# Artificial Intelligence (CS 3011)

## CHAPTER 3: **Solving problems by searching**

**Sourav Kumar Giri**
**Asst. Professor**

**School of Computer Engineering**
KIIT Deemed to be University

## Chapter Outline

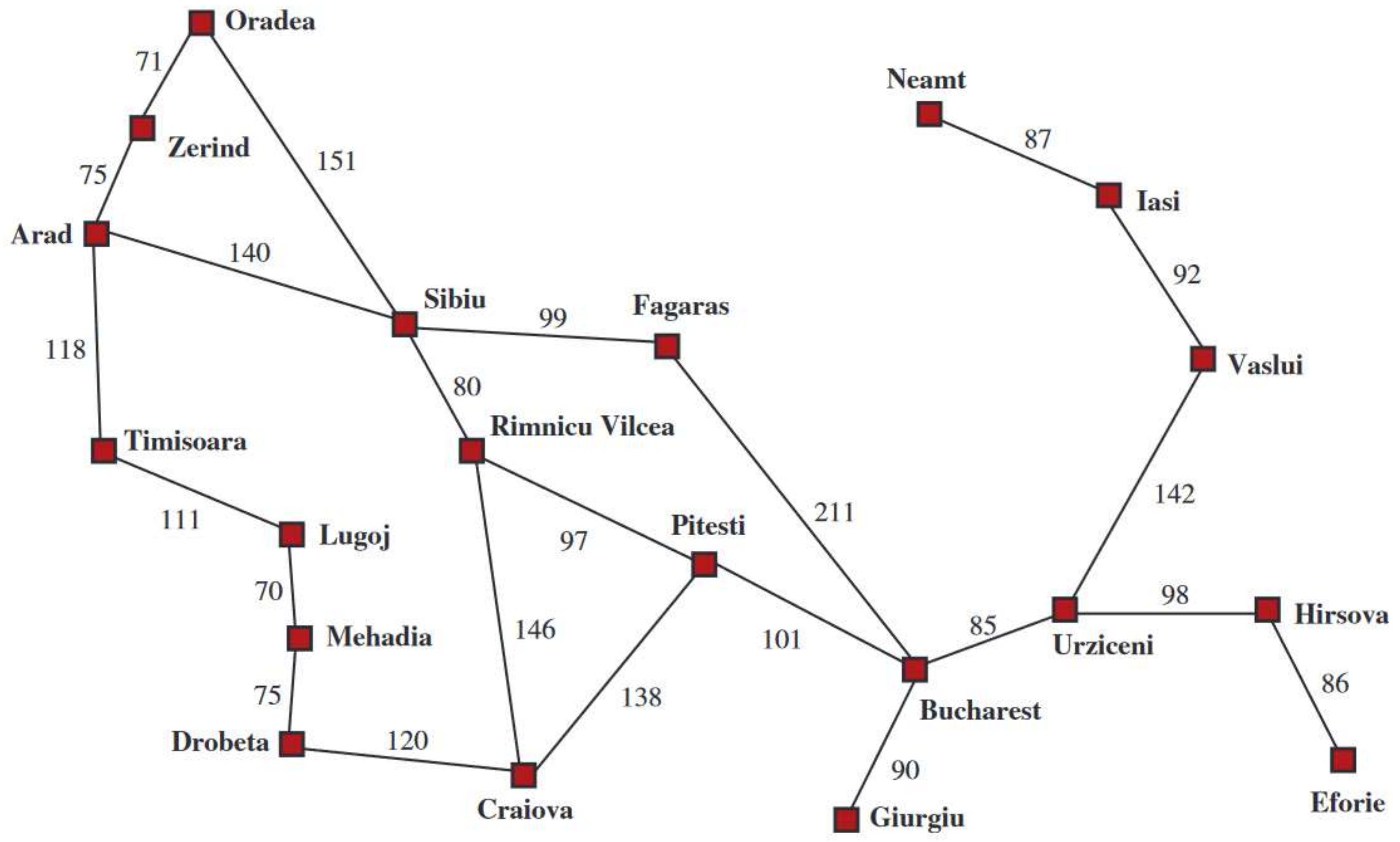
- ❑ Problem Solving Agents
- ❑ Example Problems
- ❑ Searching for Solutions
- ❑ Uninformed Search Strategies
- ❑ Avoiding repeated states
- ❑ Informed Search Strategies
- ❑ Heuristic functions

## Problem solving agent

- The simplest agents are the **reflex agents**, which base their actions on a direct mapping from states to actions. Such agents cannot operate well in environments for which this mapping would be too large to store and would take too long to learn.
- **Goal-based agents**, on the other hand, consider future actions and the desirability of their outcomes.
- Goal-based agents that use atomic representation are called **problem solving agents.**
  - states of the world are considered as wholes, with no internal structure visible to the problem solving algorithms.
- Goal-based agents that use more advanced factored or structured representations are usually called **planning agents**.

## Problem solving agent

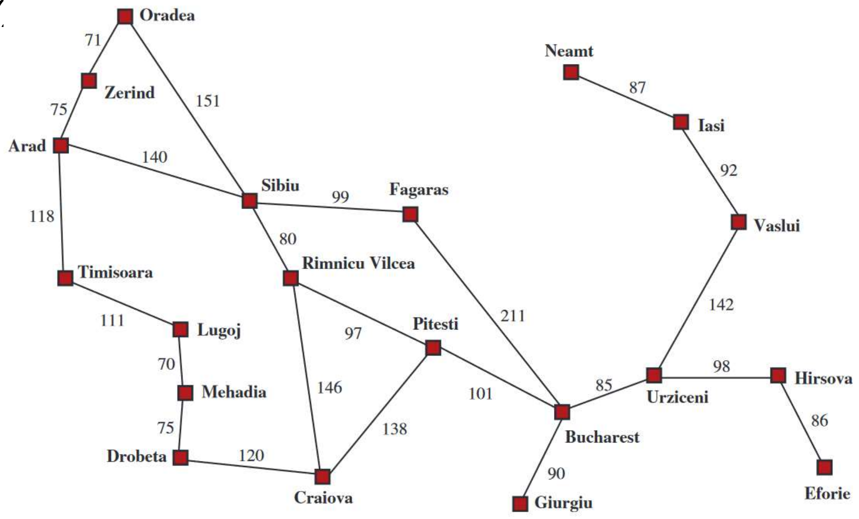
- Phases involved in problem-solving process are-
  - **Goal formulation-** the agent adopts the goal of meeting the objective. Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
  - **Problem formulation-** it is the process of deciding what actions and states to consider, given a goal.
  - **Search-** Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called a **solution**.
  - **Execution-**The agent can now execute the actions in the solution, one at a time.



**A simplified road map of part of Romania, with road distances in miles.**

## Well-defined problems and solutions

- **Initial state-** The **initial state** that the agent starts in.
  - For example, the initial state for our agent in Romania might be described as *In(Arad)*.
- **Action-** A description of the possible actions available to the agent.
  - Given a particular state *s*, ***ACTIONS(s)*** returns the set of actions that can be executed in *s*. We say that each of these actions is applicable in *s*. For example, from the state ***In(Arad)***, the applicable actions are ***{Go(Sibiu), Go(Timisoara ), Go(***



## Well-defined problems and solutions

- **Transition model-** A description of what each model does, specified by a function *RESULT(s,a)* that returns the state that results from doing action *a* in state *s*.
  - We also use the term successor to refer to any state reachable from a given state by a single action. For example, we have

    *RESULT(In(Arad ), Go(Zerind )) = In(Zerind ).*
  - Together, the initial state, actions, and transition model implicitly define the **state space** of the problem—*the set of all states reachable from the initial state by any sequence of actions.*
  - The state space forms a directed network or graph in which the nodes are states and the links between nodes are actions.
  - A **path** in the state space is a sequence of states connected by a sequence of actions.

## Well-defined problems and solutions

- **Goal test-** The goal test, which determines whether a given state is a goal state.
  - Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set *{In(Bucharest )}*
  - Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called "**checkmate**," where the opponent's king is under attack and can't escape.
- **Path cost-** A path cost function that assigns a numeric cost to each path. The problem solving agent chooses a cost function that reflects its own performance measure.
  - For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers. The step cost of taking action *a* in state *s* to reach state s′ is denoted by ***c(s, a, s')***.

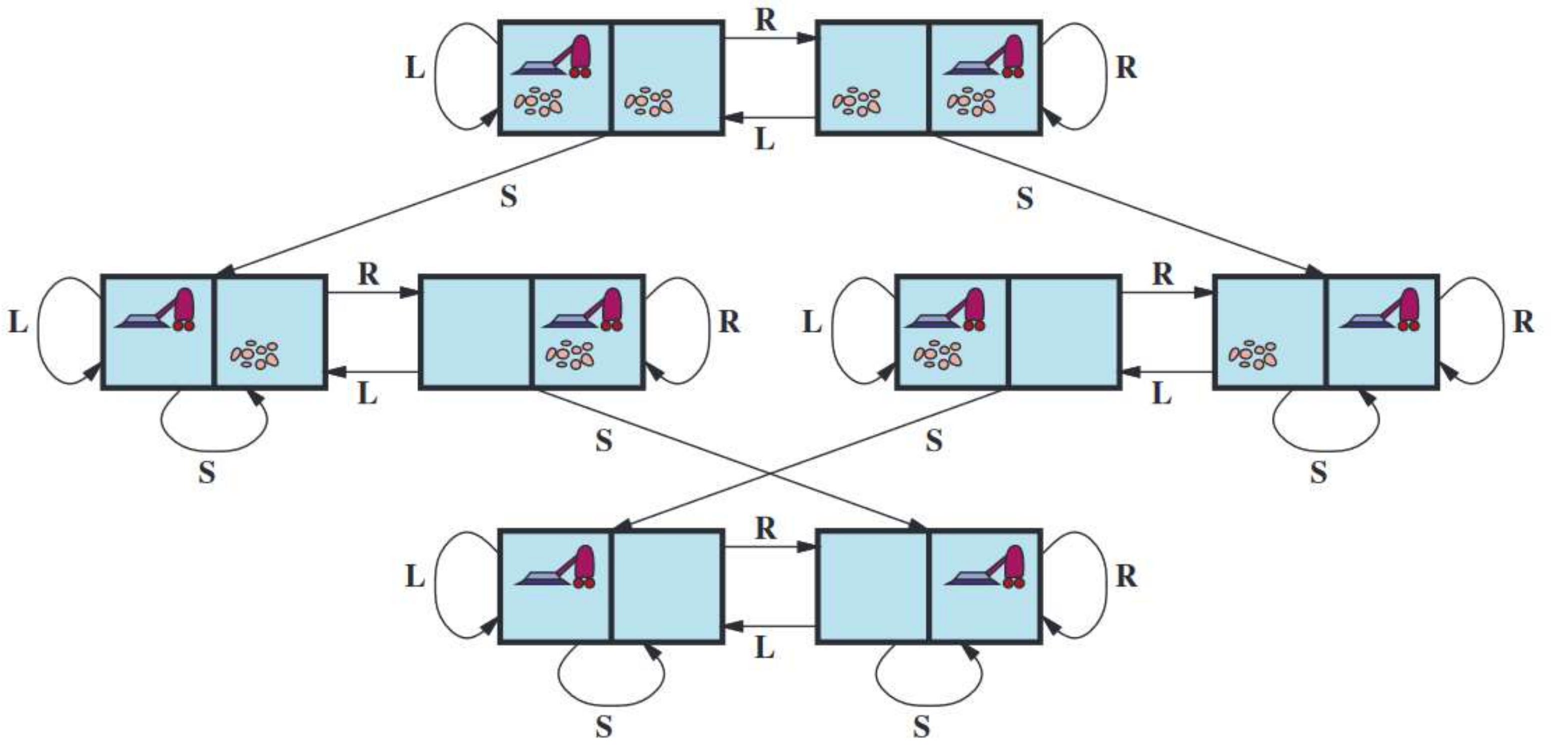
## Well-defined problems and solutions

**A solution to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.**

## Example Toy Problems

### ❑ Vacuum world-

- **States-** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $\mathbf{2 \times 2^2 = 8}$ possible world states. A larger environment with *n* locations has $\boldsymbol{n.2^n}$ states.
- **Initial state-** Any state can be designated as the initial state.
- **Actions-** In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.
- **Transition model-** The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
- **Goal test-** This checks whether all the squares are clean.
- **Path cost-** Each step costs 1, so the path cost is the number of steps in the path.

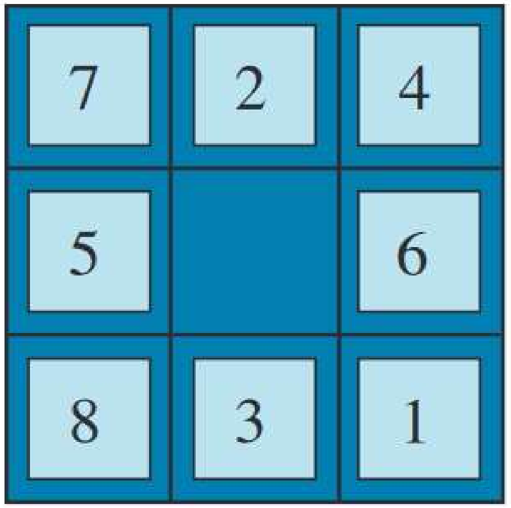


**The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.**

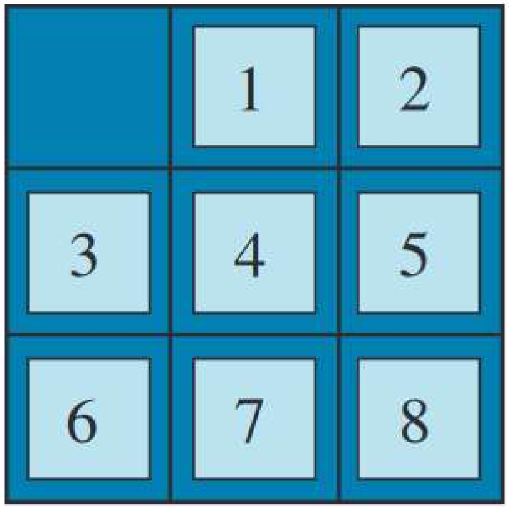
## Example Toy Problems

### ❑ 8 Puzzle

- **States-** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state-** Any state can be designated as the initial state.
- **Actions:** The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.
- **Transition model-** Given a state and action, this returns the resulting state.
- **Goal test:** This checks whether the state matches the goal configuration
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.



Start State



Goal State

**A typical instance of the 8-puzzle**

## Example Toy Problems

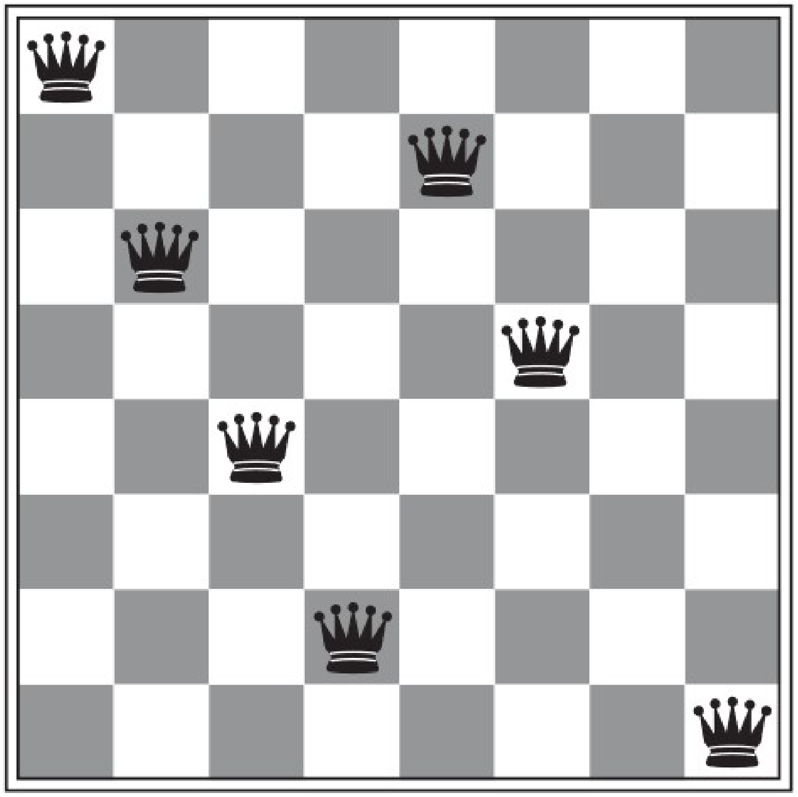
### ❑ 8-Queen Problem

- **States-** Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state-** No queens 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 are on the board, none attacked

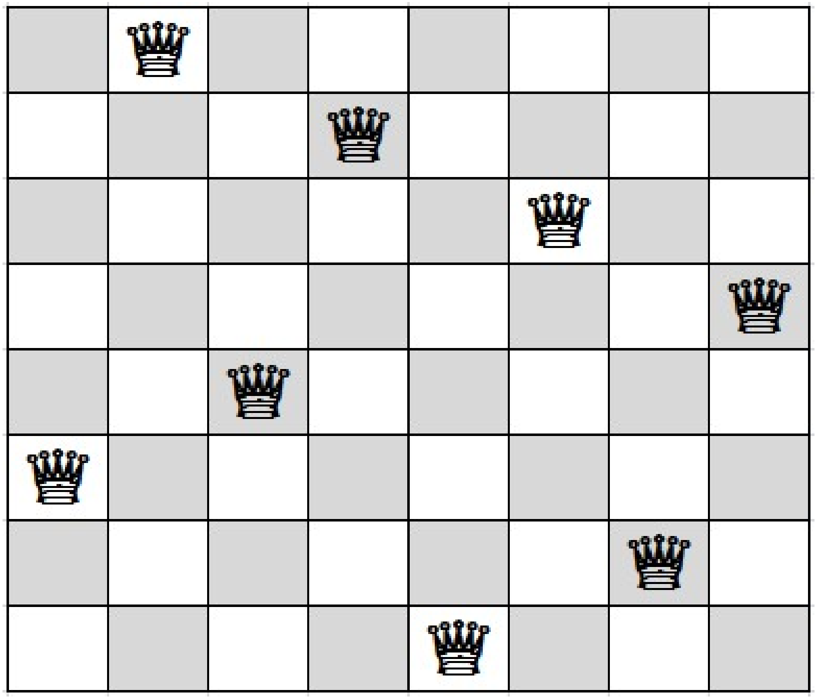
**Note-** In this formulation, we have $\mathbf{64 \cdot 63 \cdots 57 \approx 1.8\times 10^{14}}$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

- **States-** All possible arrangements of n queens ($0 \le n \le 8$), one per column in the leftmost n columns, with no queen attacking another.
- **Actions-** Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queens state space from $\mathbf{1.8\times 10^{14}}$ to just **2,057**, and solutions are easy to find.



**Almost a solution to the 8-queens problem.**



**A solution to the 8-queens problem.**

## Example Toy Problems

### ❑ Toy Problem by Donald Knuth

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = \mathbf{5}$$

- **States-** Positive numbers.
- **Initial state-** 4.
- **Actions-** Apply factorial, square root, or floor operation (factorial for integers only).
- **Transition model:** As given by the mathematical definitions of the operations.
- **Goal test:** State is the desired positive integer.

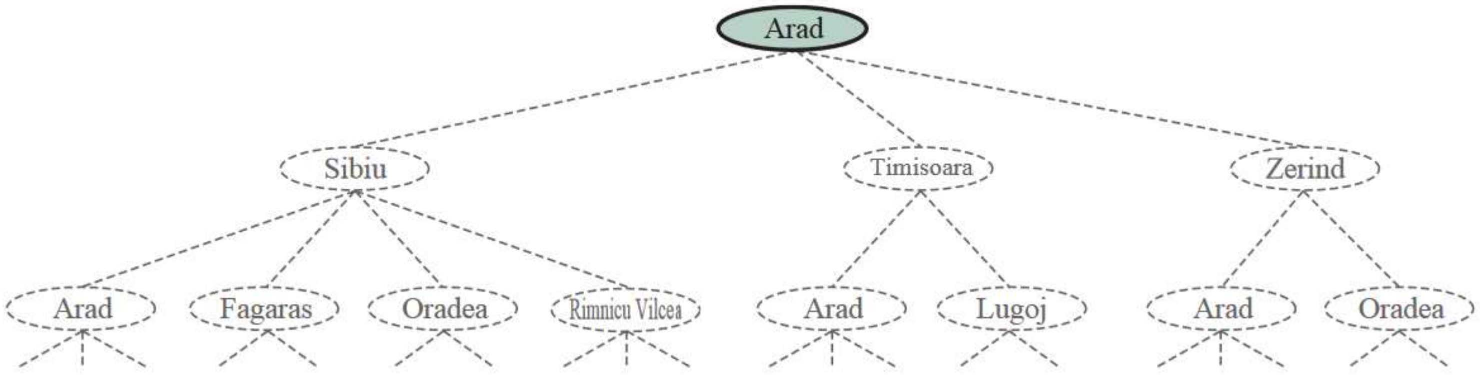
▪ ***Infinite state spaces can arise i.e.*** ***(4!)!=620,448,401,733,239,439,360,000***

## Real world problems

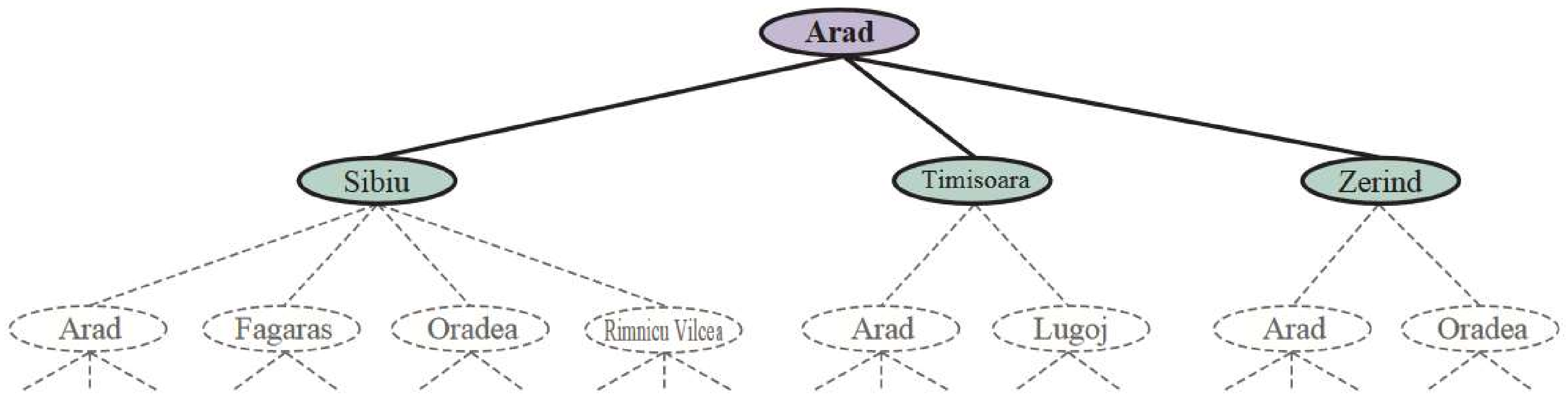
- **Route finding problems-** routing video streams in computer networks, military operations planning, airline travel planning systems etc.
- **Touring problems-** describe a set of locations that must be visited, rather than a single goal destination. Travelling salesman is touring problem where every city on a map must be visited.
- **A VLSI layout-** positioning of millions of components and connections on a chip to minimize area, minimize circuit delays, maximize manufacturing yield etc.
- **Robot Navigation-** generalization of route finding problem. Rather than following distinct paths, a robot can roam around, in effect making its own paths.
- **Automatic assembly sequencing-**Find an order to assemble parts of some object. Example- *protein design:* goal is to find sequence of amino acids that will fold into 3-D protein with the right properties.

## Search Algorithms/ Searching for solution

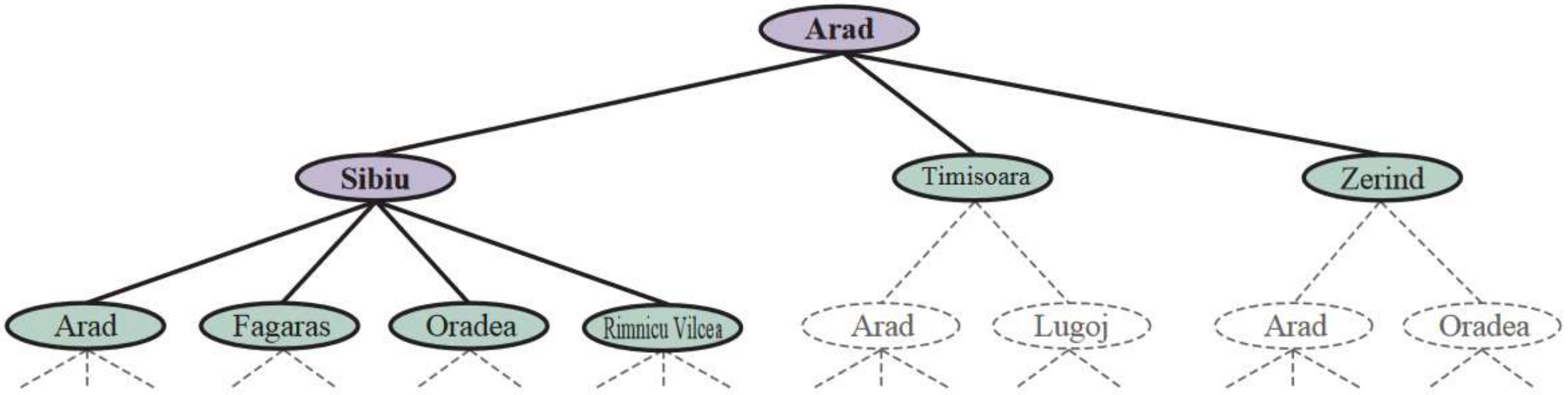
- A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure.
- We consider algorithms that superimpose a **search tree** over the **state space graph**, forming various paths from the initial state, trying to find a path that reaches a goal state.
- Each **node** in the state tree corresponds to state in the state space and the edges in the search tree corresponds to actions.
- The root of the tree corresponds to the initial state of the problem.
- The search tree may have multiple paths to multiple paths to any given state, but each node has a unique path back to the root.
- **Frontier-** Nodes that generated but not expanded to new states.
- Frontier separates two regions of the state space graph- **interior** (all nodes are expanded) and **exterior** (states that have not yet reached).



(a) Partial Tree to find a route from **Arad** to **Bucharest**



(b) Partial Tree to find a route from **Arad** to **Bucharest**



(c) Partial Tree to find a route from **Arad** to **Bucharest**

## Best-first search

- ❑ A general search approach in which we choose a node, *n* , with minimum value of some evaluation function, ***f(n)***.
- ❑ On each iteration we choose a node on the frontier with **minimum *f(n)*** value.
  - ➢ Return it if its state is a goal state
  - ➢ Otherwise apply EXAPAND to generate child nodes
- ❑ Each child node is added to the frontier if it has not been reached before, or, is re-added if it is now being reached with a path that has a lower path cost than any previous path.
- ❑ This algorithm returns either an indication of **failure**, or a node that represents a **path to a goal**.
- ❑ Different ***f(n)*** functions, we get different specific algorithms

## Best-first search: search data structures

- A node in the tree is represented by a data structure with *four components*:
  - ***node*.STATE-** the state to which node corresponds
  - ***node*.PARENT-** the node in the tree that generated this node
  - ***node*.ACTION-** the action that was applied to the parent's state to generate this node
  - ***node*.PATH-COST-** the total cost of the path from the initial state to this node. *g(n)* is a synonym for PATH-COST
- Data structure QUEUE is used to store ***frontier*** with following operations:
  - **IS-EMPTY *(frontier)*-**returns true if there are no nodes in the frontier
  - **POP*(frontier)*-**removes the top node from the frontier and returns it
  - **TOP*(frontier)*-** returns the top node of the frontier
  - **ADD*(node, frontier)*-** inserts node into its proper place in the queue.

## Best-first search: search data structures

- Three kinds of queues are used in search algorithms:
  - A **priority queue-** first pops the node with minimum cost acoording to some evaluation function f(n). It is used by **best-first search**.
  - A **FIFO Queue-** first pops the node that was added to the queue first. It is used by **breadth-first search.**
  - A **LIFO Queue/ Stack-** pops the most recently added node. It is used in depth-first search.

## Best-first search algorithm

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

## Best-first search algorithm

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

## Measuring the performance

- Performance of search strategies are evaluated along the following dimensions:
  - **Completeness:** does it always find a solution if one exists?
  - **Time complexity:** how long does it take to find a solution? (number of nodes generated)
  - **Space complexity:** how much memory is needed to perform the search? **(**maximum number of nodes in memory)
  - **Optimality:** does it always find a least-cost solution?
- **Time and space complexities** are measured in terms of
  - **b:** the *branching factor* or *maximum number of successors* of a node to be considered.
  - **d:** the *depth* or *number of actions* in an optimal solution (i.e., the number of steps along the path from the root)
  - **m:** the *maximum number of actions* in any path(may be $\infty$)

## Types of search strategies

**Uninformed search**

No idea about estimated cost from a node *n* to goal state

- ✓ Breadth-first search
- ✓ Uniform-cost search
- ✓ Depth-first search
- ✓ Depth-limited search
- ✓ Iterative deepening search
- ✓ Bidirectional search

**Informed search**

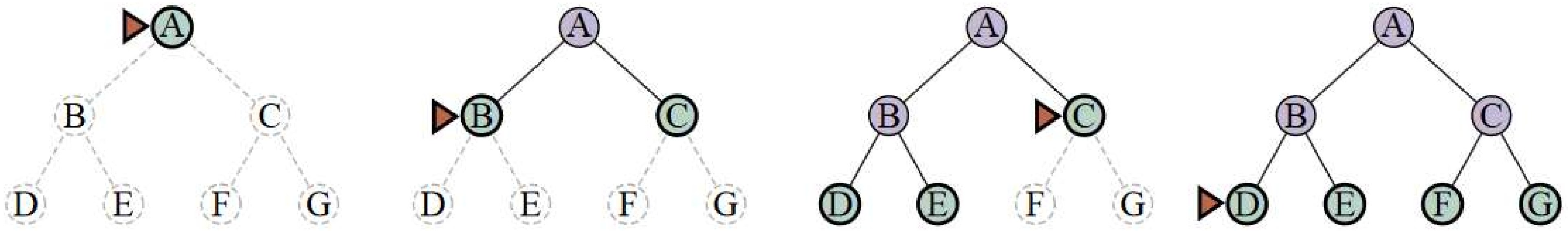
Estimate cost from a node *n* to goal state

- ✓ Greedy best-first search
- ✓ A* search
- ✓ Satisficing search
- ✓ Memory-bounded search

## Breadth-first search

- In **breadth-first search**, root node is expanded first, then all the successors of the root node are expanded, next their successors, and so on. It is systematic search strategy- *complete even on infinite state spaces.*
- Evaluation function, ***f(n)*** is the depth of the node-number of actions it takes to reach the node.
- **FIFO queue** is used *i.e.* new successors join the tail of the queue and old nodes in the queue get expanded first.
- It always find a solution with minimal number of actions, because when it is generating nodes at depth d, it has already generated all the nodes at depth d-1, so if one of them were a solution, it would have found.
- It is **cost optimal** for problems where all actions have the same cost.

## Breadth-first search



**Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.**

## Breadth-first search Algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a FIFO queue, with node as an element
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
            if s is not in reached then
                add s to reached
                add child to frontier
    return failure
```

## Performance measure of breadth-first search

- **Completeness**

  **Yes**, if b is ***finite***

- **Time complexity**

  No of nodes generated up to depth d is,

  $$1 + b + b^2 + b^3 + \cdots + b^d = \boldsymbol{O}(\boldsymbol{b}^{\boldsymbol{d}})$$

- **Space complexity**

  All the nodes remain in memory, so space complexity is also $\boldsymbol{O}(\boldsymbol{b}^{\boldsymbol{d}})$

- **Optimality**

  Yes, if all actions have equal cost. No, otherwise.

## Drawbacks of breadth-first search

- ❑ The memory requirement is a bigger problem for breadth first search than is the execution time.
- ❑ *Assumptions:* branching factor, b = **10**, Time=**10000 nodes/second**, Memory space=**1000** bytes/node.

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| **2** | 100 | 11 seconds | 1 megabyte |
| **4** | 111100 | 11 seconds | 106 megabyte |
| **6** | $10^{7}$ | 19 minutes | 10 gigabyte |
| **8** | $10^{9}$ | 31 hours | 1 terabytes |
| **10** | $10^{11}$ | 129 days | 101 terabytes |
| **12** | $10^{13}$ | 35 years | 10 petabytes |
| **14** | $10^{15}$ | 3523 years | 1 exabyte |

## Uniform-cost search

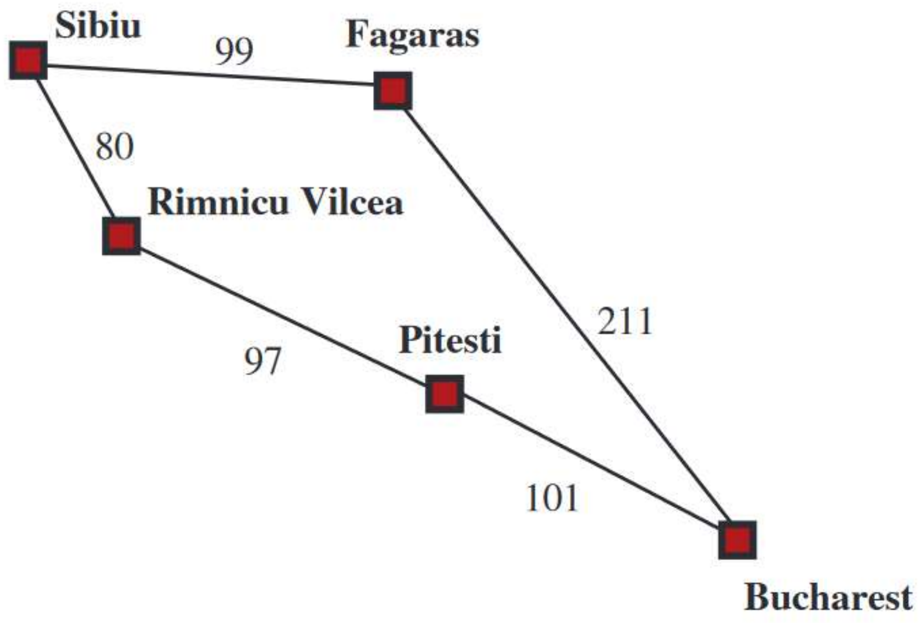
- ❑ Uniform-cost search expands the node *n* with the lowest path cost.
  - ➢ Note that if all step costs are equal, this is identical to *breadth-first search*.
- ❑ Uniform-cost search does not care about the number of steps a path has, but only about their total cost.
- ❑ It will get stuck in an *infinite loop* if it ever expands a node that has a zero-cost action leading back to the same state.

**Algorithm-**

```
function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
    return BEST-FIRST-SEARCH(problem, PATH-COST)
```

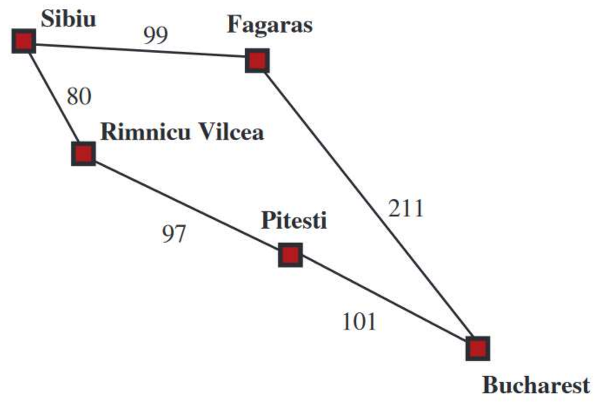
## Uniform-cost search example

- The successors of *Sibiu* are *Rimnicu Vilcea* and *Fagaras*, with costs 80 and 99, respectively.
- The least-cost node, *Rimnicu Vilcea*, is expanded next, adding *Pitesti* with cost 80 + 97=177.



## Uniform-cost search example

- The least-cost node is now *Fagaras*, so it is expanded, adding *Bucharest* with cost 99+211=310.
- Now a goal node has been generated, but uniform-cost search keeps going, choosing *Pitesti* for expansion and adding a second path to *Bucharest* with cost 80+97+101= 278.
- Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. *Bucharest*, now with cost 278, is selected for expansion and the solution is returned.



## Performance measure of uniform-cost search

- **Completeness**

  **Yes,** if step cost ≥ ε where every action costs at least ε

- **Time complexity**

  No of nodes with g ≤ cost of optimal solution, $O\left(b^{1+\lfloor C*/\varepsilon \rfloor}\right)$, where C* is the cost of the optimal solution

- **Space complexity**

  No of nodes with g ≤ cost of optimal solution, $O\left(b^{1+\lfloor C*/\varepsilon \rfloor}\right)$, where C* is the cost of the optimal solution

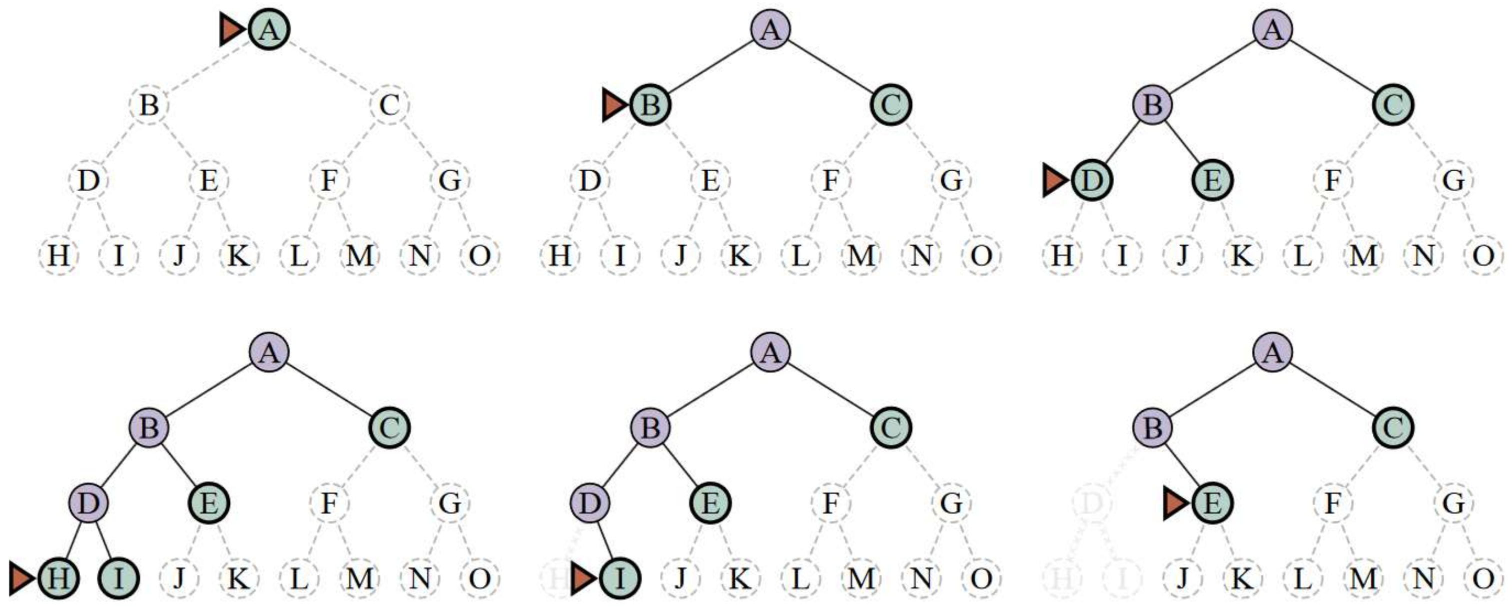
- **Optimality**

  **Yes** – nodes expanded in increasing order of g(n)

## Depth-first search (DFS)

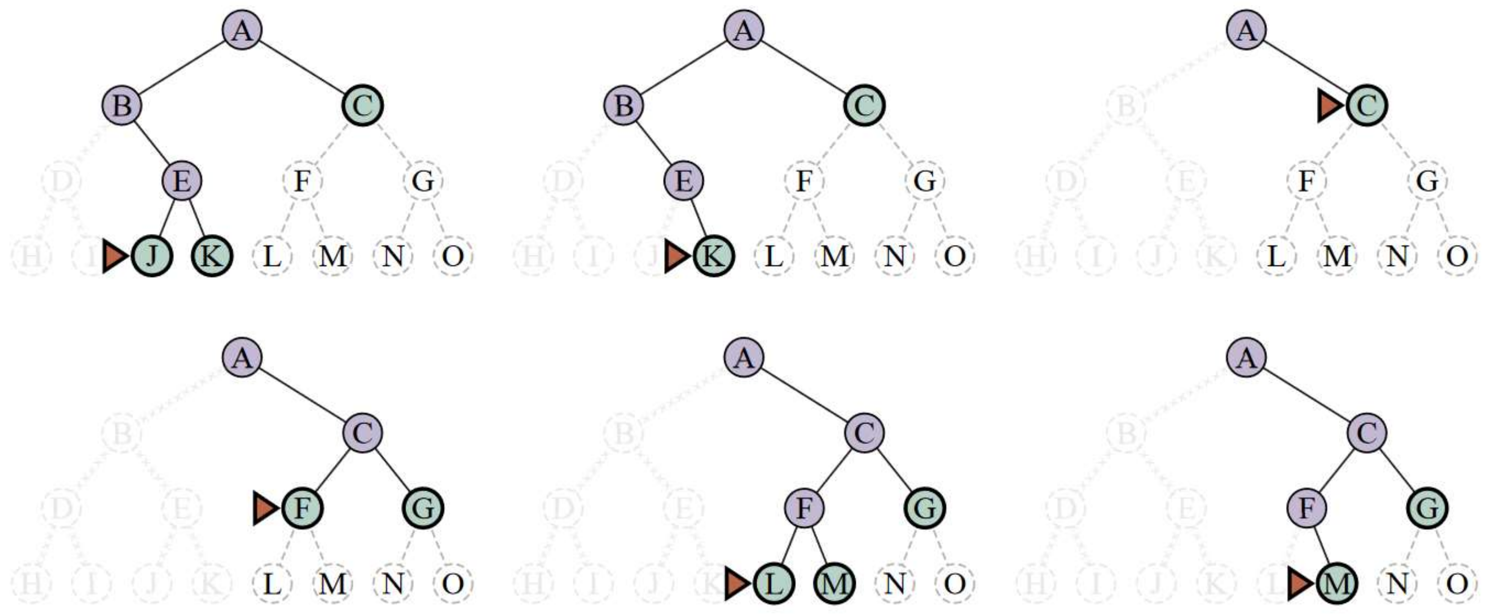
- ❑ Depth-first search always expands the *deepest node* in the current frontier of the search tree.
- ❑ **DFS Algorithm**
  1) Form a one element stack consisting of the root node.
  2) Until the stack is empty or the goal node is found out, repeat the following:-
     - ➢ Remove the first element from the stack. If it is the goal node announce success, return.
     - ➢ If not, add the first element's children if any, to the top of the stack.
  3) If the goal node is not found announce *failure*.

## Depth-first search (DFS)



Progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

## Depth-first search (DFS)



Progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

## Performance measure of depth-first search

- **Completeness**

  No

- **Time complexity**

  $O(b^m)$

- **Space complexity**

  $O(bm)$

- **Optimality**

  No

## Depth-limited search

- The problem with DFS is that the search can go down an infinite branch and thus never return.
- Depth limited search avoids this problem by imposing a depth limit $l$ which effectively terminates the search at that depth. That is , nodes at depth $l$ are treated as if they have no successors.
- The choice of depth parameter $l$ is an important factor.
  - If $l$ is too deep , it is wasteful in terms of both time and space.
  - But if $l < d$ (the depth at which solution exists) i.e. the shallowest goal is beyond the depth limit, then this algorithm will never reach a goal state.

## Depth-limited search algorithm

```
function DEPTH-LIMITED-SEARCH(problem, $\ell$) returns a node or failure or cutoff
  frontier $\leftarrow$ a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result $\leftarrow$ failure
  while not IS-EMPTY(frontier) do
    node $\leftarrow$ POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) $>$ $\ell$ then
      result $\leftarrow$ cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

## Performance measure of depth-limited search

- **Completeness**

  Yes, if l >= d

- **Time complexity**

  $O(b^l)$

- **Space complexity**

  $O(bl)$

- **Optimality**

  No

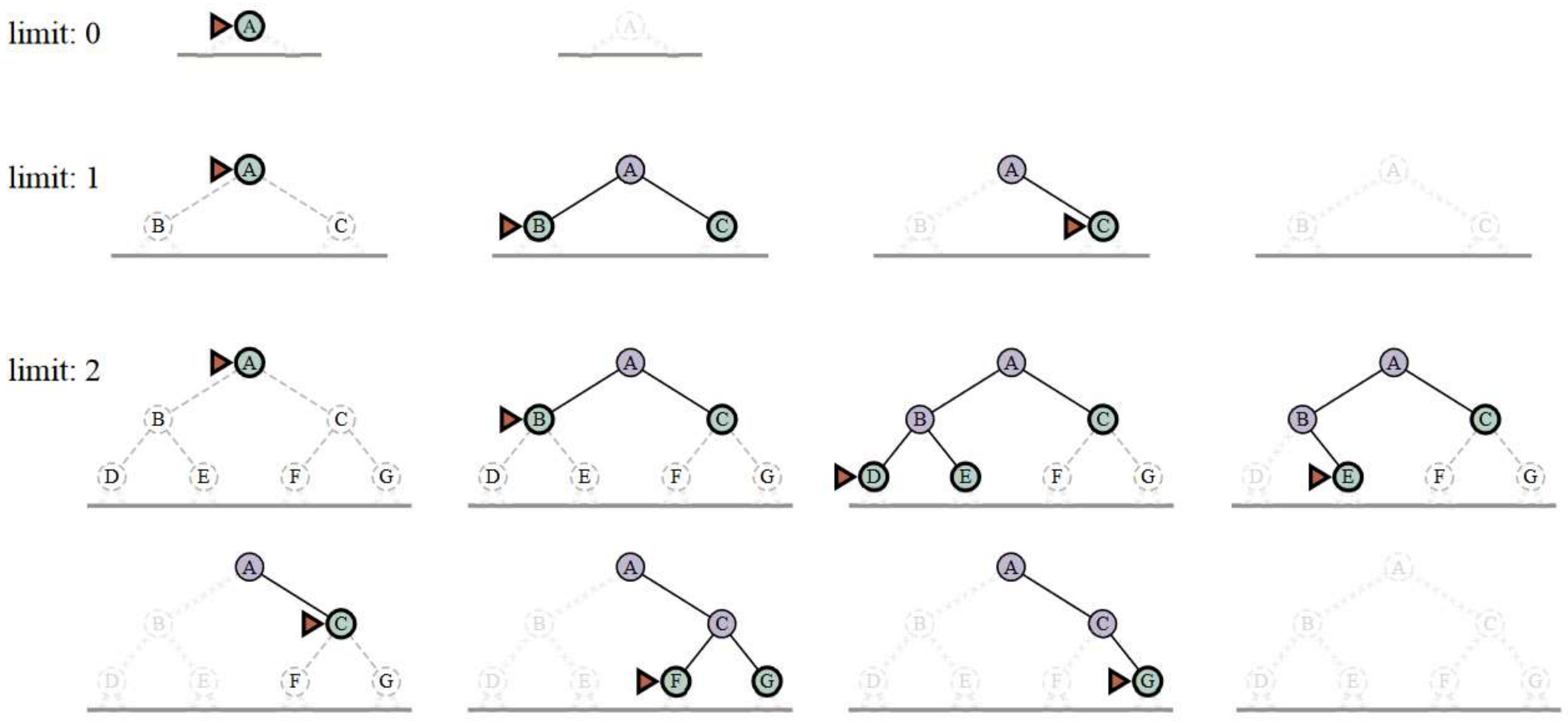
## Iterative deepening search

- The problem with depth-limited search is deciding on a suitable depth parameter, which is not always easy.
- To overcome this problem there is another search called *iterative deepening search*. This search method simply tries all possible depth limits; first 0, then 1, then 2 etc. until a solution is found.
- Iterative deepening combines the ***benefits*** of ***DFS*** and ***BFS***. It may appear wasteful as it is expanding nodes multiple times. But the overhead is small in comparison to the growth of an exponential search tree.

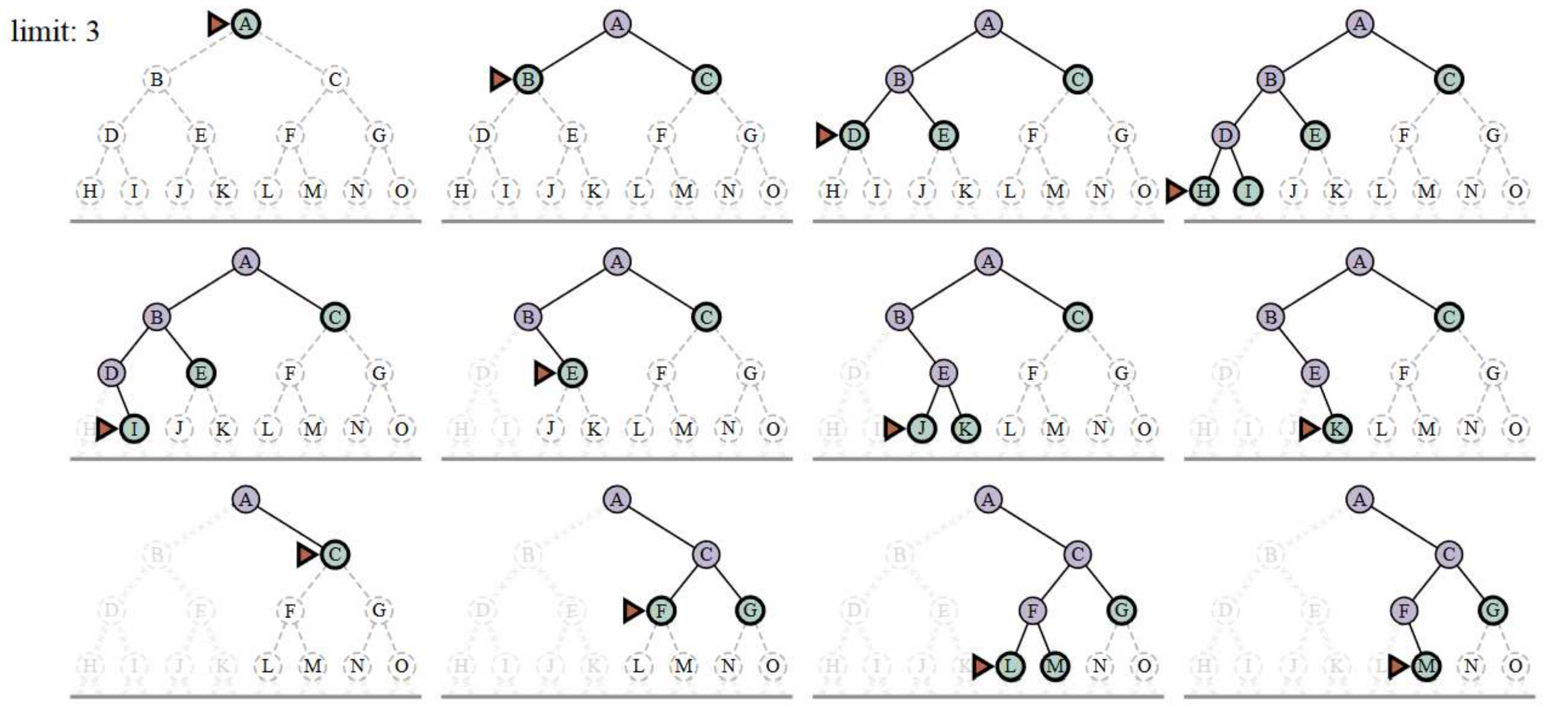
## Iterative deepening search Algorithm

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to $\infty$ do
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(problem, depth)
    if result $\neq$ cutoff then return result
```

## Illustration of Iterative deepening search



## Illustration of Iterative deepening search



## Performance measure of Iterative deepening search

- **Completeness**

  Yes , like BFS it is complete when b is finite

- **Time complexity**

  $O(b^d)$

- **Space complexity**

  $O(bd)$, like DFS memory requirement is linear

- **Optimality**

  Yes, if like BFS path cost is a non-decreasing function of the depth of the node, it is optimal.

## Bidirectional search

- The idea behind bidirectional search is to run two simultaneous searches—
  - one ***forward*** from the *initial state* and
  - the other ***backward*** from the *goal state*
- Motivation
  - the two searches meet in the middle
  - $\boldsymbol{b^{d/2}} + \boldsymbol{b^{d/2}}$ is much less than $\boldsymbol{b^{d}}$

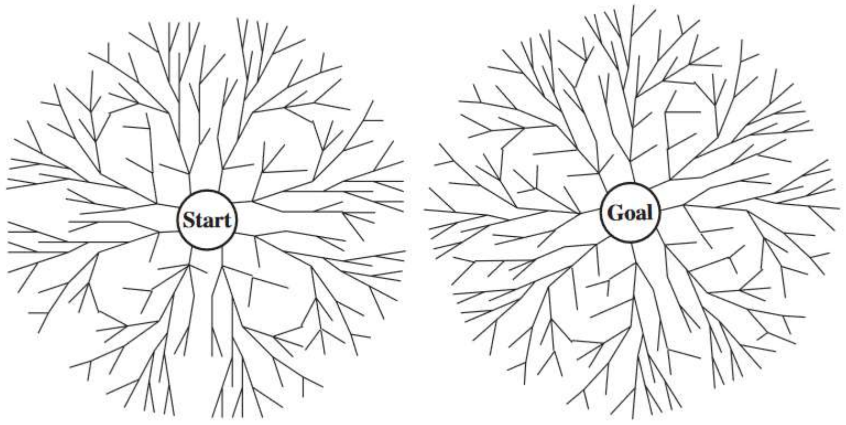
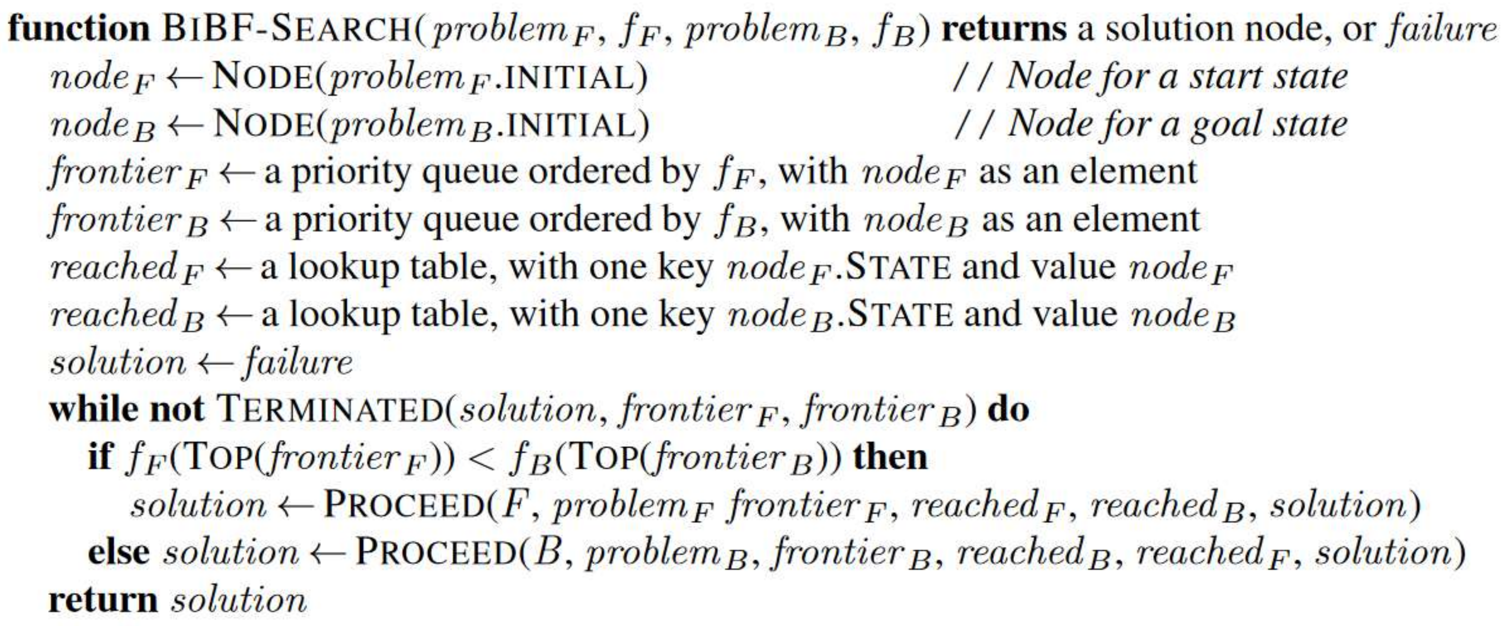


**Figure.** A schematic view of a bidirectional search that is about to succeed when a branch from the *start node* meets a branch from the *goal node*.

## Bidirectional search

- The reduction in time complexity makes bidirectional search attractive, but
  - ***how do we search backward?***
  - When all the actions in the state space are reversible, the predecessors of *x* are just its successors.
- Bidirectional search is implemented by replacing the *goal test* with a check to see whether the *frontiers of the two searches intersect*; if they do, a solution has been found.

## Bidirectional search algorithm



```
function BIBF-SEARCH($problem_F$, $f_F$, $problem_B$, $f_B$) returns a solution node, or failure
  $node_F \leftarrow$ NODE($problem_F$.INITIAL)                    // Node for a start state
  $node_B \leftarrow$ NODE($problem_B$.INITIAL)                    // Node for a goal state
  $frontier_F \leftarrow$ a priority queue ordered by $f_F$, with $node_F$ as an element
  $frontier_B \leftarrow$ a priority queue ordered by $f_B$, with $node_B$ as an element
  $reached_F \leftarrow$ a lookup table, with one key $node_F$.STATE and value $node_F$
  $reached_B \leftarrow$ a lookup table, with one key $node_B$.STATE and value $node_B$
  $solution \leftarrow failure$
  while not TERMINATED($solution$, $frontier_F$, $frontier_B$) do
    if $f_F$(TOP($frontier_F$)) $<$ $f_B$(TOP($frontier_B$)) then
      $solution \leftarrow$ PROCEED($F$, $problem_F$ $frontier_F$, $reached_F$, $reached_B$, $solution$)
    else $solution \leftarrow$ PROCEED($B$, $problem_B$, $frontier_B$, $reached_B$, $reached_F$, $solution$)
  return $solution$
```

## Bidirectional search algorithm

```
function PROCEED($dir$, $problem$, $frontier$, $reached$, $reached_2$, $solution$) returns a solution
        // Expand node on frontier; check against the other frontier in $reached_2$.
        // The variable "dir" is the direction: either F for forward or B for backward.
  $node \leftarrow$ POP($frontier$)
  for each $child$ in EXPAND($problem$, $node$) do
    $s \leftarrow child$.STATE
    if $s$ not in $reached$ or PATH-COST($child$) $<$ PATH-COST($reached[s]$) then
      $reached[s] \leftarrow child$
      add $child$ to $frontier$
      if $s$ is in $reached_2$ then
        $solution_2 \leftarrow$ JOIN-NODES($dir$, $child$, $reached_2$[s]))
        if PATH-COST($solution_2$) $<$ PATH-COST($solution$) then
          $solution \leftarrow solution_2$
  return $solution$
```

## Performance measure of bidirectional search

- **Completeness**

  **Yes**, if *b* is finite and both directions use breadth-first search

- **Time complexity**

  $\mathbf{O}\left(\boldsymbol{b}^{\boldsymbol{d}/2}\right)$

- **Space complexity**

  $\mathbf{O}\left(\boldsymbol{b}^{\boldsymbol{d}/2}\right)$

- **Optimality**

  **Yes**, if step costs are all identical and both directions use breadth-first search

## Comparison of performance measures of uninformed search

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Complete? | Yes[1] | Yes[1,2] | No | No | Yes[1] | Yes[1,4] |
| Optimal cost? | Yes[3] | Yes | No | No | Yes[3] | Yes[3,4] |
| Time | $O(b^d)$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(b^m)$ | $O(b^\ell)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^d)$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(bm)$ | $O(b\ell)$ | $O(bd)$ | $O(b^{d/2})$ |

**Figure** Evaluation of search algorithms. $b$ is the branching factor; $m$ is the maximum depth of the search tree; $d$ is the depth of the shallowest solution, or is $m$ when there is no solution; $\ell$ is the depth limit. Superscript caveats are as follows: [1] complete if $b$ is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

## Informed(heuristic) search strategies

- ❑ An informed search strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions *more efficiently* than can an uninformed strategy.
- ❑ The general approach we consider is called ***best-first search***. Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function, *f(n)*.
- ❑ The choice of *f* determines the search strategy.
- ❑ Most best-first algorithms include as a component of *f* a heuristic function, denoted *h(n)*:
  - ➢ *h(n)* = estimated cost of the cheapest path from the state at *node n* to a *goal state*.

## Best-first search

- ❑ When nodes are ordered so that the node with the best evaluation is expanded first the resulting strategy is called *best-first search*.
- ❑ An algorithm is best-first search algorithm if it minimizes the total cost of a path from *start node* to the *goal node*.
- ❑ Evaluation function *f*(n), which denotes the estimate of total cost along a path through n is given by

$$\boldsymbol{f(n) = g(n) + h(n)}$$

Where,

***g(n)-*** actual cost to reach node n

***h(n)-*** estimate of cost to reach goal state from node n

## Best-first search

- **Idea:** use an evaluation function f(n) for each node Use this to estimate the degree of "desirability“ for each node. Then expand the most desirable unexpanded node.
- **Implementation:** Order the nodes in fringe or sequence in decreasing order of desirability.
- Two special cases of Best-first search where *h(n)* is used as *f(n)* to guide search:
  - **Greedy best-first search** [only *h(n)* is used as *f(n)* i.e. ***f(n)=h(n)***]
  - **A* search** [*h(n)* + *g(n)* is used as *f(n)* i.e. ***f(n)=g(n)+h(n)***]

## Conditions for optimality: *Admissibility* and *consistency*

- **Admissibility-**

The first condition we require for optimality is that *h(n)* be an admissible heuristic.

- An admissible heuristic is one that never overestimates the cost to reach the goal.
- Because *g(n)* is the actual cost to reach n along the current path, and ***f(n) = g(n) + h(n)***, we have as an immediate consequence that f(n) never overestimates the true cost of a solution along the current path through n.
- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is. Example, straight-line distance $h_{SLD}$

## Conditions for optimality: ***Admissibility*** and ***consistency***

- **Consistency-**

  A heuristic *h(n)* is ***consistent*** if, for every node $n$ and every successor $n'$ of $n$ generated by any action $a$, the estimated cost of reaching the goal from $n$ is no greater than the step cost of getting to $n'$ plus the estimated cost of reaching the goal from $n'$.

$$\boldsymbol{h(n) \leq c(n, a, n') + h(n')}$$

  This is a form of the general ***triangle inequality***, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides.

## Greedy best-first search

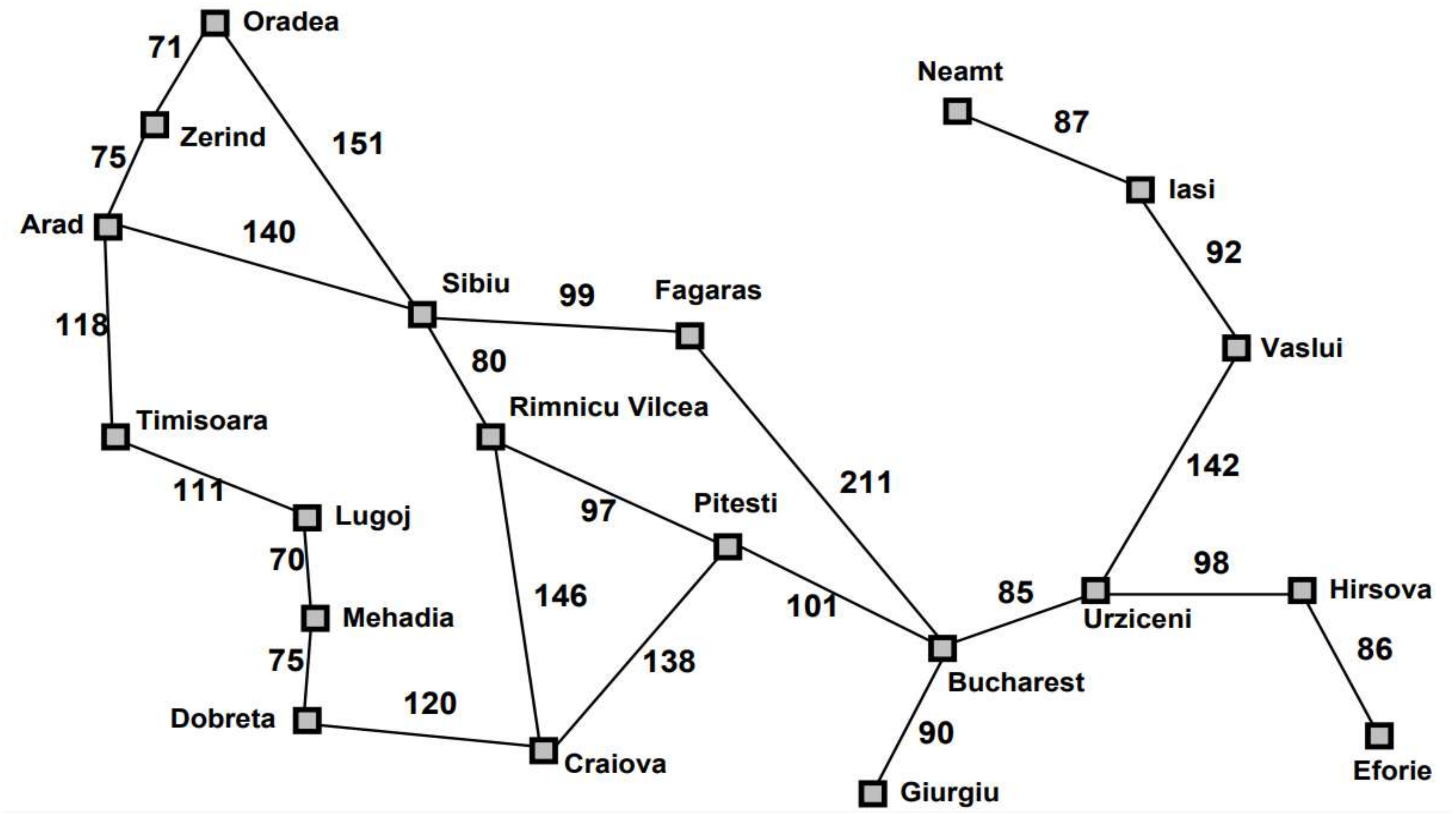
- It is an *informed search* algorithm where the ***evaluation function*** is strictly equal to the ***heuristic function***, disregarding the edge weights in a weighted graph.
- To get from a start node to a target node, the lowest value resulting from some heuristic function, *h(x)*, is considered as the successive node to traverse to.
- The goal is to choose the quickest and shortest path to the target node.
- The evaluation function, ***f(x),*** for the greedy best-first search algorithm is the following: ***f(x)=h(x)***

  Here, the evaluation function is equal to the heuristic function. Since this search disregards edge weights, finding the lowest-cost path is not guaranteed.

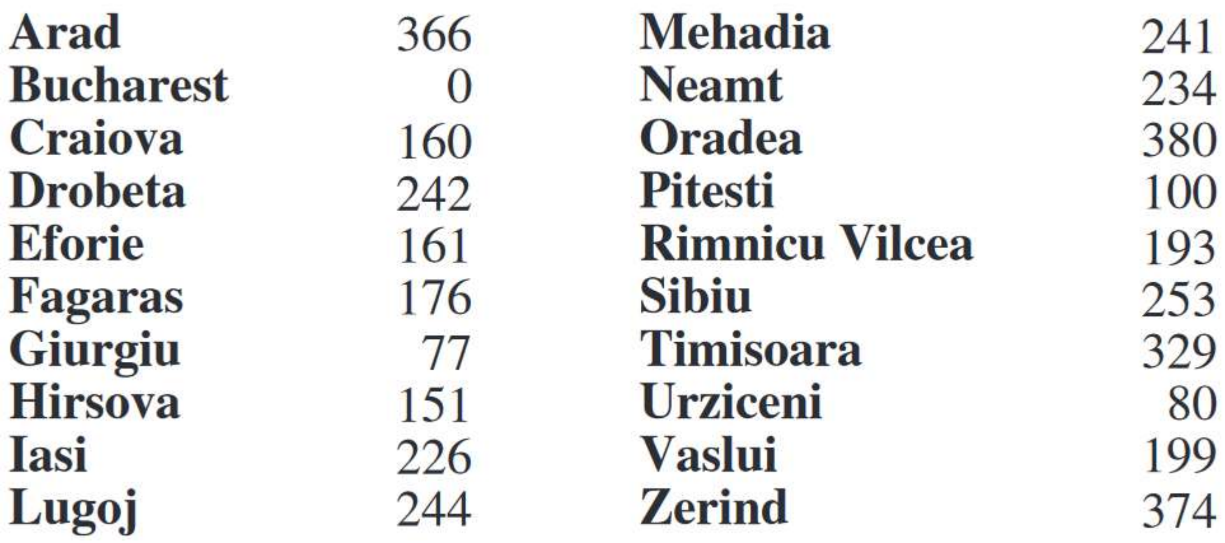
## Greedy best-first search

- **Heuristic Function-** A heuristic function, *h(x)*, evaluates the successive node based on *how close it is to the target node*. In other words, it chooses the immediate low-cost option.
  - it does not necessarily find the shortest path to the goal.
- **Greedy best-first search algorithm-**
  - Initialize a tree with the root node being the start node in the OPEN list.
  - If the open list is empty, return a failure, otherwise, add the current node to the CLOSED list.
  - Remove the node with the lowest *h(x)* value from the OPEN list for exploration.
  - If a child node is the target, return a success. Otherwise, if the node has not been in either the OPEN or CLOSED list, add it to the OPEN list for exploration.

## **Illustration of greedy best-first search:** *Romanian example*



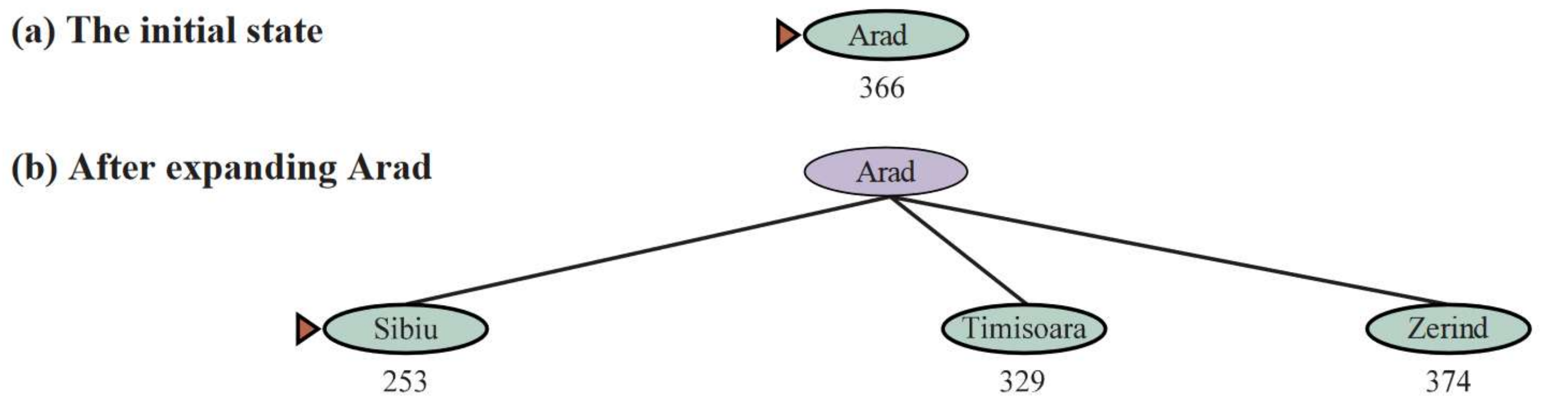
## **Illustration of greedy best-first search:** *Romanian example*



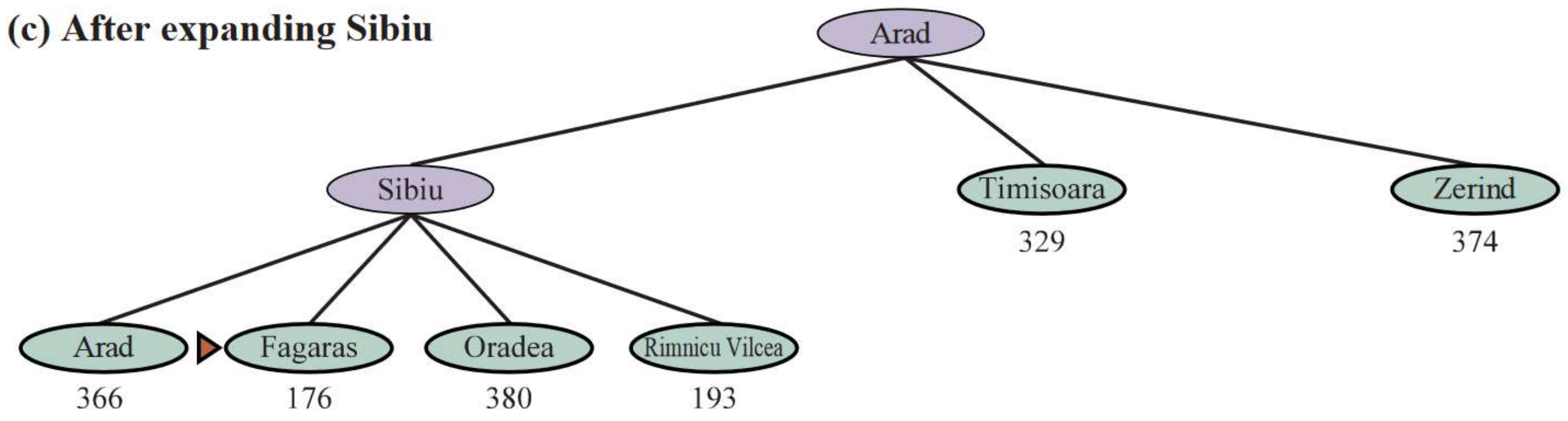
| | | | |
|---|---|---|---|
| Arad | 366 | Mehadia | 241 |
| Bucharest | 0 | Neamt | 234 |
| Craiova | 160 | Oradea | 380 |
| Drobeta | 242 | Pitesti | 100 |
| Eforie | 161 | Rimnicu Vilcea | 193 |
| Fagaras | 176 | Sibiu | 253 |
| Giurgiu | 77 | Timisoara | 329 |
| Hirsova | 151 | Urziceni | 80 |
|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

Values of $\boldsymbol{h_{SLD}}$ — straight-line distances to Bucharest

## Illustration of greedy breadth-first search

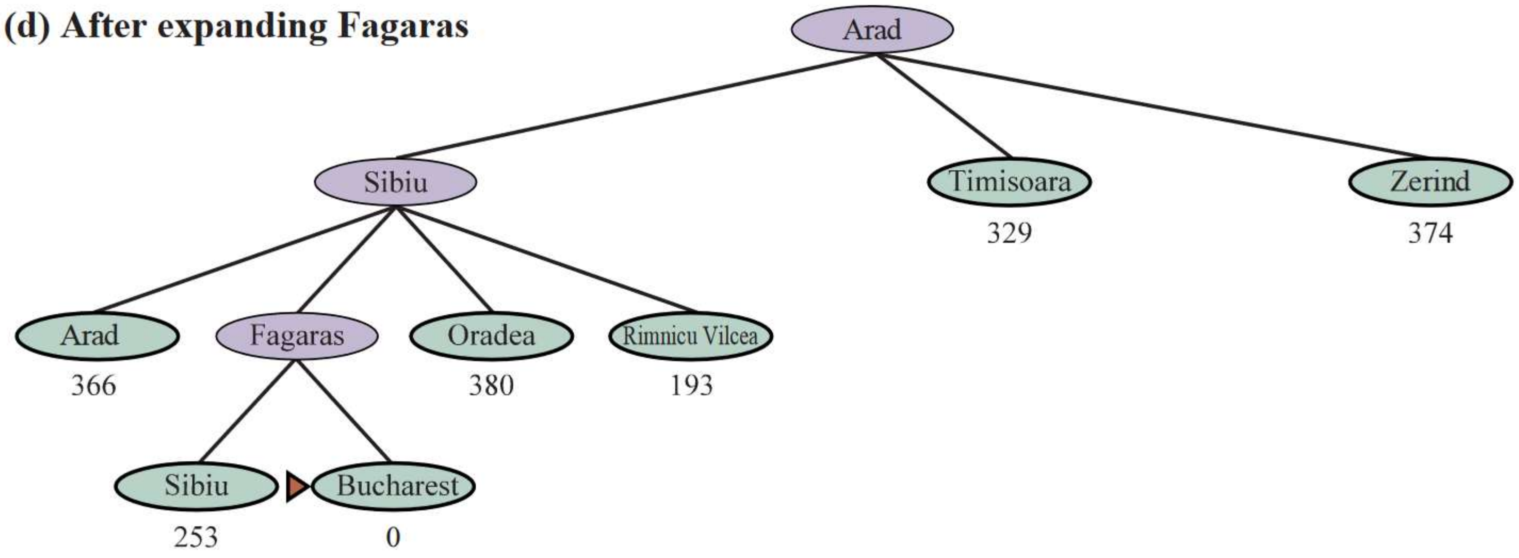


## Illustration of greedy breadth-first search



## Illustration of greedy breadth-first search

(d) After expanding Fagaras



## Drawbacks of greedy best-first search

- ❑ Greedy best-first search results the path cost 450 to reach *Bucharest* from *Arad* using greedy choice of heuristic function value.

  **Arad** (0)→**Sibiu**(140)→ **Fagaras** (140+99) → **Bucharest** (140+99+211)

  **Path cost=450**

  But **actual minimum cost to reach *bucharest* is 418** via…

  **Arad** (0)→**Sibiu** (140)→ **Rim. Vilcea** (140+80) → **Pitesti** (140+80+97)

  → **Bucharest**(140+80+97+101)

  **Path cost=418**

- ❑ Greedy best-first search is ***not optimal.***

## Performance measure of greedy best-first search

- ❑ **Completeness**

  **Yes** , in finite state spaces. No, otherwise.

- ❑ **Time complexity**

  $\mathbf{O}(\boldsymbol{b}^{\boldsymbol{m}})$-a good heuristic can reduce this substantially up to $O(bm)$

- ❑ **Space complexity**

  $\mathbf{O}(\boldsymbol{b}^{\boldsymbol{m}})$- keeps all nodes in memory

- ❑ **Optimality**

  **No**

## A* search

**Idea:** Avoid expanding paths that are already expensive

Evaluation function, ***f(n) = g(n) + h(n)***

***g(n)*** = cost so far to reach node ***n*** from the ***initial state***

***h(n)*** = estimated cost from node ***n*** to ***goal state***

***f(n)*** = estimated total cost of path through ***n*** to reach ***goal state***

- *The Best first search algorithm is a simplified version of A* algorithm. A* uses the same f, g ,h functions as well as the lists OPEN and CLOSED.*

## A$^*$ search algorithm

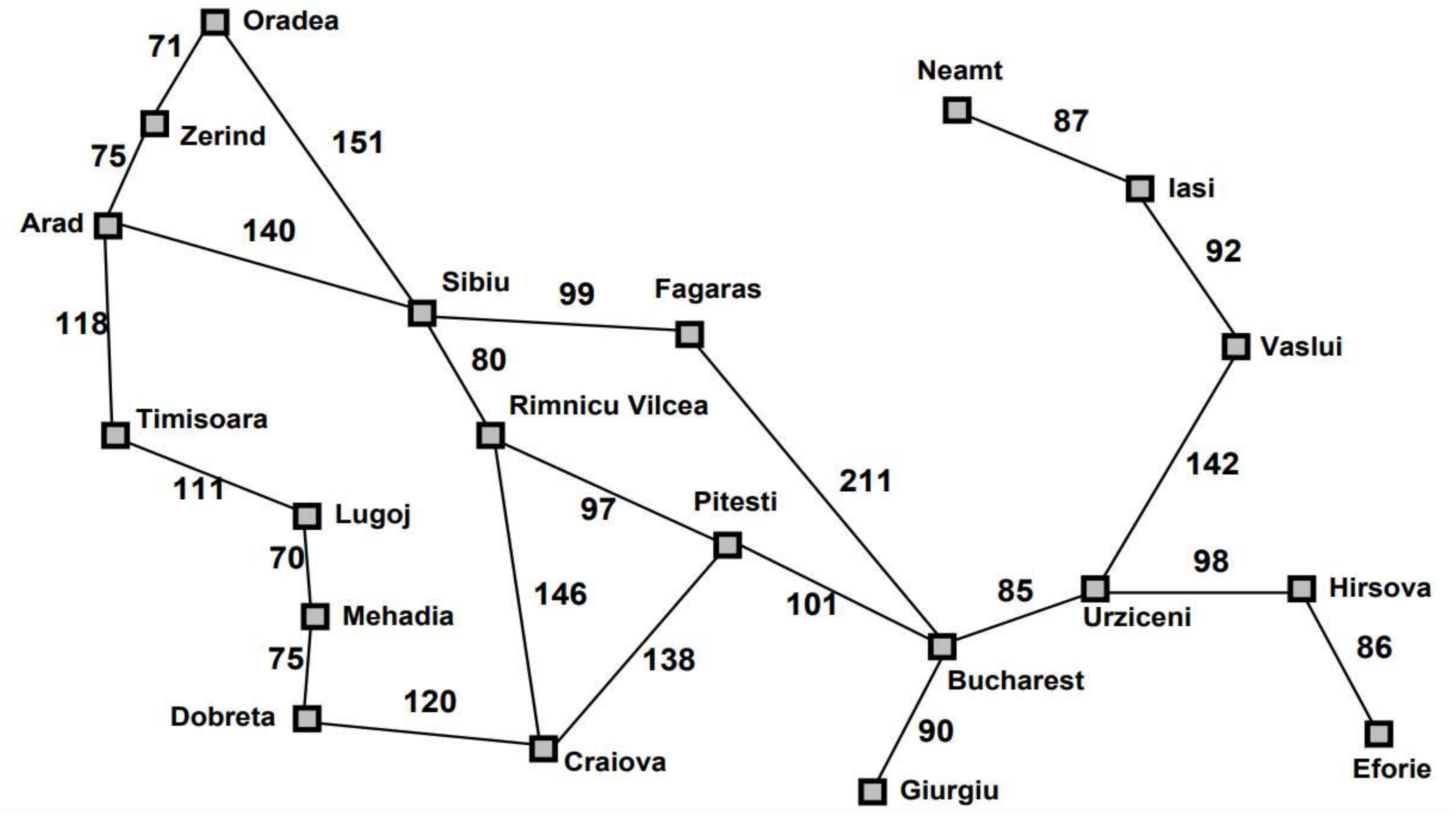
Create two lists *OPEN* and *CLOSED*. OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet. *CLOSED* contains those nodes that have already been visited, which is initially empty.

```
1) Put the starting node in the OPEN list
2) while OPEN≠empty
    Find a node $q$ from the OPEN list with minimum $f(n)$ value and
    add it to CLOSED list
     if $q = goal\ state$
          return success and exit
```

## A$^*$ search algorithm

```
    Find the set of successors of $q$ (let say $S$)
    For every, $s \epsilon S$
    if $s = goal\ state$
        return success and exit
    else
        Compute $g(n)$ and $h(n)$ for successor $s$ as follows
        $s.g(n) = q.g(n) + d(s,q)$
        $s.h(n) = d(s, goal\_state)$ //Apply Heuristics to find $d(s, goal_{state})$
        $s.f(n) = s.g(n) + s.h(n)$
        $\forall s' \epsilon\ OPEN, if\ f(s') < f(s)$, then skip $s$ (where $s'$ has same position as successor $s$)
        $\forall s' \epsilon\ CLOSED, if\ f(s') < f(s)$, then skip $s$, otherwise add $s'$ to the OPEN list.
3) Add $q$ to the closed list
4) Exit
```

## Illustration of $A^*$ search: *Romanian example*

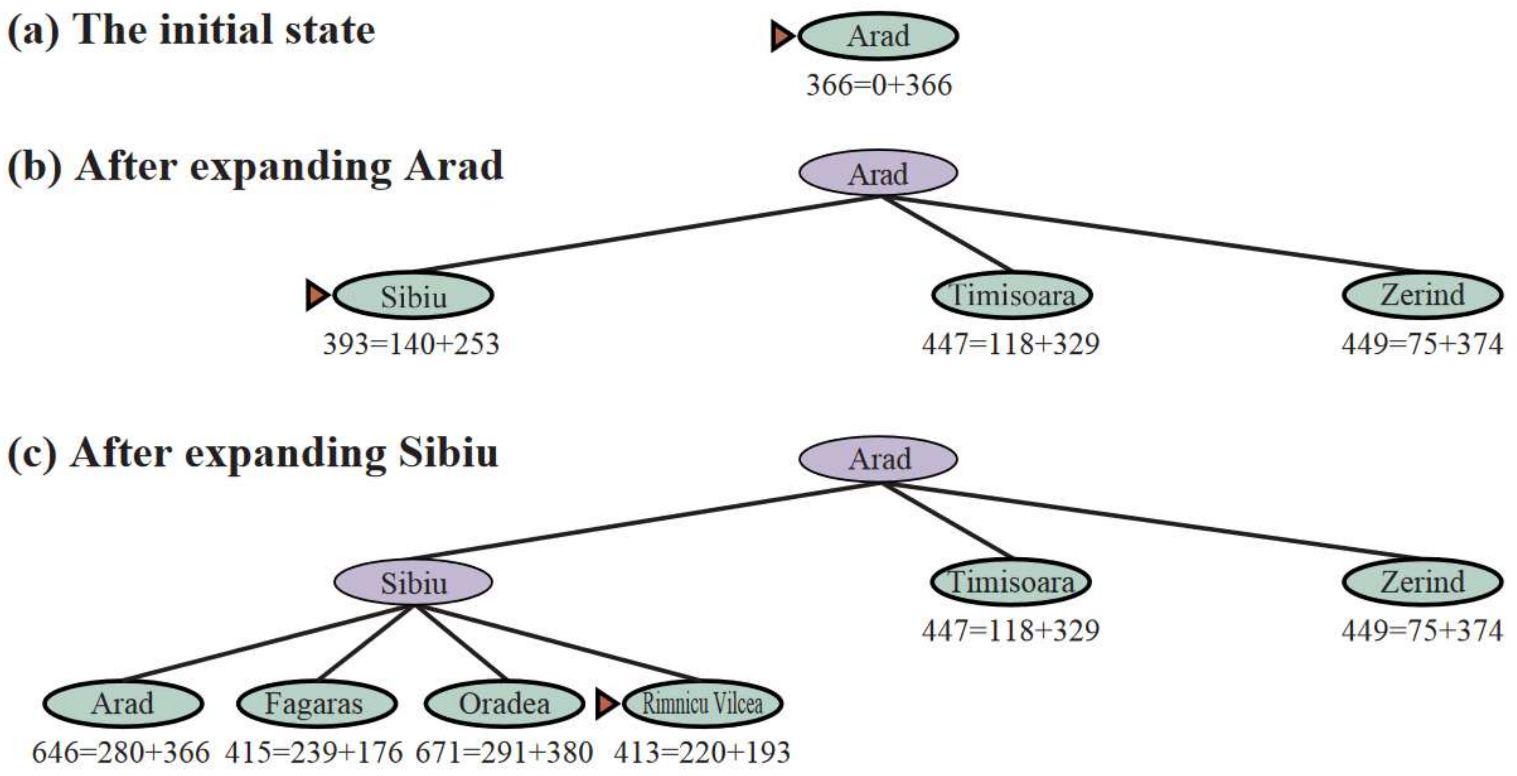


## Illustration of $A^*$ search: *Romanian example*

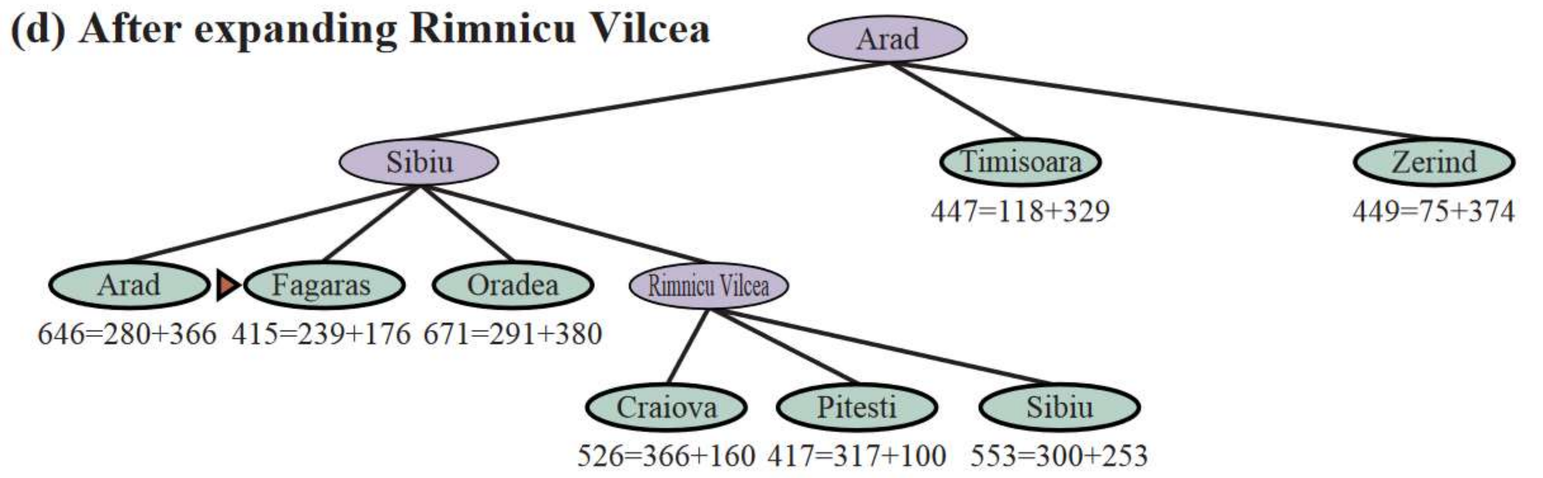
**Table.** Values of $h_{SLD}$—straight-line distances to Bucharest

| | | | |
|---|---|---|---|
| **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** |

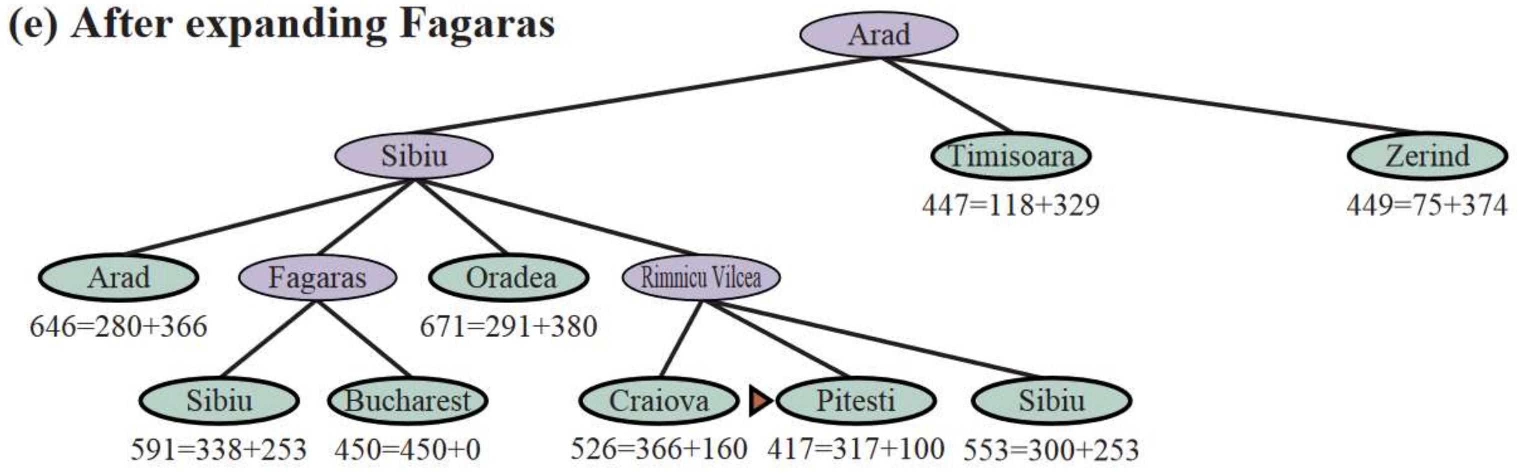
## Illustration of $A^*$ search



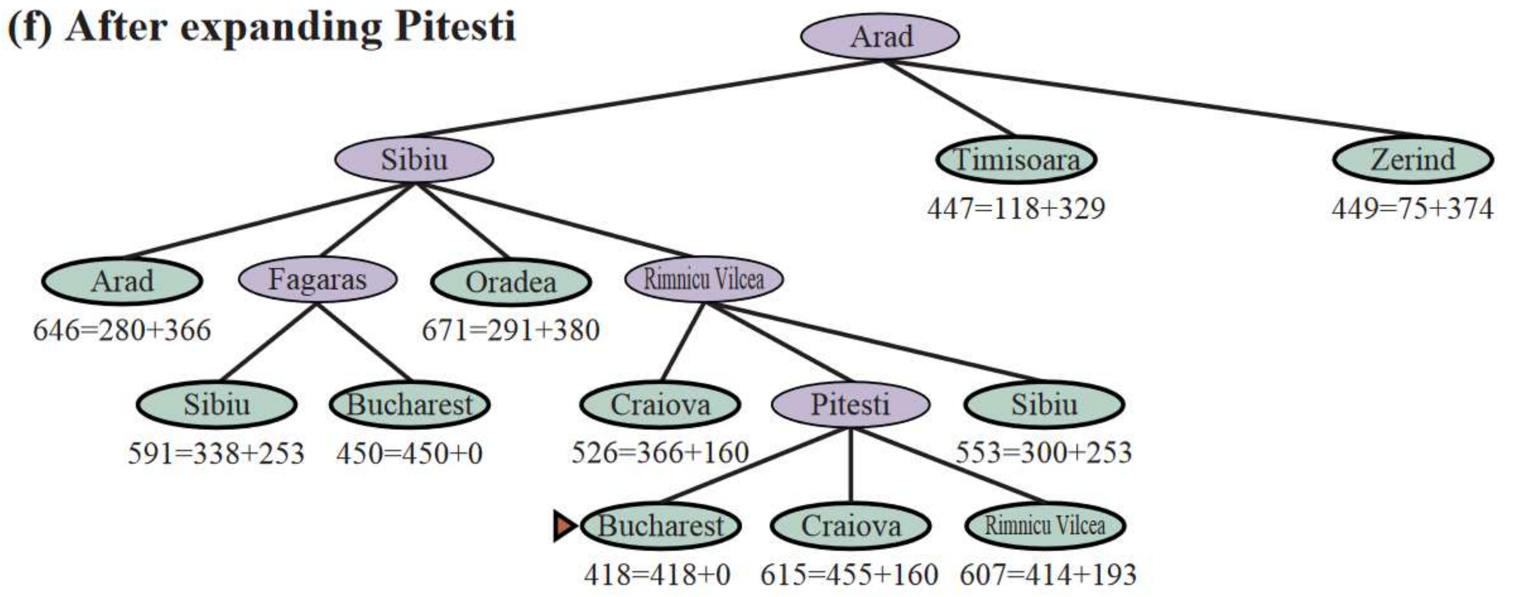
## Illustration of $A^*$ search



## Illustration of $A^*$ search

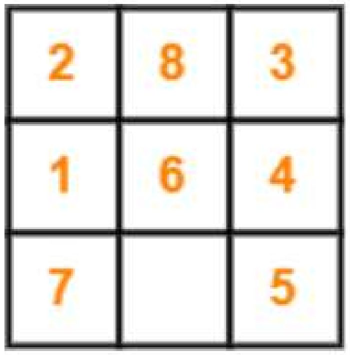


## Illustration of $A^*$ search



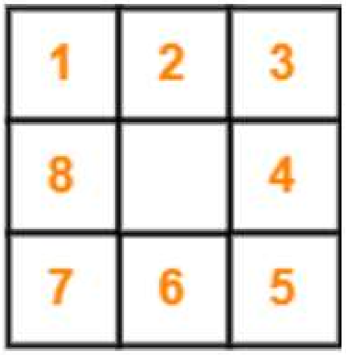
## A* search: *More examples*

Given an initial state of a ***8-puzzle*** problem and final state to be reached as follows. Find the most cost-effective path to reach the final state from initial state using A* Algorithm. Consider ***g(n)* = Depth of node** and ***h(n*) = Number of misplaced tiles**.



| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

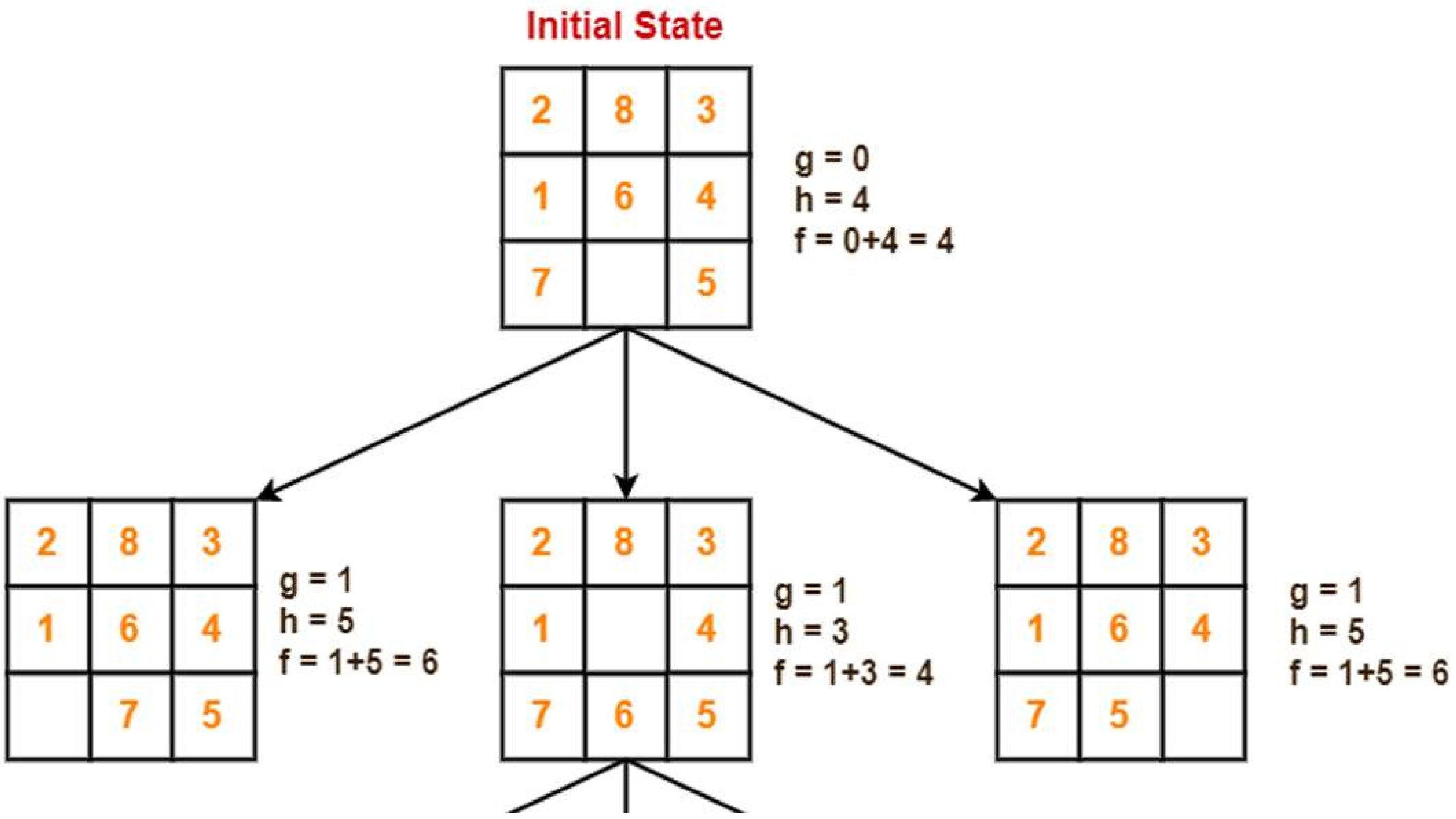
**Intial State**



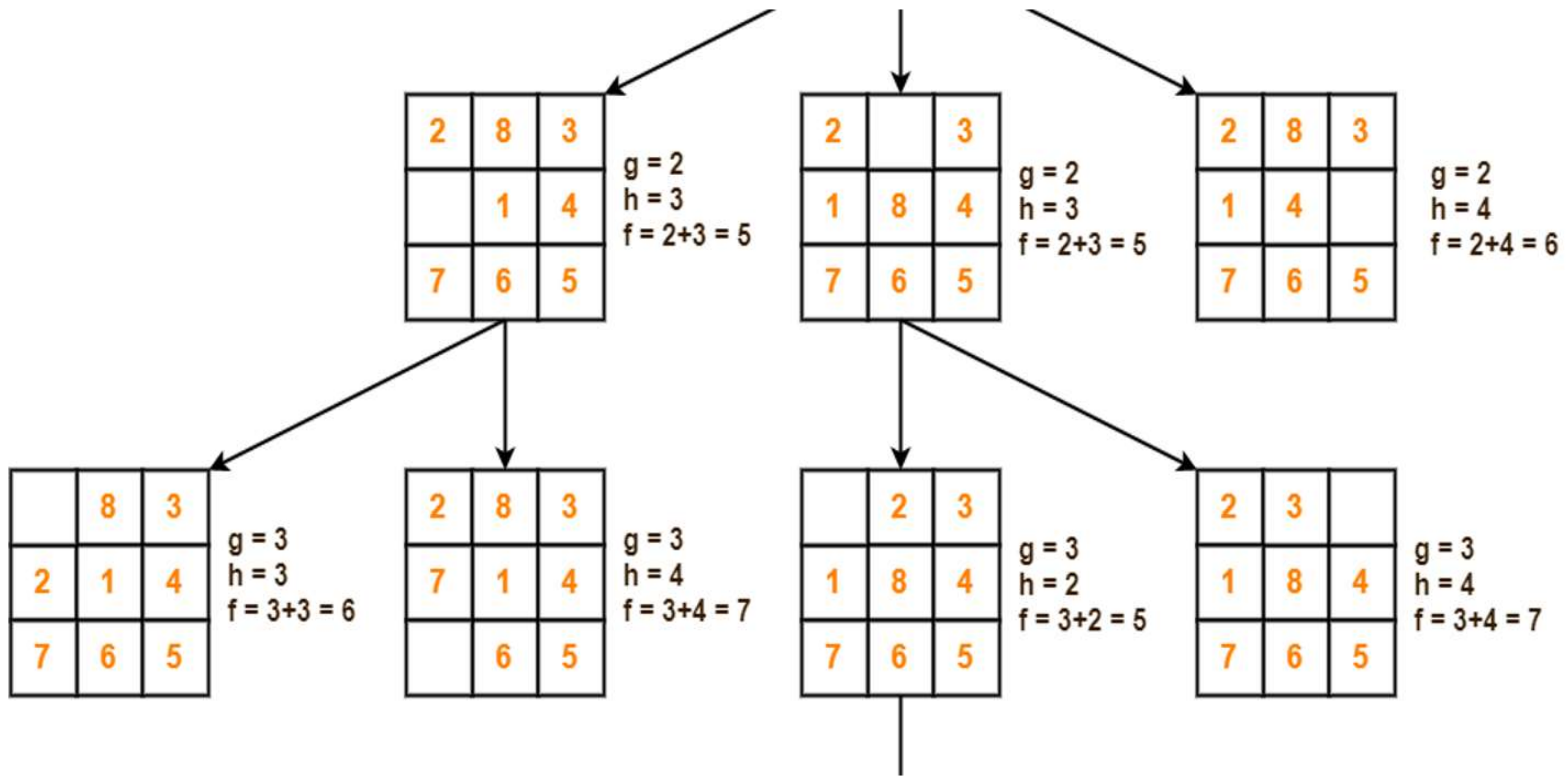
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

**Final State**

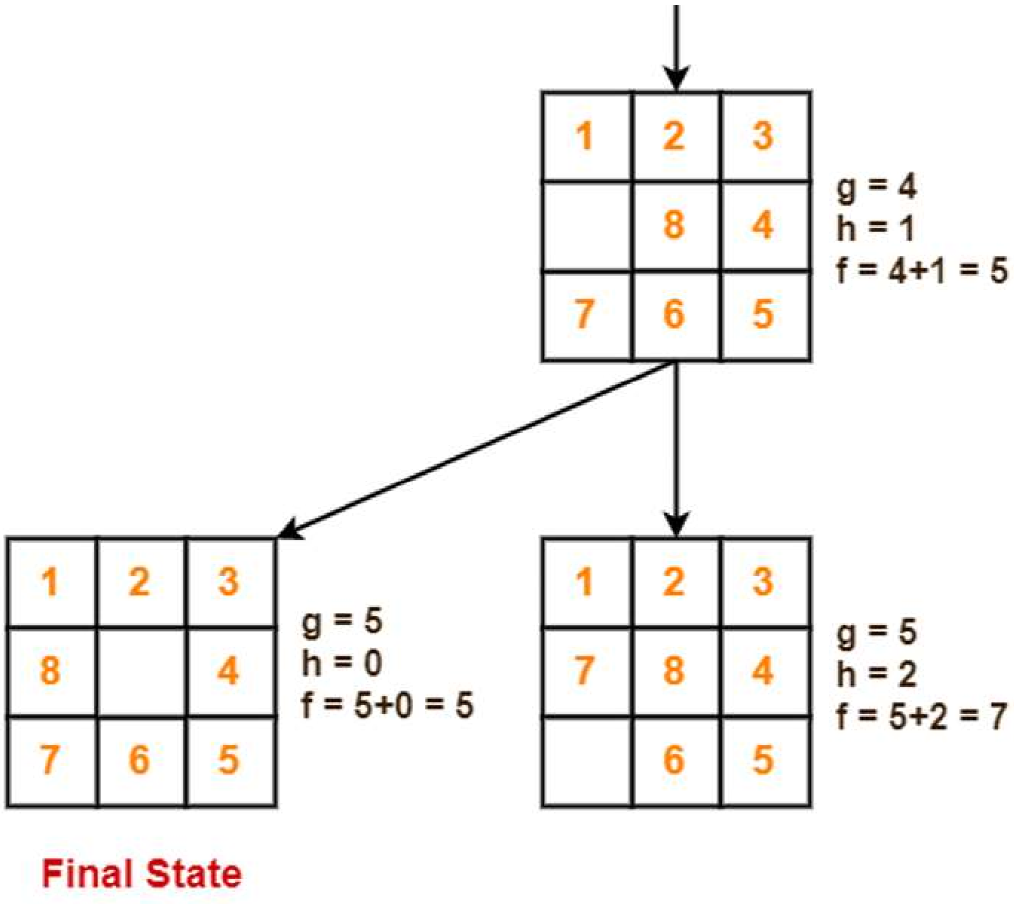
## A* search: *More examples*



## A* search: *More examples*

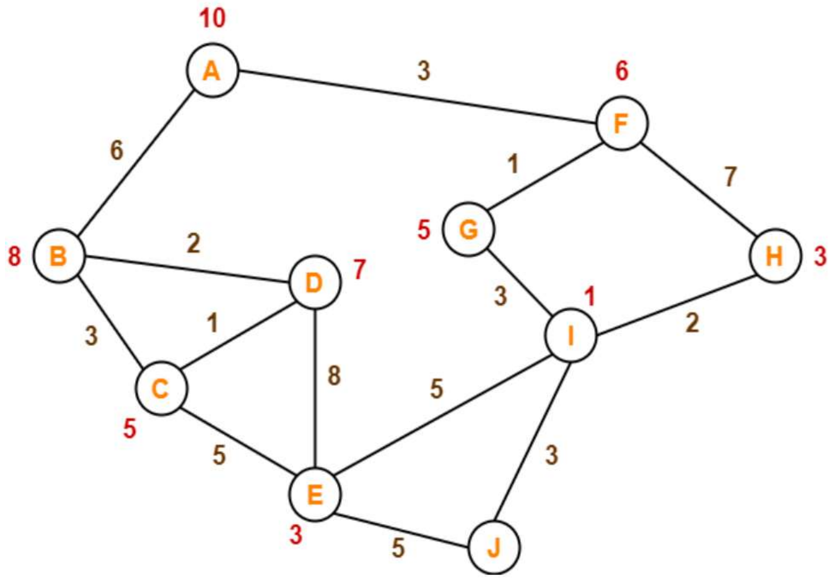


## A* search: *More examples*



## A$^*$ search: *More examples*

Find the most cost-effective path to reach from *start state* A to *final state* J using *A* Algorithm*. Assume that the numbers written on edges represent the distance between the nodes and the numbers written on nodes represent the ***heuristic value***.



## A* search: *More examples*

**Iteration 1**

We start with node A. Node B and Node F can be reached from node A. A* Algorithm calculates $f(B)$ and $f(F)$.

$f(B)=6+8=14$

$f(F)=3+6=9$

Since, $f(F)<f(B)$, so it decides to go to node F.

Path: $\mathbf{A} \rightarrow \mathbf{F}$

## A$^*$ search: *More examples*

### Iteration 2

Node G and Node H can be reached from node F. A* Algorithm calculates $f$(G) and $f$(H).

$f(G)=(3+1)+5=9$

$f(H)=(3+7)+3=13$

Since $f(G)<f(H)$, so it decides to go to node G.

Path: **A** → **F** → **G**

## $A^*$ search: *More examples*

### Iteration 3

Node I can be reached from node G.A* Algorithm calculates $f(\mathrm{I})$.

$f(\mathrm{I}) = (3+1+3) + 1 = 8$

It decides to go to node I.

Path: $\mathbf{A} \rightarrow \mathbf{F} \rightarrow \mathbf{G} \rightarrow \mathbf{I}$

## A* search: *More examples*

**Iteration 4**

Node E, Node H and Node J can be reached from node I. A* Algorithm calculates *f*(E),*f*(H) and *f*(J).

*f*(E)=(3+1+3+5)+3=15

*f*(H)=(3+1+3+2)+3=12

*f*(J)=(3+1+3+3)+0=10

Since f(J) is least, so it decides to go to node J.

Path: **A** → **F** → **G** → **I** → **J**

## $A^*$ search: *More examples*

**Iteration 4**

Node E, Node H and Node J can be reached from node I. A* Algorithm calculates $f$(E),$f$(H) and $f$(J).

$f(E)=(3+1+3+5)+3=15$

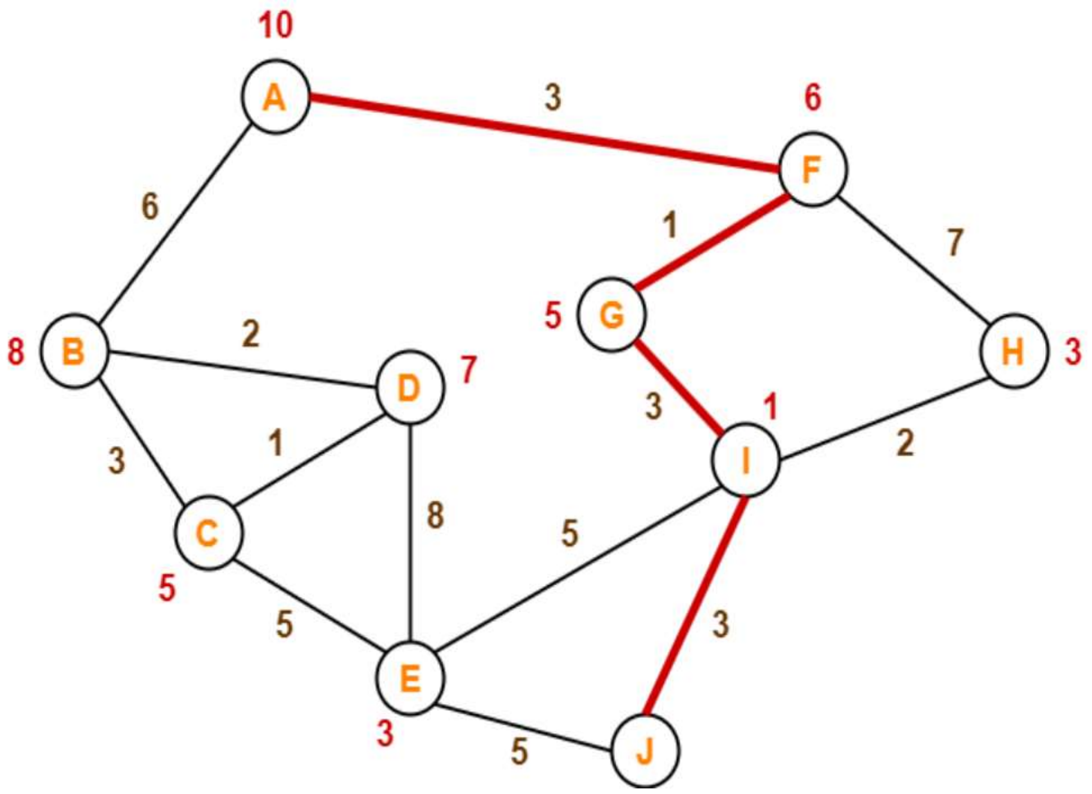
$f(H)=(3+1+3+2)+3=12$

$f(J)=(3+1+3+3)+0=10$

Since f(J) is least, so it decides to go to node J.

Path: **A** → **F** → **G** → **I** → **J**

## A* search: *More examples*



## Performance measure of A* search

- **Completeness**

  **Yes** , in finite state spaces. No, otherwise.

- **Time complexity**

  $\mathbf{O}(\boldsymbol{b^m})$-a good heuristic can reduce this substantially up to $O(bm)$

- **Space complexity**

  $\mathbf{O}(\boldsymbol{b^m})$**-** keeps all nodes in memory

- **Optimality**

  **Yes,** for admissible heuristics