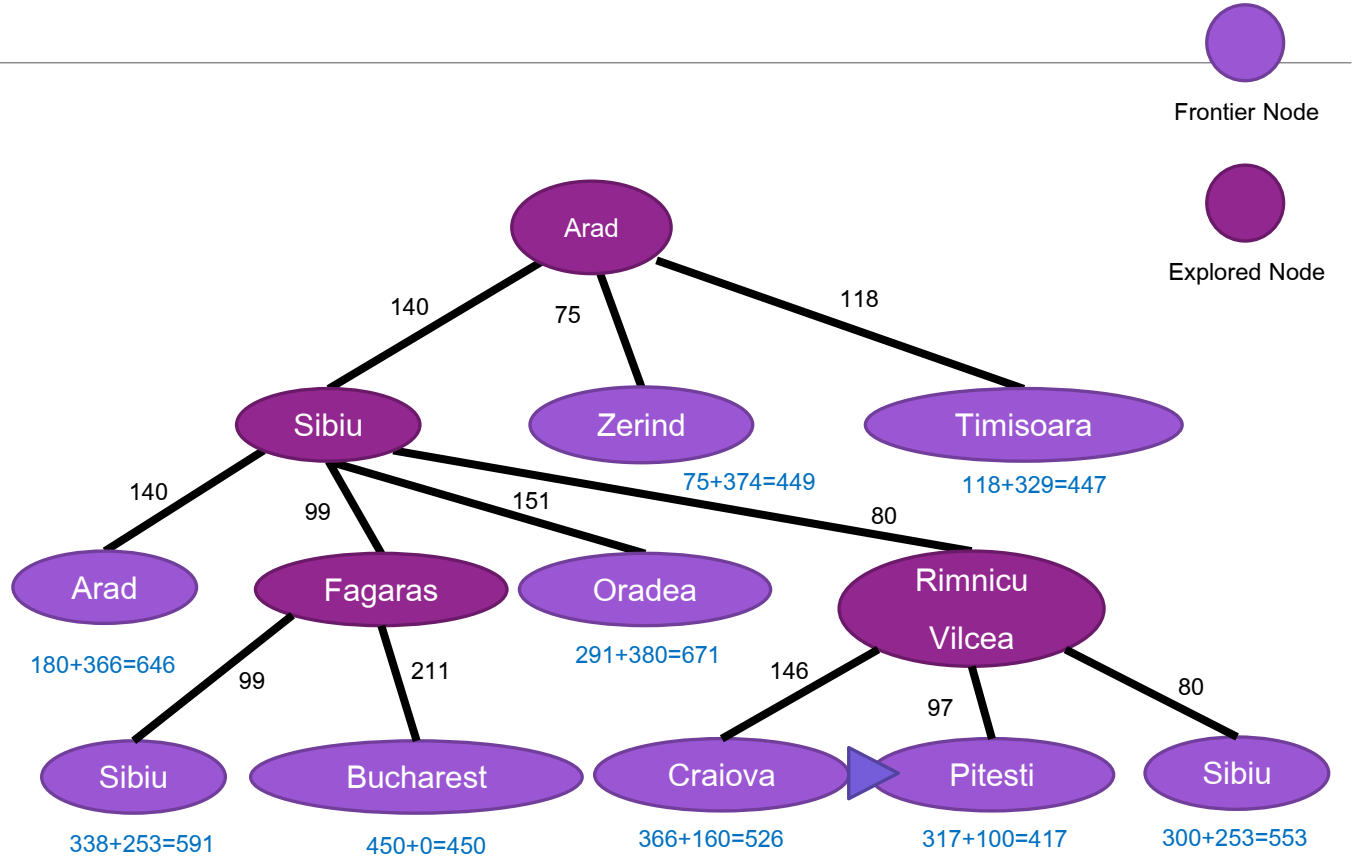
A* search Example

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



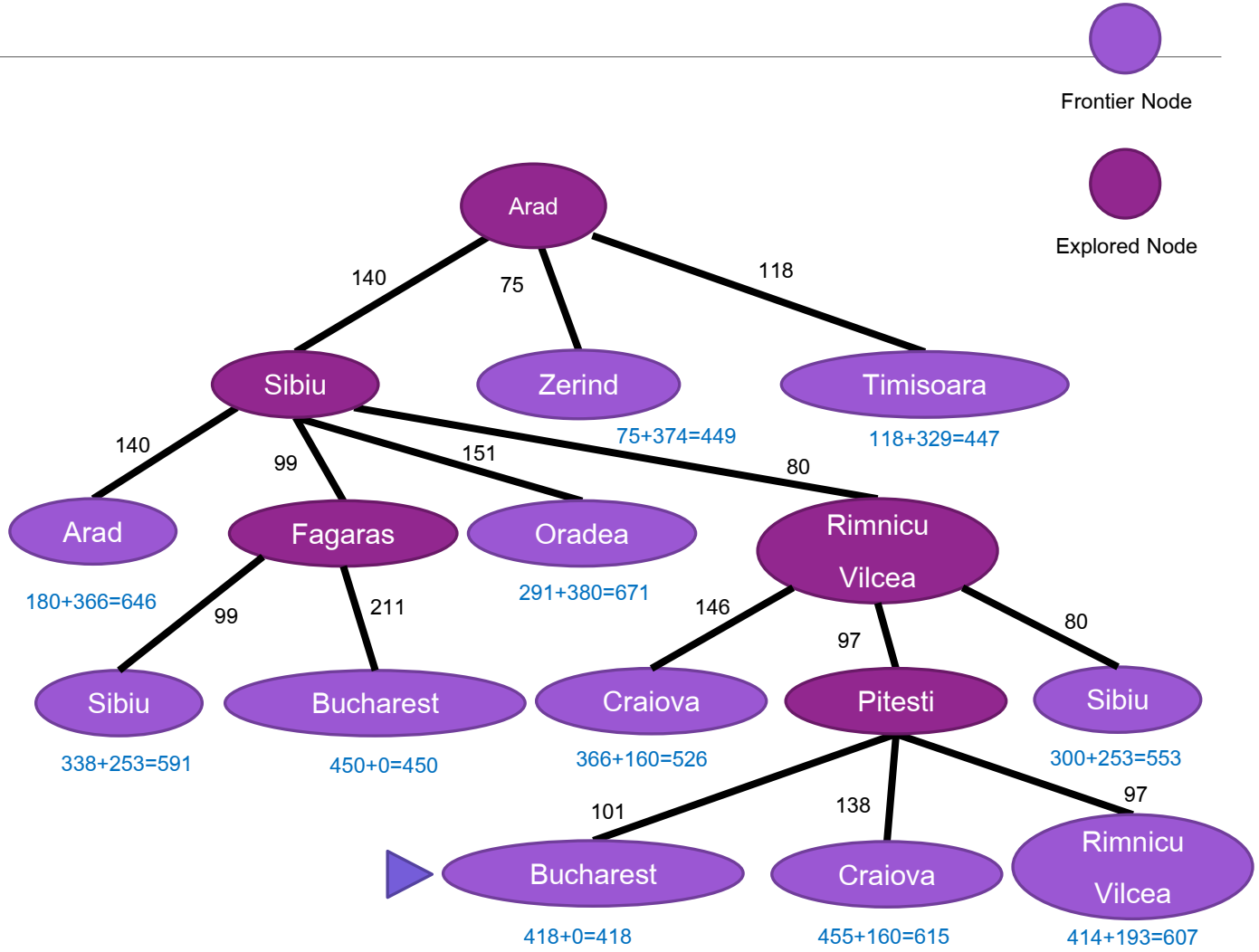
A* search Example

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



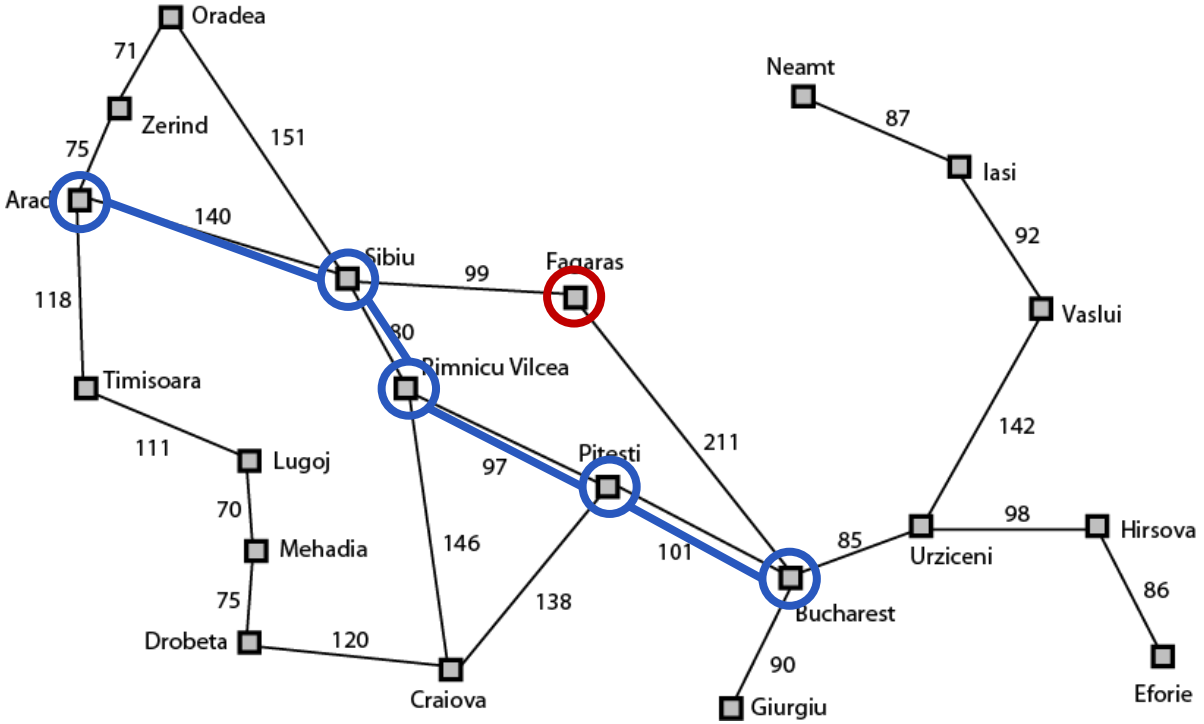
A* search Example

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

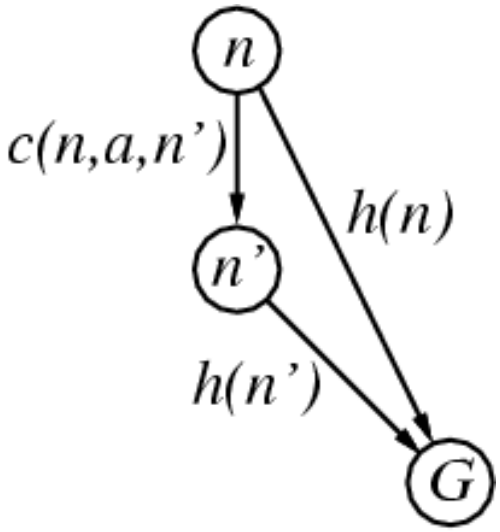


A* search Search

Expanded Nodes



Optimality of A* Search



- admissible *h(n)* will be *consistent* when:
  - Let *n'* be successor of *n*
  - *a* is the Action that can cause state change from *n* to *n'*
  - $\forall n'$: $h(n) \leq c(n, a, n') + h(n')$

- A* graph-search will be optimal search if uses consistent *h*(*n*)
- A* tree-search will be optimal search if uses admissible *h*(*n*)

Proof

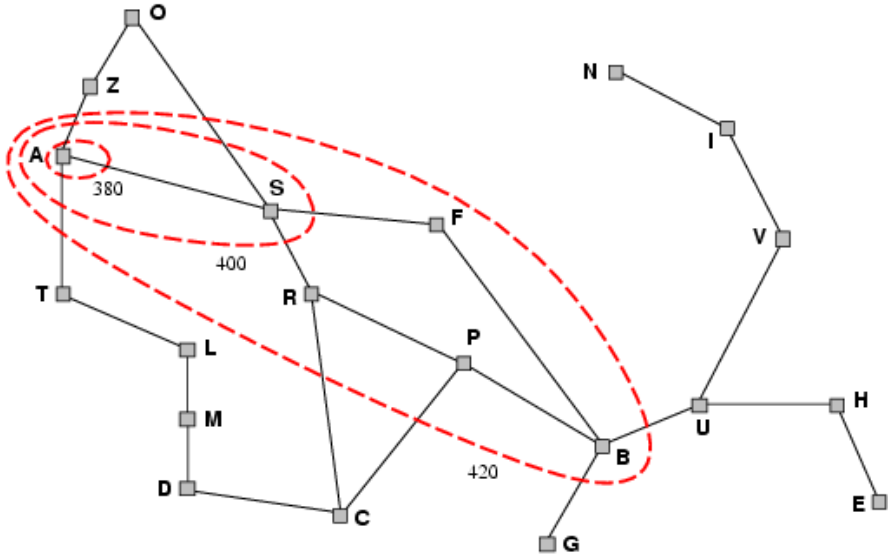
1. From consistent *h(n)* we'll get *f(n)* that is non-decreasing in the path

$$
\begin{aligned}
f(n') &= g(n') + h(n') \\
&= g(n) + c(n,a,n') + h(n') \\
&\geq g(n) + h(n) \\
&= f(n)
\end{aligned}
$$

Proof (ต่อ)

2. If node *n* is chosen to be expanded = we got optimal path to *n*

- *f*(*n*) is non-decreasing
- Otherwise, we will find a path to *n* with lower cost already
- Or, we expand in *f-contour*



A* will not expand node with *f* higher than C*

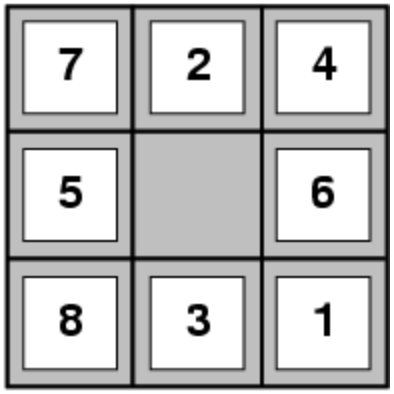
- When C* is Optimal Path Cost

Properties of A* Search

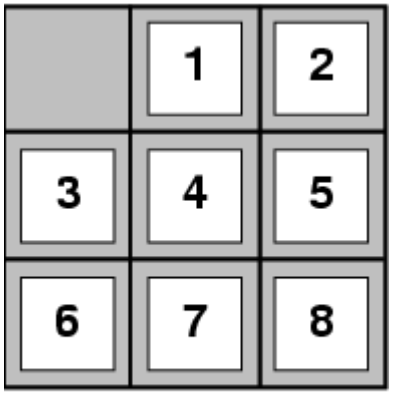
- Complete?
  - Yes. Except if there are unlimited numbers of nodes with f $\leq$ *f(G)*
- Time?
  - Exponential (f-contour can be large)
- Space?
  - Need to keep most nodes in memory
- Optimal?
  - Yes (Depend on *h*(*n*))

How to Create Admissible Heuristic

- Can obtain admissible heuristic by relaxing the restriction of the problem
- Cost of optimal solution for relaxed problem will be admissible heuristic for original problem
- Example: 8-Puzzle
  - Let we move the tiles to any position – number of misplaced tiles $h_1(n)$
  - Let me swap any adjacent tile pair – Manhattan distance $h_2(n)$
  - From the example:
    - $h_1(Start)$ = 8
    - $h_2(Start)$ = 3+1+2+2+2+3+3+2 = 18



Start State



Goal State

How to Choose Heuristic

- **Dominance**: if $\forall n$: $h_2(n) \geq h_1(n)$
  - And both are admissible (or consistent) Heuristic
  - Then h_2 Dominates h_1
  - h_2 is better for searching
- Effective Branching Factor (b^*)
  - In search, we create N nodes, solution depth is d

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

  - We want b^* closer to 1 as possible

Search Cost Comparison

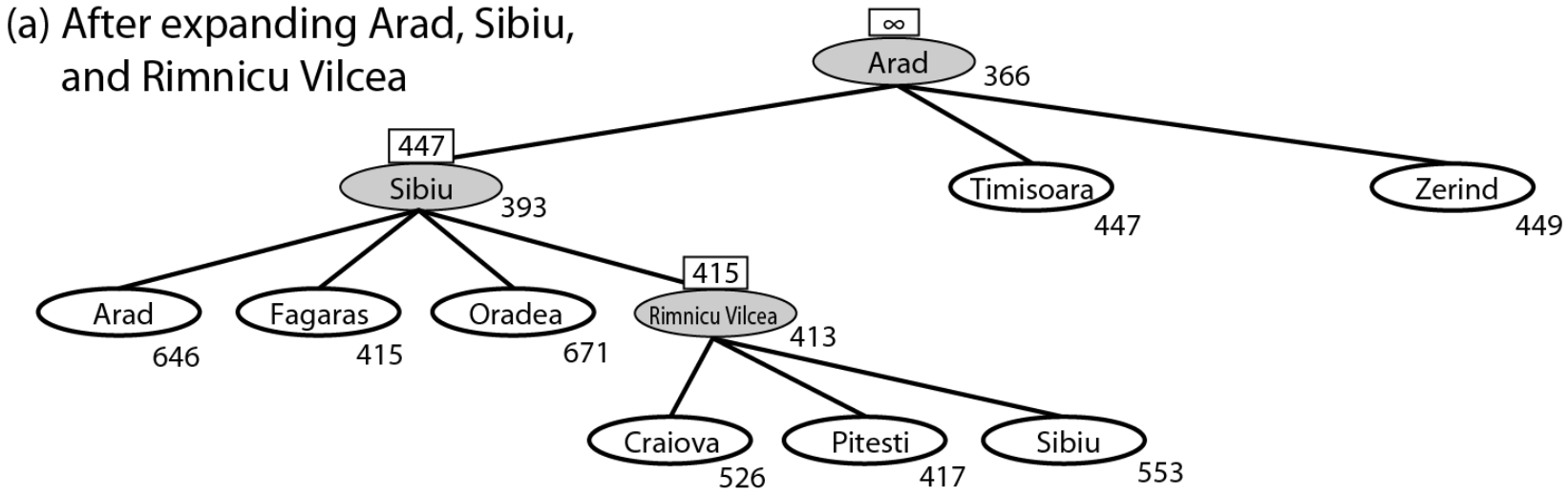
	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

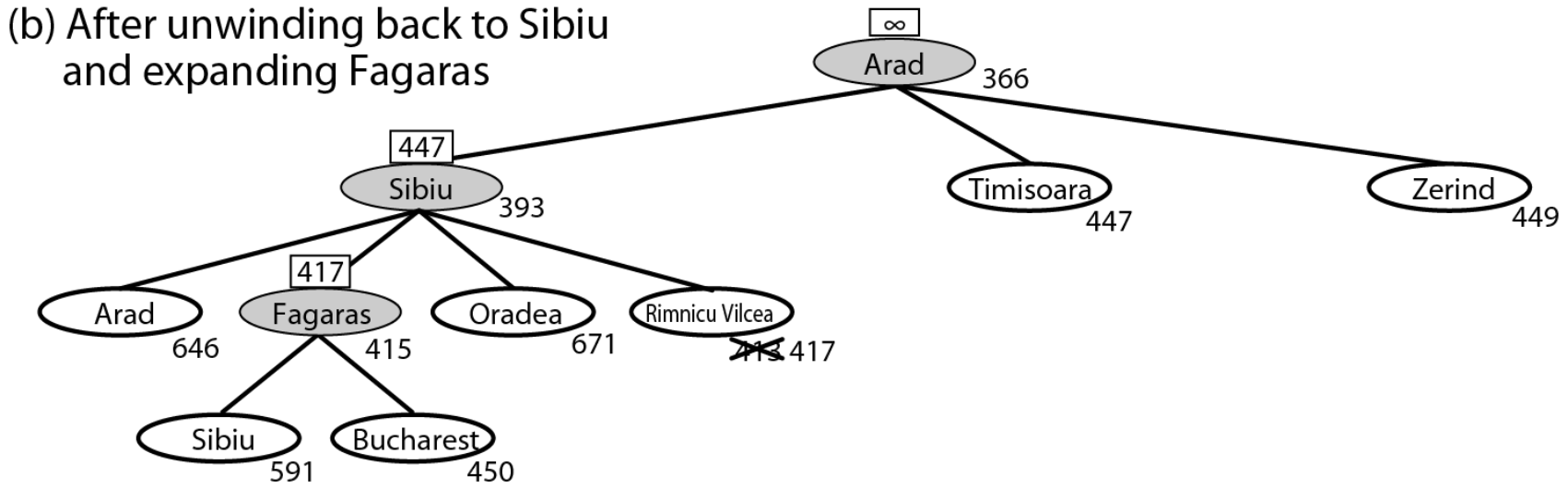
Memory-bounded Heuristic Search

- One main limitation for A* search is memory requirement
- And modified as follow:
  - Iterative-deepening A* (IDA*)
    - IDS that use *f-cost* instead of depth, Expand to the node with smallest cost that exceed limit on the last iteration
  - Recursive Best-first Search (RBFS)
    - Similar to Depth-first Search, but remember best f-cost of the alternatives
    - If f-costs of current child nodes are worse than alternative → remove the children from memory (and transfer best cost to parent) and go expand the alternative
  - Simplified Memory-bounded A* (SMA*)
    - Similar to RBFS but with limit to number of nodes in frontier set. To keep to the limit, worst nodes will be removed, and transfer f-cost to parent node

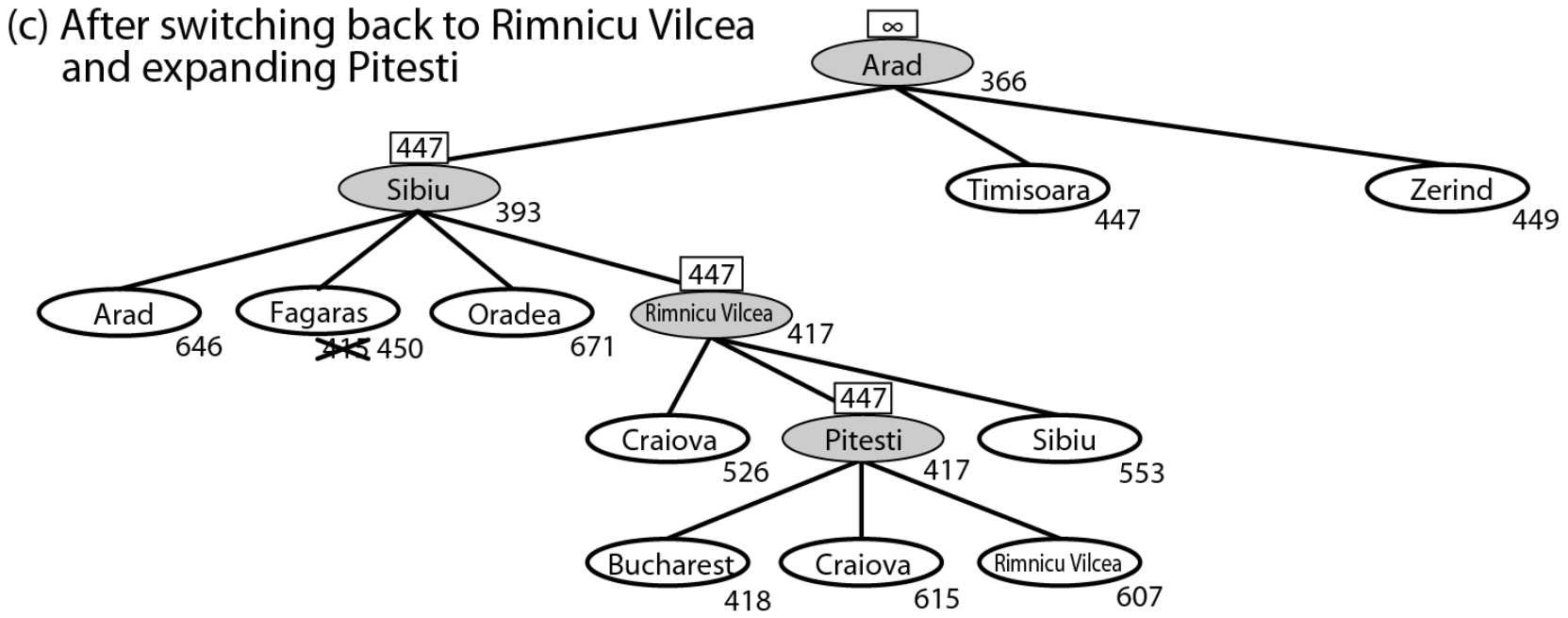
Recursive Best-first Search



Recursive Best-first Search



Recursive Best-first Search



Summary

- In order to perform search, one needs to formulate the problem first
  - States, actions, transitions, goal test, cost
- BFS guaranteed optimality, DFS saves space, IDS meets in the middle
- Admissible heuristic can be used to make A* search more efficient (time-wise) than uninformed search
  - Some modification can make A* more space-efficient
- We want admissible heuristic that is closest to actual cost as possible