# Chapter 3: Problem Solving by Searching

Artificial Intelligence

# Introduction

- **Problems are the issues which comes across any system.**
- **A solution is needed to solve that particular problem.**
- Problem solving, particularly in artificial intelligence, may be characterized as a systematic search through a range of possible actions in order to reach some predefined goal or solution.

# Four general steps in problem solving:

- **Goal formulation**
  - Helps to organize behavior by isolating and representing the task knowledge necessary to solve problem,
  - What are the successful world states
- **Problem formulation**
  - What actions and states to consider given the goal, Define the problem precisely with initial states, final state and acceptable solutions.
- **Search**
  - Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence.
- **Execute**
  - Once the search algorithm returns a solution to the problem, the solution is then executed by the agent
  - Give the solution perform the actions.

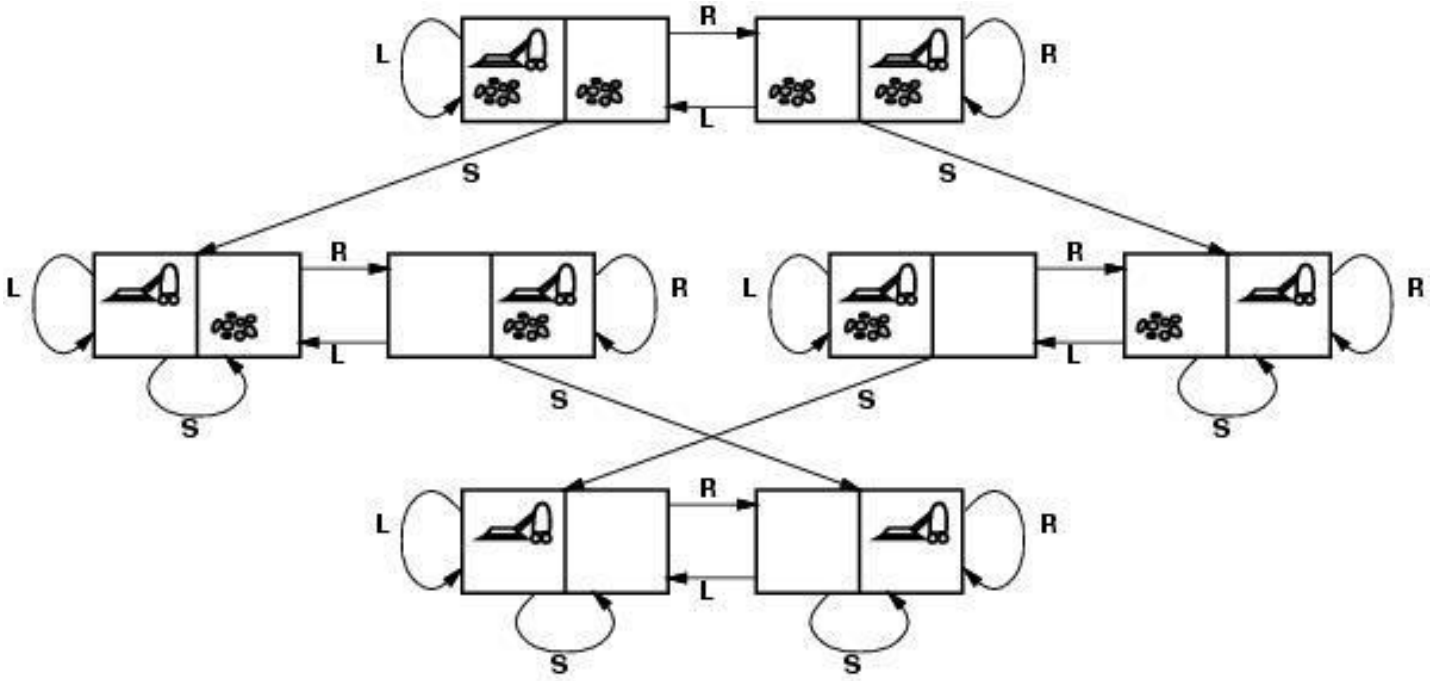
# Problem formulation

- Well-defined problems: Problems where the goal or solution is recognizable--where there is a right answer.
- A problem is defined by:
  - **An initial state**: State from which agent start
  - **Operator or Successor function**: Description of possible actions available to the agent.
  - **Goal test**: Determine whether the given state is goal state or not
  - **Path cost**: Sum of cost of each path from initial state to the given state.
- A problem when defined with these components is called ***well defined problem***.
- A solution is a sequence of actions from initial to goal state. Optimal solution has the lowest path cost.

# State Spacerepresentation

- The state space is commonly defined as a directed graph in which each node is a state and each arc represents the application of an operator transforming a state to a successor state.
- A solution is a path from the initial state to a goal state.

# State Space representation of Vacuum World Problem

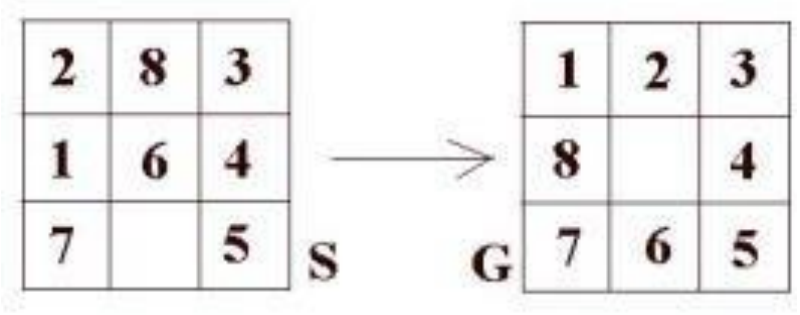


# Problem Formulation of Vacuum World Problem

- States?? two locations with or without dirt: 2 x $2^2$=8 states.
  - Eg [A,Dirty]
- Initial state?? Any state can be initial
- Actions?? {*Left*, *Right*, *Vacuum*}
- Goal test?? Check whether squares are clean.
- Path cost?? Number of actions to reach goal.

# The 8-puzzle problem

- State: location of blank
- Operator: blank moves left, right, up and down
- Goal Test: match G
- Path Cost: each step costs 1 so cost is l

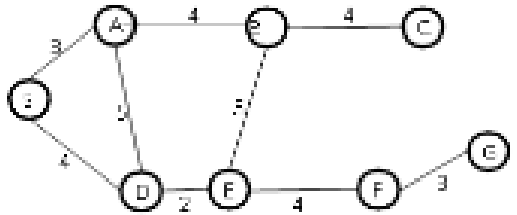


# Whatis Search?

- Search is the systematic examination of states to find path from the start/root state to the goalstate.
- The set of possible states, together with *operators* defining their connectivity constitute the *search space*.
- The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies thegoal test.
- In real life search usually results from a lack of knowledge. In AI too search is merely a offensive instrument with which to attack problems that we can't seem to solve any better way.
- Search techniques fall into three groups:
  - Methods which find *any* start - goal path,
  - Methods which find the *best* path, and finally
  - Search methods in the face of adversaries.

# Graph

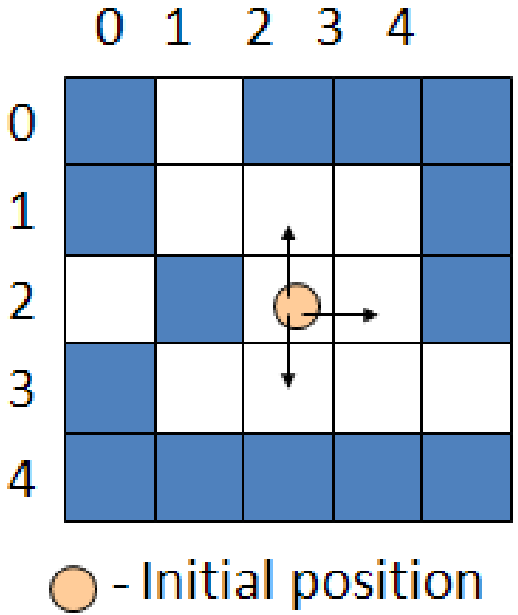
- Figure below contains a representation of a map.
- The nodes represent cities, and the links represent direct road connections between cities.
- The number associated to a link represents the length of the corresponding road.
- The search problem is to find a path from a city S to a city G
- A graph representation of a map is shown below:

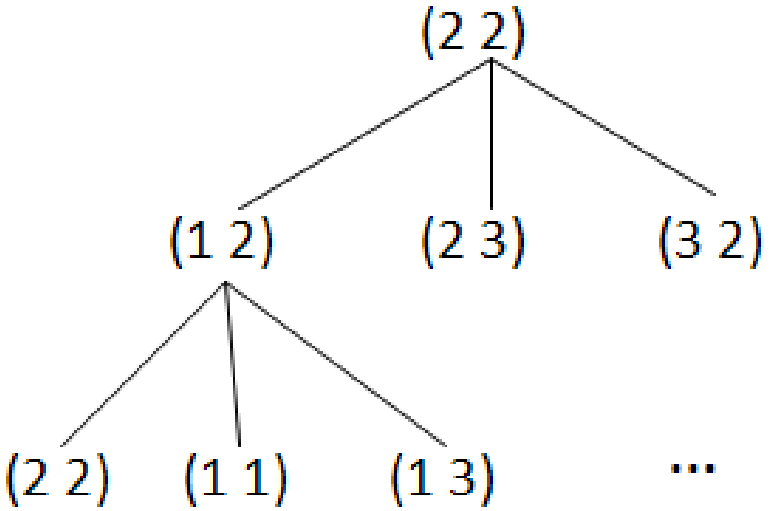


# Searchtrees

- The map of all paths within a state-space is a *graph* of *nodes* which are connected by *links*.
- We can trace out all possible paths through the graph to produce a search *tree*.
- Like graphs, trees have nodes, but they are linked by *branches*. The start node is called the *root* and nodes at the other ends are *leaves*.
- Nodes have *generations* of *descendants*. The first generation are *children*.

- These children have a single *parent* node, and the list of nodes back to the root is their *ancestry*.
- A node and its descendants form a *subtree* of the node's parent.
- If a node's subtrees are unexplored or only partially explored, the node is *open*, otherwise it is *closed*.
- If all nodes have the same number of children, this number is the *branching factor*.
- The aim of search is *not* to produce complete physical trees in memory, but rather explore as little of the virtual tree looking for root-goal paths.


0 1 2 3 4
0
1
2
3
4
- Initial position


(2 2)
(1 2)
(2 3)
(3 2)
(2 2)
(1 1)
(1 3)
...

- Some of the most popularly used problem solving with the help of artificial intelligence are:
  - Sudoku
  - Chess
  - Travelling Salesman Problem
  - Water-Jug Problem
  - N-Queen Problem
- Search problems are part of a large number of real world applications:
  - Path planning
  - Robot navigation
  - VLSI layout, etc.

- In general, searching refers to as finding information one needs.
- Searching is the most commonly used technique of problem solving in artificial intelligence.
- The searching algorithm helps us to search for solution of particular problem.

# Measuring problem Solving Performance

We will evaluate the performance of a search algorithm in four ways

- **Completeness:** An algorithm is said to be complete if it definitely finds solution to the problem, if exist.
- **Time Complexity:** How long (worst or average case) does it take to find a solution? Usually measured in terms of the **number of nodes expanded**
- **Space Complexity:** How much space is used by the algorithm? Usually measured in terms of the **maximum number of nodes in memory at a time**
- **Optimality/Admissibility:** If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

- *Time and space complexity are measured in terms of*
  - ***b*** *-- maximum branching factor (number of successor of any node) of the search tree*
  - ***d*** *-- depth of the least-cost solution*
  - ***m*** *-- maximum length of any path in the space*

# Uninformed (Blind) Search

- In the case of uninformed search methods, the order in which potential solution paths are considered is arbitrary, using no domain-specific information to judge where the solution is likely to lie.
- Various blind searches are:
  - Depth First Search
  - Breadth First Search
  - Depth Limited Search
  - Iterative Deepening Search
  - Uniform Cost Search
  - Bidirectional Search

# Informed (Heuristic) Search

- In the case of informed search methods, one uses domain-dependent (heuristic) information in order to search the space more efficiently.
- **Ways of using heuristic information:**
  - Deciding which node to expand next, instead of doing the expansion in a strictly breadth-first or depth-first order;
  - In the course of expanding a node, deciding which successor or successors to generate, instead of blindly generating all possible successors at one time;
  - Deciding that certain nodes should be discarded, or *pruned*, from the search space.
- **Why Use Heuristic Searches?**
  - It may be too resource intensive (both time and space) to use a blind search
  - Even if a blind search will work we may want a more efficient search method

- Informed Search uses domain specific information to improve the search pattern
  - Define a heuristic function, h(n), that estimates the "goodness" of a node n.
  - Specifically, **h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.**
  - The **heuristic function** is an estimate, based on domain-specific information that is computable from the current state description, of how close we are to a goal.
- Various Informed Search Techniques are:
  - Greedy Best First Search
  - A* Search
  - Hill Climbing Search
  - Simulated Annealing Search

# Game Search/ Adversarial Search

- Games are a form of *multi-agent environment*
  - What do other agents do and how do they affect our success?
  - Cooperative vs. competitive multi-agent environments.
- Competitive multi-agent environments give rise to adversarial search often known as *games*
- Games – adversary
  - Solution is strategy (strategy specifies move for every possible opponent reply).
  - Time limits force an *approximate* solution
  - Evaluation function: evaluate the goodness of game position

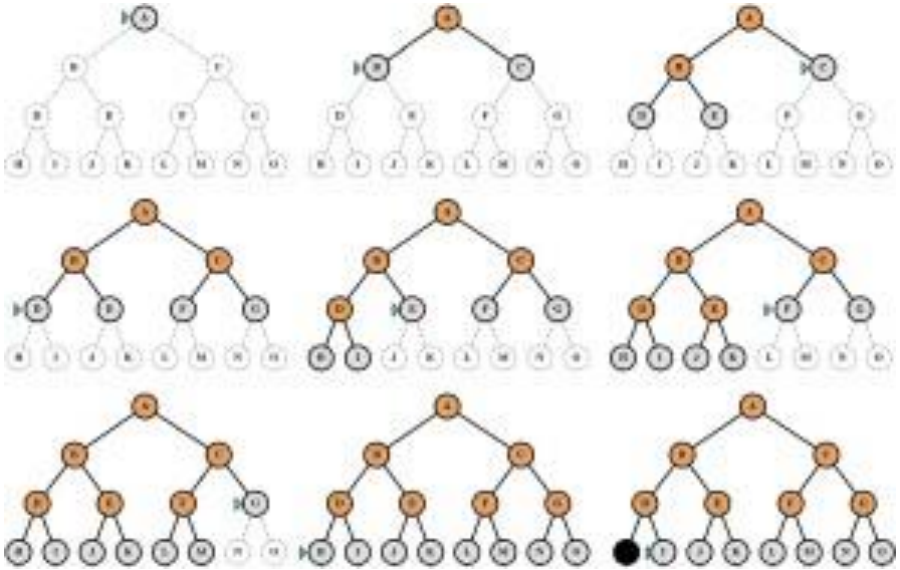
- Examples: chess, checkers, Othello, etc.
- In search space of a game it represents the moves of two (or more) players, whereas in the search space of a problem it represents the "moves" of a single problem-solving agent.
- Various Adversarial Search Techniquesare:
  - Mini-max Search
  - Alpha-Beta Pruning

# Breadth First Search

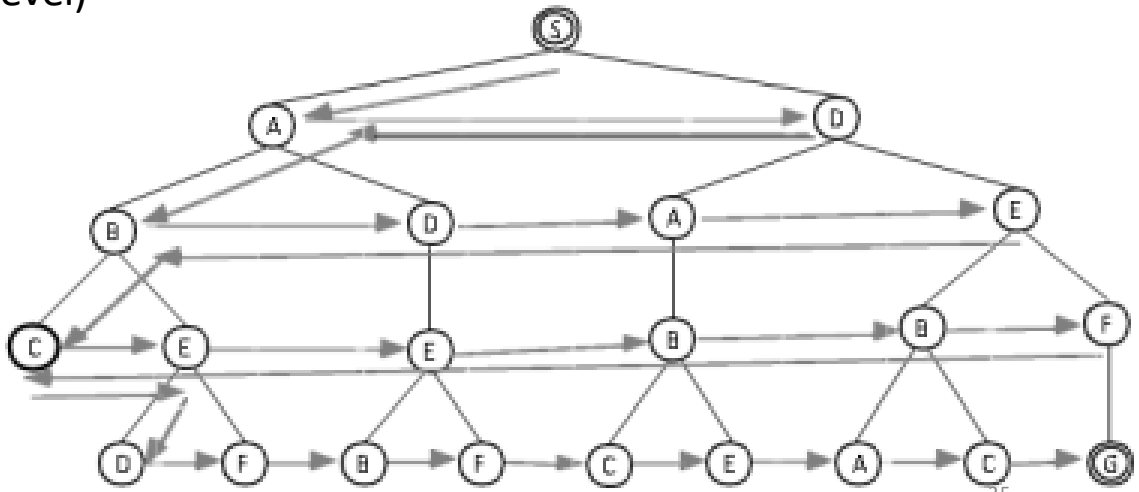
- All nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded until the goal reached.
- Expand *shallowest* unexpanded node (node with least depth).
- implemented as a FIFO queue
- In this technique you search for the goal node among all nodes of a particular generation before expanding further.
- This way you always catch the node nearest the root in generations.

# Breadth-first search (BFS)

BFS: Expand *shallowest* node first

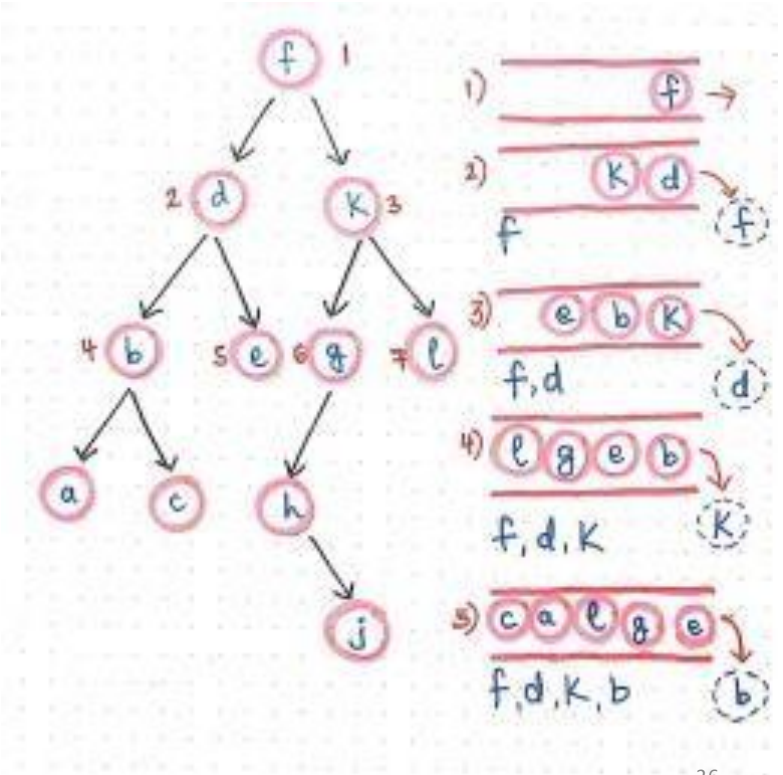


- As the name BFS suggests, you are required to traverse the graph breadthwise as follows:
  - First move horizontally and visit all the nodes of the current layer(level)
  - Move to the next layer (level)



# Breadth-first algorithm

- 1. Form a one element queue Q consisting of the root node.
- 2. Until the Q is empty or the goal has been reached, determine if the first element in the Q is the goal.
  - 2a. If it is, do nothing.
  - 2b. If is isn't, remove the first element from the Q, and add the first element's children, if any, to the BACK of the Q.
- 3. If the goal is reached, success; else failure.



# BFS Evaluation

- Completeness:
  - *Does it always find a solution if one exists?*
  - YES, if shallowest goal node is at some finite depth d and if branching factor *b* is finite
- Time complexity:
  - Assume a state space where every state has *b* successors.
  - root has *b* successors, each node at the next level has again b successors (total $b^2$), …
  - Assume solution is at depth *d*
  - Worst case; expand all except the last node at depth *d*
  - Total no. of nodes generated:
    - $b + b^2 + b^3 + \ldots\ldots\ldots\ldots\ldots\ldots . . \; b^d + ( b^{d+1} - b) = \mathbf{O(b^{d+1})}$

- Space complexity:
  - Each node that is generated must remain in memory
  - Total no. of nodes in memory:
    - $1+ b + b^2 + b^3 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots b^d + ( b^{d+1} - b) = \mathbf{O(b^{d+1})}$
- Optimal (i.e., admissible):
  - Optimal only if all paths have the same cost.
  - Otherwise, not optimal but finds solution with shortest path length (shallowest solution).
  - If each path does not have same path cost shallowest solution may not be optimal

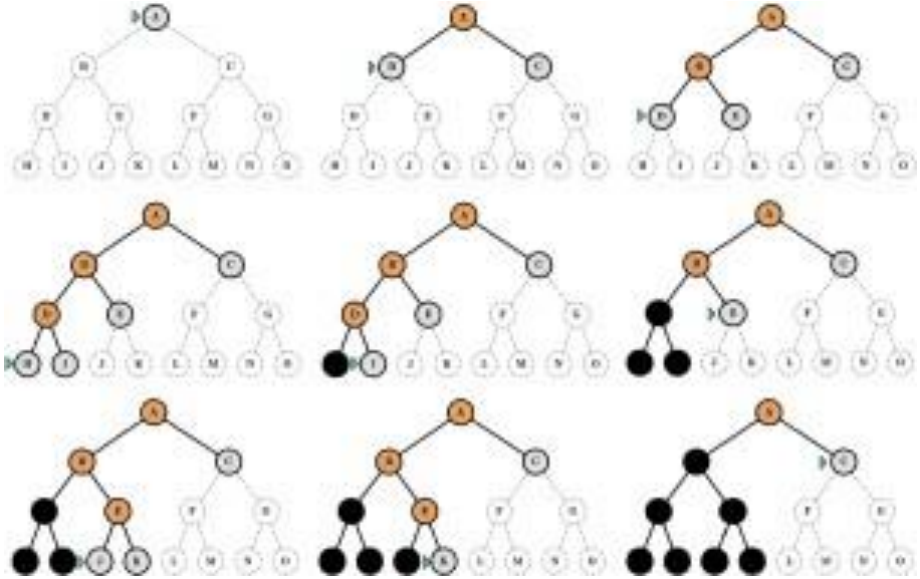
- Two lessons:
  - Memory requirements are a bigger problem than its execution time.
  - Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances
- Performance Measures
  - Completeness: Yes
  - Time efficiency: No
  - Space efficiency: No
  - Optimality: No

# Depth First Search

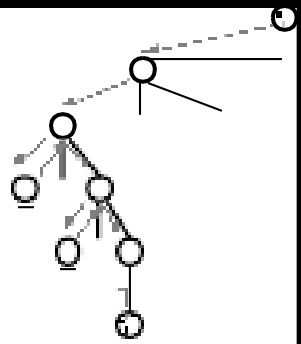
- Looks for the goal node among all the children of the current node before using the sibling of this node i.e. **expand *deepest* unexpanded node.**
- implemented as a LIFO queue (=stack)
- The DFS algorithm is a recursive algorithm that uses the idea of backtracking.
- It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.
  - Here, the word backtracking means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse.
- All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

# Depth-first Search (DFS)

DFS: Expand *deepest* node first



# DFS Evaluation



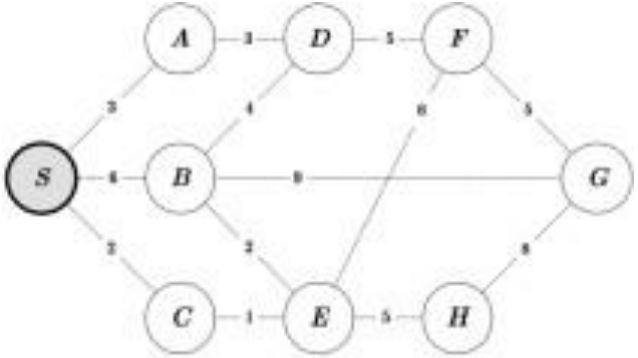
- Completeness: *Does it always find a solution if one exists?*
  - NO, if search space is infinite and search space contains loops then DFS may not find solution.
- Time complexity;
  - Let m is the maximum depth of the search tree.
  - In the worst case, Solution may exist at depth m.
  - root has *b* successors, each node at the next level has again b successors (total $b^2$), …
  - Worst case; expand all except the last node at depth *m*
  - Total no. of nodes generated:
    - $b + b^2 + b^3 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots . .\ b^m = \mathbf{O(b^m)}$

- Space complexity:
  - It needs to store only a single path from the root node to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.
  - Total no. of nodes in memory:
    - 1+b +b +b +…….. b m times = **O(bm)**
- Optimal (i.e., admissible):
  - DFS expand deepest node first, if expands entire left sub-tree even if right sub-tree contains goal nodes at levels 2 or 3. Thus we can say DFS may not always give optimal solution.

# In short:

- Depth-first search is simple, but is very unsatisfactory if the tree is unbalanced, with some leaves being at the end of much longer branches than others.
  - Completeness: No
  - Time efficiency: No
  - Space efficiency: Yes
  - Optimality: No

# Exercise: BFS

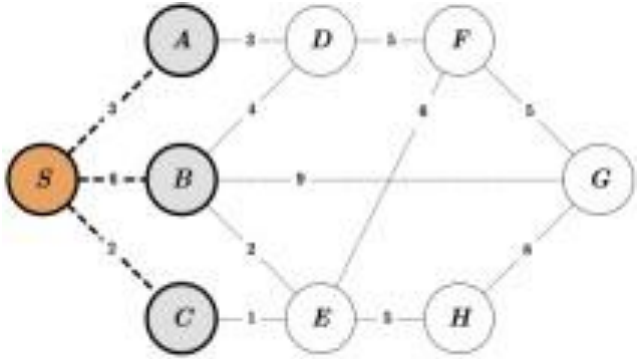



**Queue:**

| S |
|---|

**Order of Visit:**

# Exercise: BFS



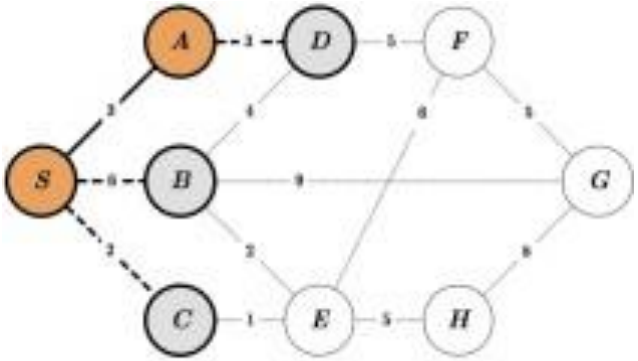

**Queue:**

| S | A | B | C |
|---|---|---|---|

**Order of Visit:**

*S*

# Exercise: BFS



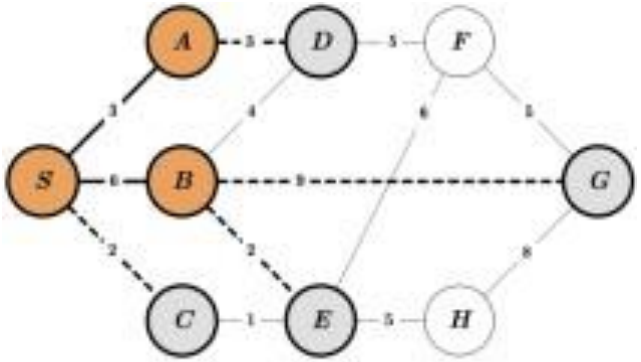

**Queue:**

| S | A | B | C | D |
|---|---|---|---|---|

**Order of Visit:**

*S* *A*

# Exercise: BFS



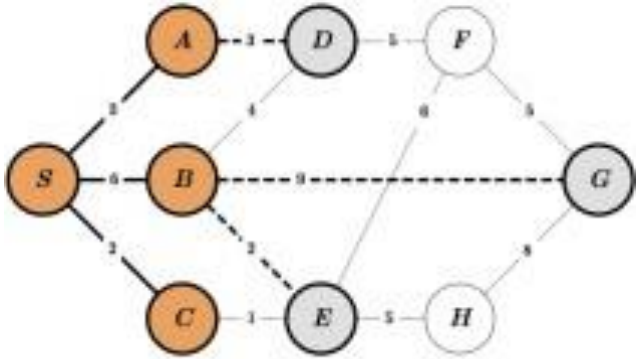

Queue:

| S | A | B | C | D | E | G |
|---|---|---|---|---|---|---|

Order of Visit:

S A B

# Exercise: BFS



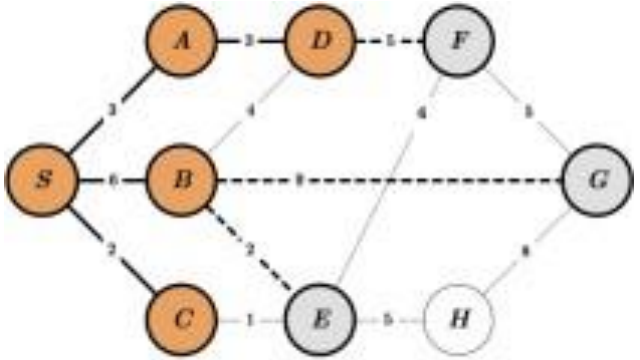

Queue:

| S | A | B | C | D | E | G |
|---|---|---|---|---|---|---|

Order of Visit:

S A B C

# Exercise: BFS



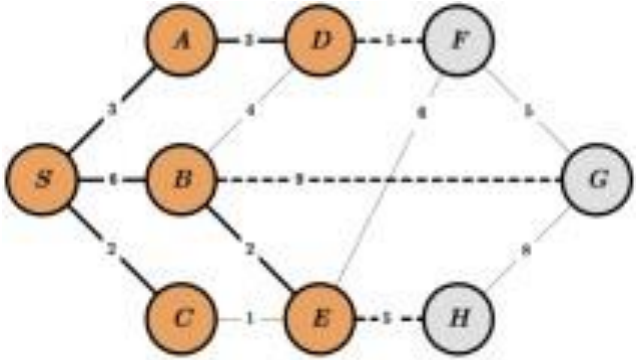

**Queue:**

| S | A | B | C | D | E | G | F |
|---|---|---|---|---|---|---|---|

**Order of Visit:**

*S* *A* *B* *C* *D*

# Exercise: BFS



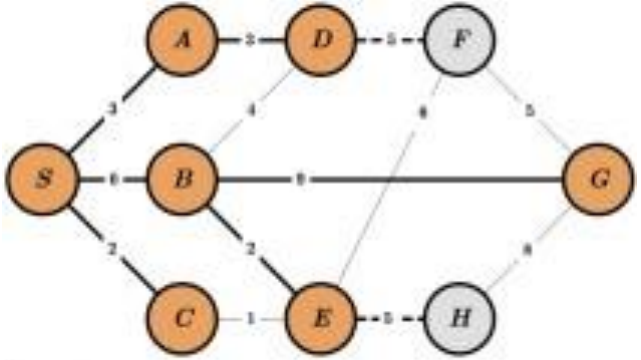

Queue:

| S | A | B | C | D | E | G | F | H |
|---|---|---|---|---|---|---|---|---|

Order of Visit:

S A B C D E

# Exercise: BFS



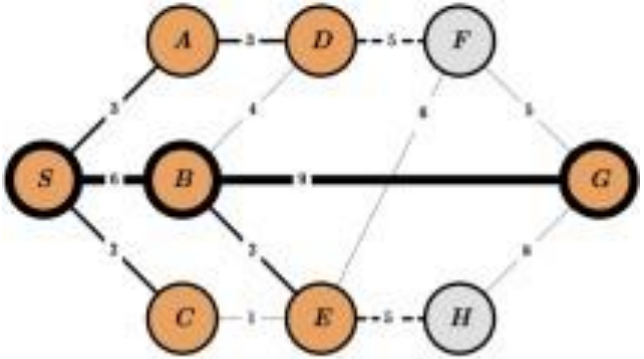

**Queue:**

| S | A | B | C | D | E | G | F | H |
|---|---|---|---|---|---|---|---|---|

**Order of Visit:**

*S* *A* *B* *C* *D* *E* *G*

# Exercise: BFS



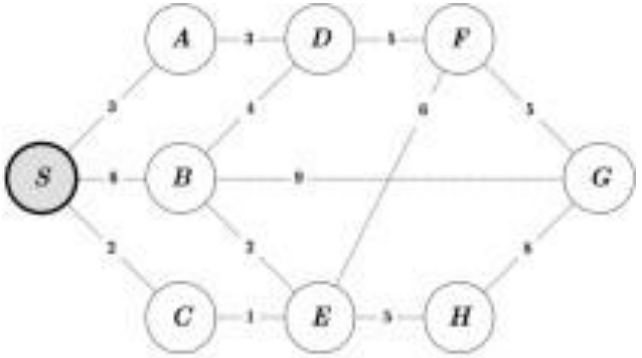

**Queue:**

| S | A | B | C | D | E | G | F | H |
|---|---|---|---|---|---|---|---|---|

**Order of Visit:**

*S* *A* *B* *C* *D* *E* *G*

# Exercise: DFS

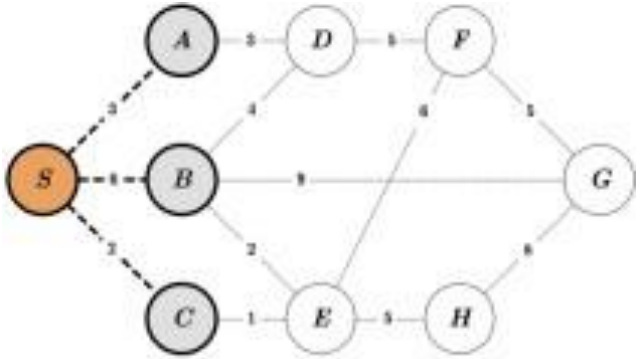



**Stack:**

| |
|---|
| *S* |

**Order of Visit:**

# Exercise: DFS



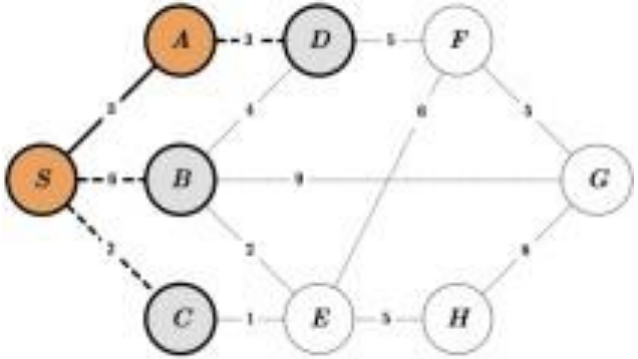

**Stack:**

| | | | |
|---|---|---|---|
| *S* | *C* | *B* | *A* |

**Order of Visit:**

*S*

# Exercise: DFS



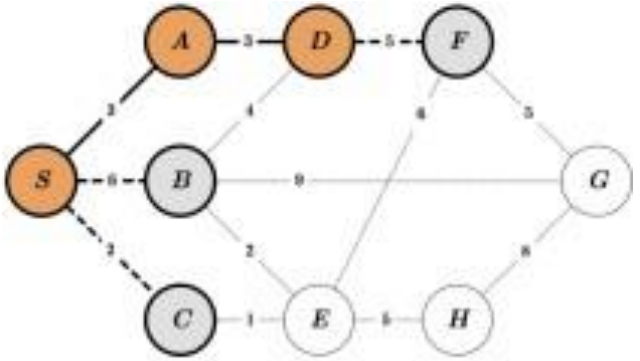

**Stack:**

| S | C | B | A | D |
|---|---|---|---|---|

**Order of Visit:**

*S* *A*

# Exercise: DFS



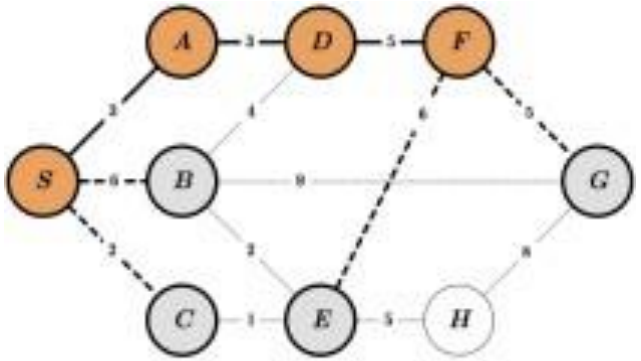

**Stack:**

| S | C | B | A | D | F |
|---|---|---|---|---|---|

**Order of Visit:**

*S* *A* *D*

# Exercise: DFS



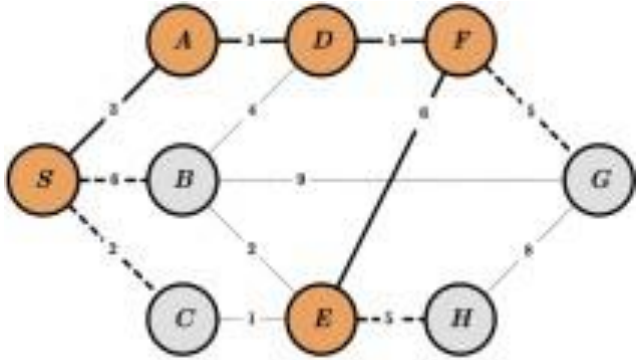

**Stack:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* | *G* | *E* |

**Order of Visit:**

*S* *A* *D* *F*

# Exercise: DFS



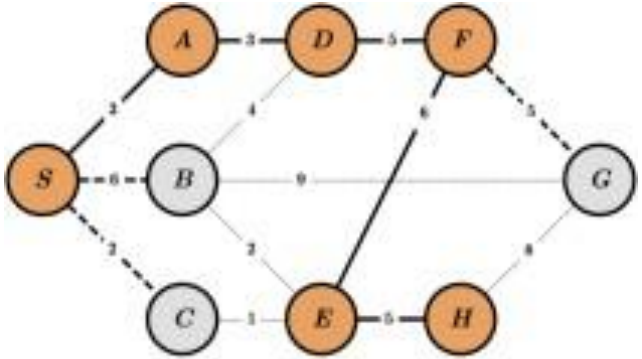

**Stack:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* | *G* | *E* | *H* |

**Order of Visit:**

*S* *A* *D* *F* *E*

# Exercise: DFS



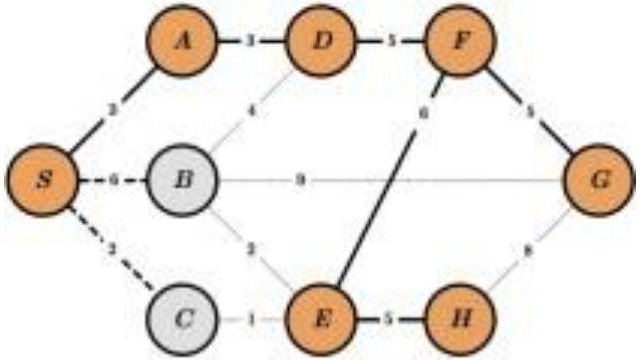

**Stack:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* | *G* | *E* | *H* |

**Order of Visit:**

*S* *A* *D* *F* *E* *H*

# Exercise: DFS



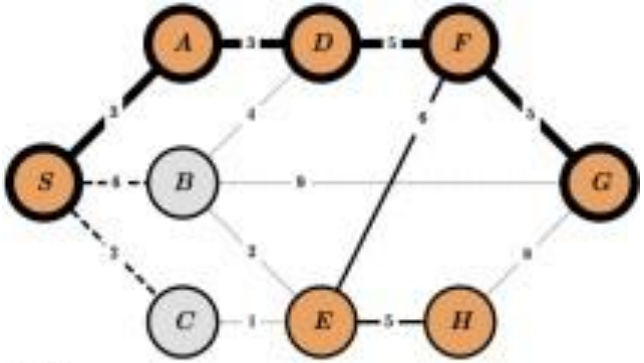

**Stack:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* | *G* | *E* | *H* |

**Order of Visit:**

***S*** ***A*** ***D*** ***F*** ***E*** ***H*** ***G***

# Exercise: DFS




**Stack:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* | *G* | *E* | *H* |

**Order of Visit:**

*S* *A* *D* *F* *E* *H* *G*

# Depth Limited Search

- The problem of unbounded trees can be solved by supplying depth-first search with a determined depth limit
- nodes at certain depth are treated as they have no successors
- is an uninformed search
- modification of depth-first search and is used for example in the iterative deepening depth-first search algorithm.
- It works exactly like depth-first search, but avoids its drawbacks regarding completeness by imposing a maximum limit on the depth of the search.

- Even if the search could still expand a vertex beyond that depth, it will not do so and thereby it will not follow infinitely deep paths or get stuck in cycles.
- Therefore depth-limited search will find a solution if it is within the depth limit, which guarantees at least completeness on all graphs.
- It solves the infinite-path problem of DFS.

- Yet it introduces another source of problem if we are unable to find good **guess of maximum *level of depth l*.**
- Let **d is the depth of shallowest solution (depth of goal node)**.
  - If ***l < d*** then incompleteness results.
  - If ***l > d*** then not optimal.
- Time complexity: **$O(b^l)$**
- Space complexity: **$O(bl)$**

# Iterative Deepening Depth First Search

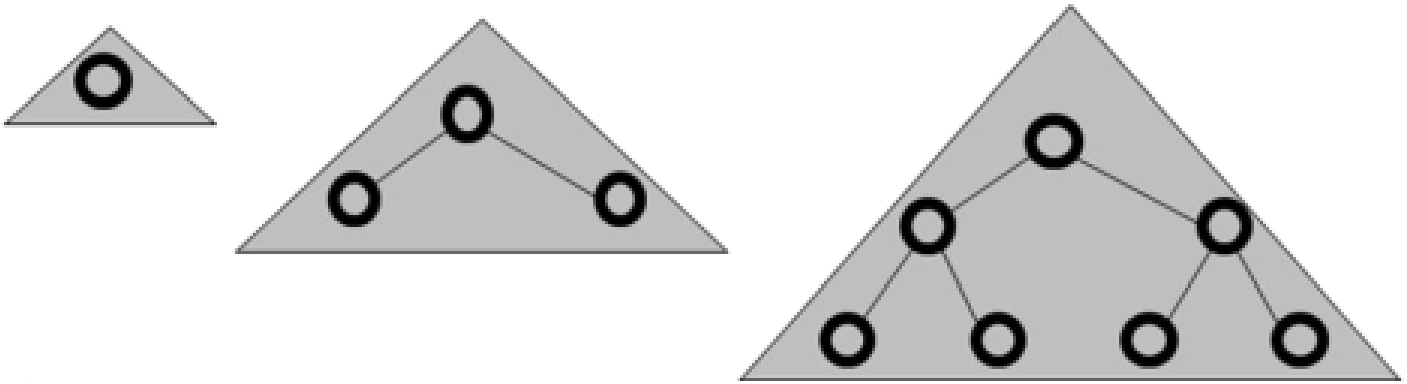
- In this strategy, depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches *d*, the depth of the shallowest goal state.
- On each iteration, IDDFS visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first.
- IDDFS combines depth-first search's space-efficiency and breadth-first search's completeness (when the branching factor is finite).
- It is optimal when the path cost is a non-decreasing function of the depth of the node.

- *Iterative deepening* is depth-first search to a fixed depth in the tree being searched.
- If no solution is found up to this depth then the depth to be searched is increased and the whole `bounded' depth-first search begun again.
- It works by setting a depth of search -say, depth 1- and doing depth-first search to that depth.
- If a solution is found then the process stops -otherwise, increase the depth by, say, 1 and repeat until a solution is found.
  - *Note that every time we start up a new bounded depth search we start from scratch - i.e. we throw away any results from the previoussearch.*

- Depth-first search can be implemented to be much cheaper than breadth-first search in terms of memory usage
  - However, depth-first search is not guaranteed to find a solution even where one is guaranteed.
- On the other hand, breadth-first search can be guaranteed to terminate if there is a winning state to be found and will always find the `quickest' solution (in terms of how many steps need to be taken from the root node).
  - It is, however, breadth-first search is very expensive method in terms of memory usage.
- *Iterative deepening* is liked because it is an effective compromise between the two other methods of search.

- It is a form of depth-first search with a lower bound on how deep the search can go.
  - Iterative deepening terminates if there is a solution.
  - It can produce the same solution that breadth-first search would produce but does not require the same memory usage (as for breadth-first search).
- Note that depth-first search achieves its efficiency by generating the next node to explore only when this needed.
  - The breadth-first search algorithm has to grow all the search paths available until a solution is found -and this takes up memory.
  - Iterative deepening achieves its memory saving in the same way that depth-first search does -at the expense of redoing some computations again and again (a time cost rather than a memory one).

- Complete (like BFS)
- Has linear memory requirements (like DFS)
- This is the preferred method for large state spaces, where the solution path length is unknown.



# Iterative Deepening search evaluation

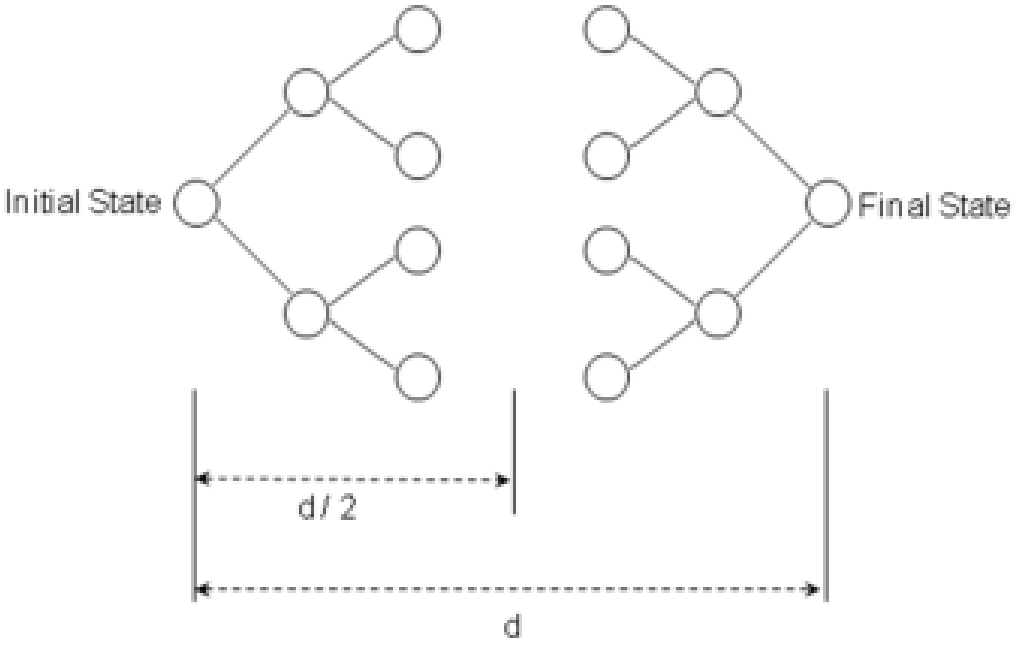
- Completeness:
  - YES (no infinite paths)
- Time complexity:
  - Algorithm seems costly due to repeated generation of certain states.
  - Total no. of nodes generated: **$O(b^d)$**

- Space complexity:
  - It needs to store only a single path from the root node to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.
  - Total no. of nodes in memory:
    - 1+b +b +b +…… b d times =**O(bd)**
- Optimality:
  - YES if path cost is non-decreasing function of the depth of the node.
- *Notice that BFS generates some nodes at depth d+1, whereas IDS does not. The result is that IDS is actually faster than BFS, despite the repeated generation of node.*

# Bidirectional Search

- This is a search algorithm which replaces a single search graph, which is likely to with two smaller graphs –
  - One starting from the initial state and
  - Another one starting from the goal state
- It then, expands nodes from the start and goal state simultaneously.
- Check at each stage if the nodes of one have been generated by the other, i.e, they meet in the middle.
- If so, the path concatenation is the solution.

- Completeness: YES
- Optimality: YES (If done with correct strategy- e.g. breadth first and paths have the same cost)
- Time complexity: $O(b^{d/2})$
- Space complexity: $O(b^{d/2})$



# Drawbacks of uniformed search :

- Criterion to choose next node to expand depends only on a global criterion: level or depth.
- One may prefer to use a more flexible rule, that takes advantage of what is being discovered on the way, and hunches about what can be a good move.
- Very often, we can select which rule to apply by comparing the current state and the desired state but uninformed search does not provide this feature.

# Heuristic Search:

- Heuristic Search Uses domain-dependent (heuristic) information in order to search the space more efficiently.
- *Ways of using heuristic information:*
  - Deciding which node to expand next, instead of doing the expansion in a strictly breadth-first or depth-first order;
  - In the course of expanding a node, deciding which successor or successors to generate, instead of blindly generating all possible successors at one time;
  - Deciding that certain nodes should be discarded, or *pruned*, from the search space.

# Heuristic Searches - Why Use?

- It may be too resource intensive (both time and space) to use a blind search
- Even if a blind search will work we may want a more efficient search method
- Informed Search uses domain specific information to improve the search pattern
  - Define a heuristic function, h(n), that estimates the "goodness" of a node n.
  - Specifically, h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.
- The heuristic function is an estimate, based on domain-specific information that is computable from the current state description, of how close we are to a goal.

# Best-First Search

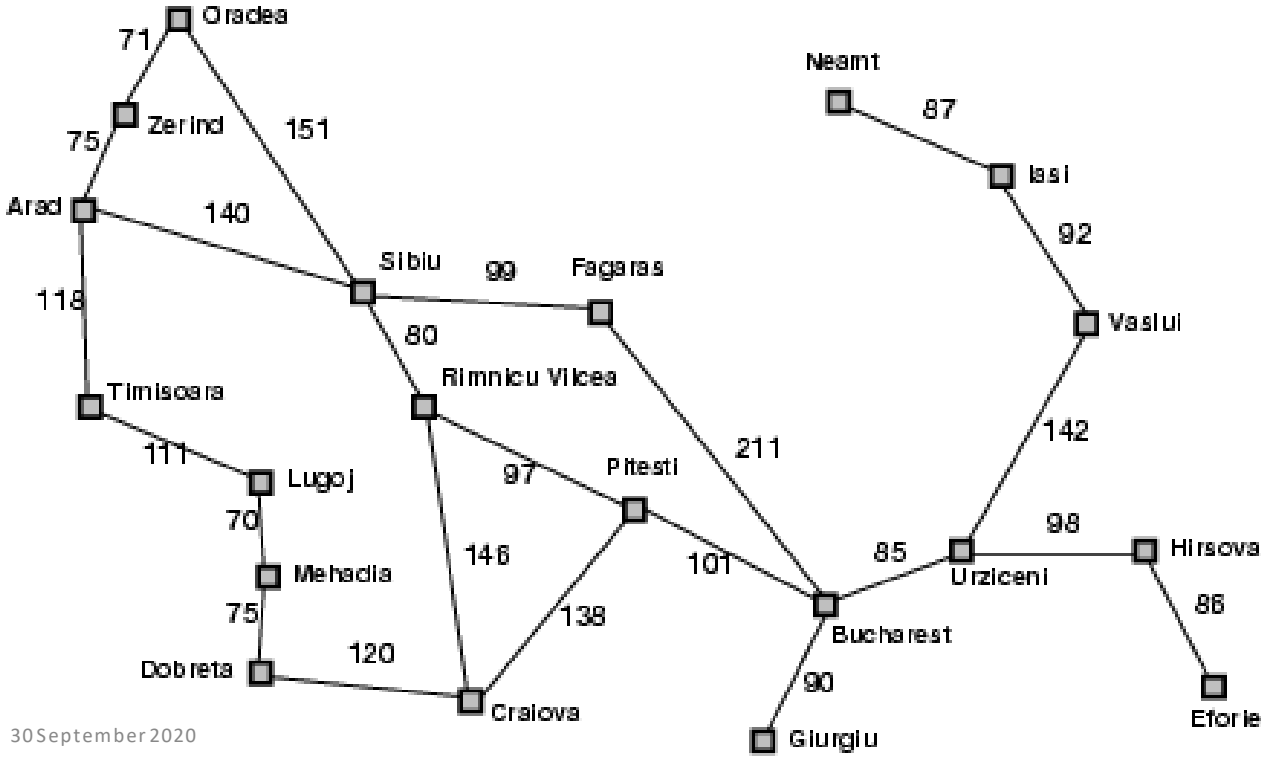
- **Idea:** use an *evaluation function f(n)* that gives an indication of which node to expand next for each node.
  - usually gives an estimate to the goal.
  - the node with the lowest value is expanded first.
- A key component of *f(n*) is a heuristic function, *h(n),* which is a additional knowledge of the problem.
  - There is a whole family of best-first search strategies, each with a different evaluation function.
- Typically, strategies use estimates of the cost of reaching the goal and try to minimize it.
- Special cases: based on the evaluation function.
  - Greedy best-first search
  - A*search

# Greedy Best First Search

- The best-first search part of the name means that it uses an evaluation function to select which node is to be expanded next.
- The node with the lowest evaluation is selected for expansion because that is the *best* node, since it supposedly has the closest path to the goal (if the heuristic is good).
- Evaluation function $f(n) = h(n)$ (heuristic) = estimate of cost from $n$ to *goal.*
  - e.g., $h_{SLD}(n)$ = straight-line distance from $n$ to goal
- Greedy best-first search expands the node that appears to be closest to goal.

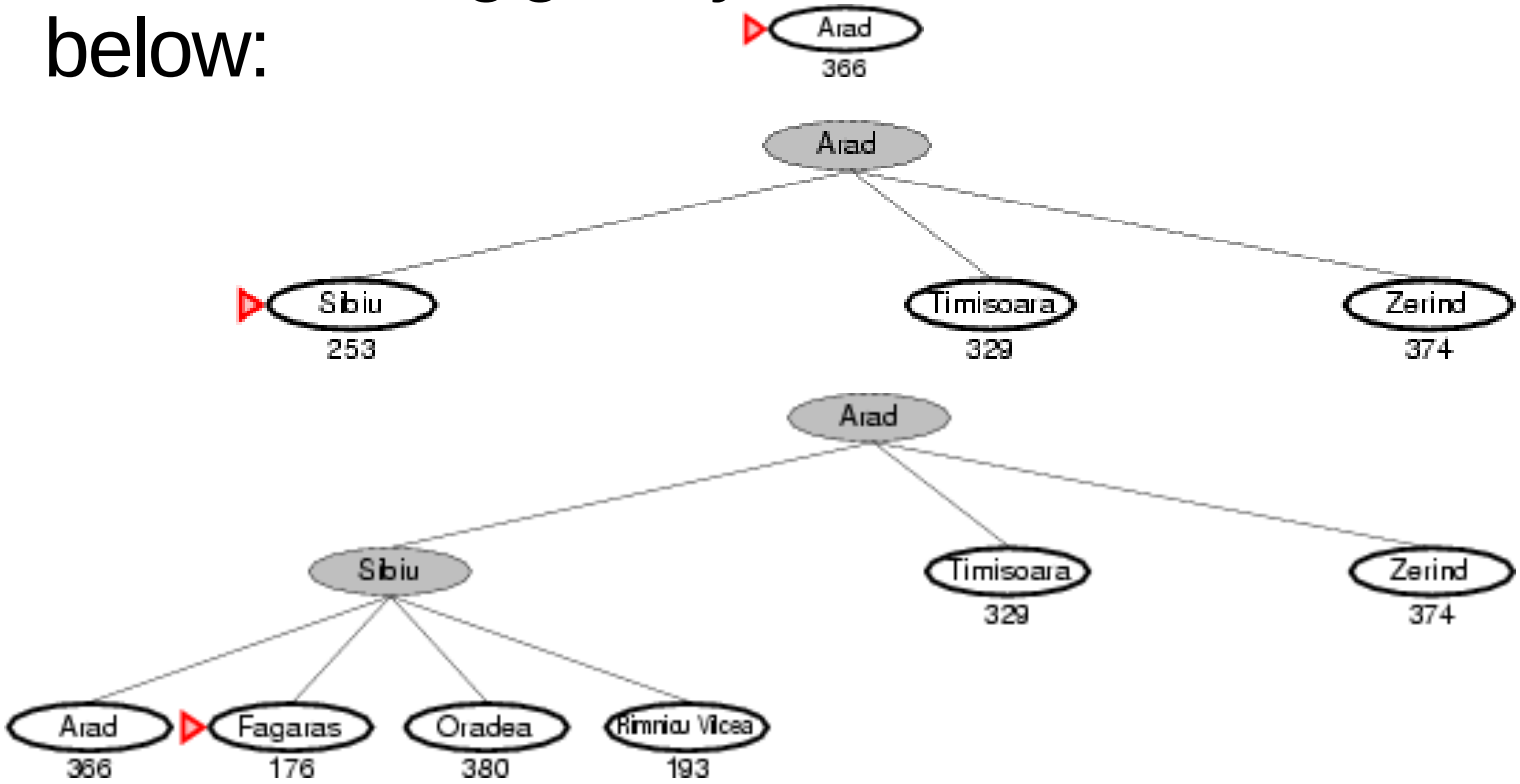
- The greedy best-first search algorithm is $O(b^m)$ in terms of space and time complexity.
  - *b* is the average branching factor (the average number of successors from a state), and
  - *m* is the maximum depth of the search tree.)
- Unlike A* which uses both the link costs and a heuristic of the cost to the goal, greedy best-first search uses only the heuristic, and not any link costs.
- A disadvantage of this approach is that if the heuristic is not accurate, it can go down paths with high link cost since there might be a low heuristic for the connecting node.

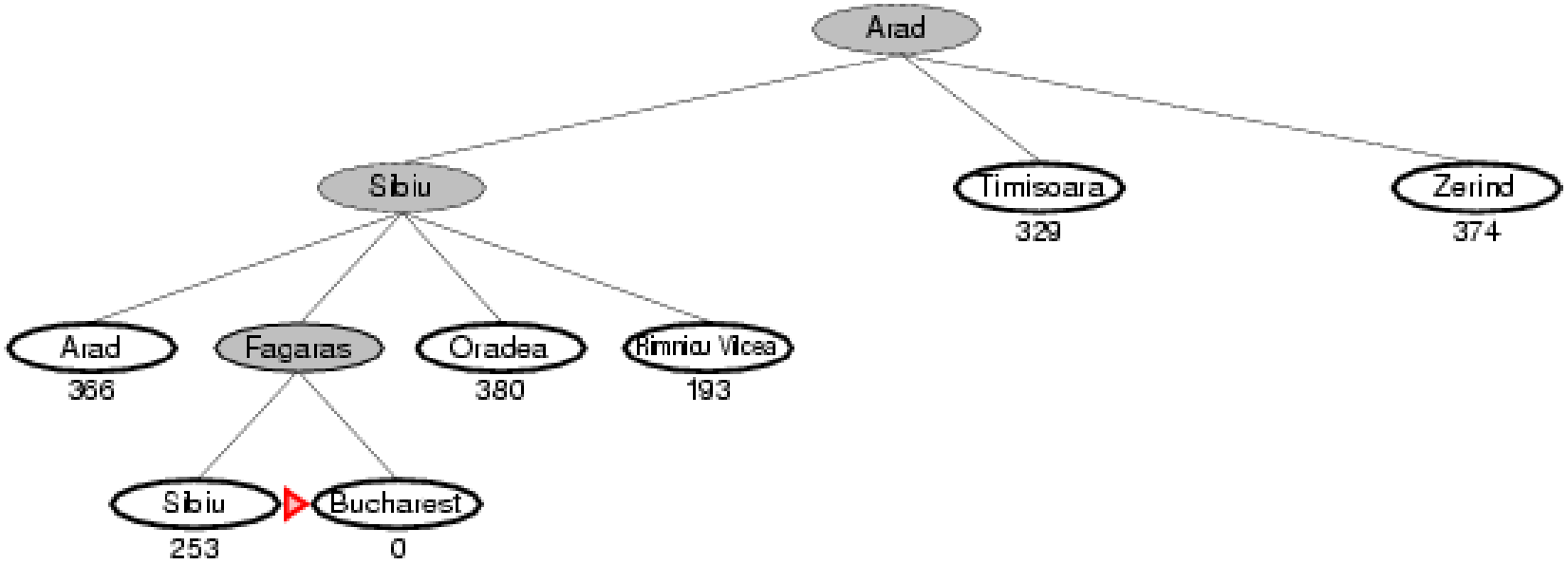
# Example: Given following graph of cities, starting at Arad city, problem is to reach to the Bucharest.




| 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 |

# Solution using greedy best first can be as below:



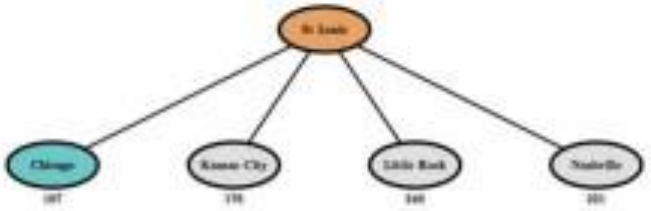

Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
Oradea
380
Rimnicu Vilcea
193
Sibiu
253
Bucharest
0

# Greedy Search Example

The initial state:

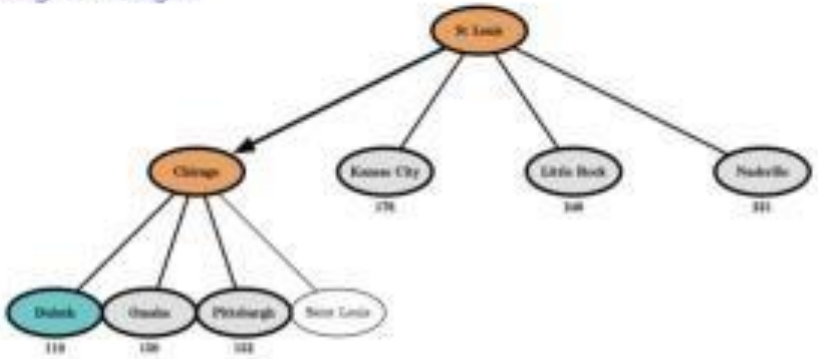
# Greedy Search Example

After expanding St Louis:



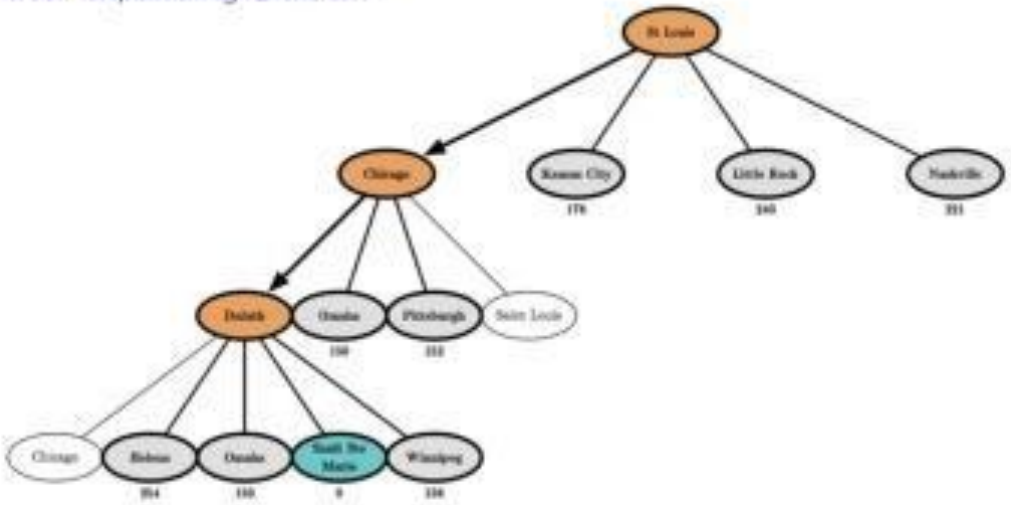
# Greedy Search Example

After expanding Chicago:



# Greedy Search Example

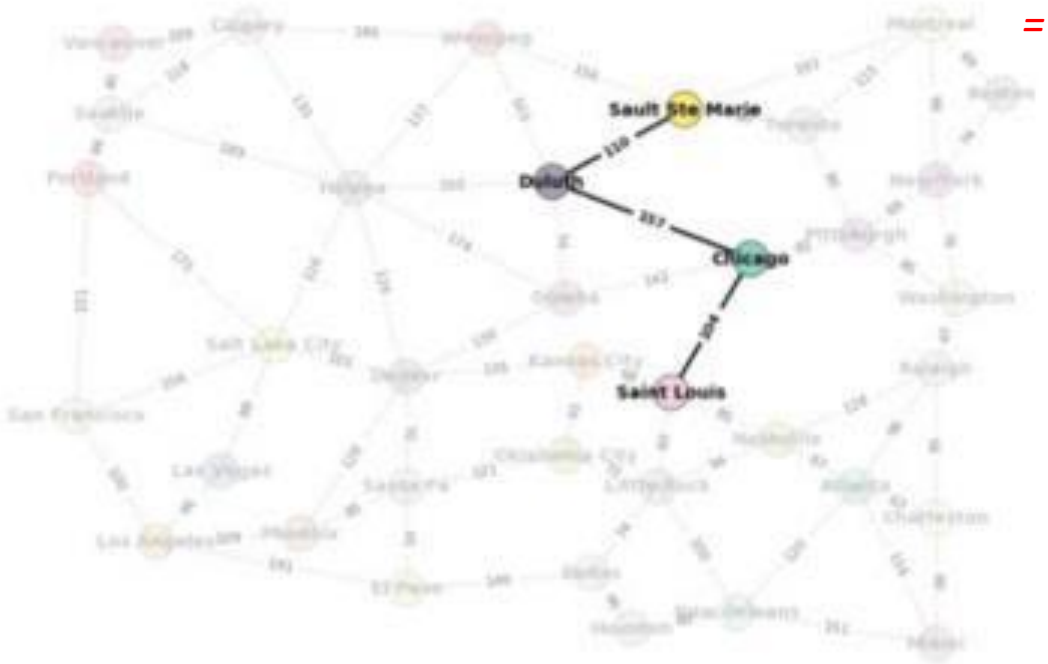
After expanding Duluth:



Start: Saint Louis

Goal: Sault Ste Marie

*total cost:*
*104 + 157 + 110*
*= 371 kms*




Greedy search

- **Greedy Best-first search**
  - minimizes estimated cost h(n) from current node n to goal;
  - is informed but (almost always) suboptimal and incomplete.

# A* search

- A* is a best first, informed graph search algorithm.
- A* is different from other best first search algorithms in that it uses a heuristic function h(x) as well as the path cost to the node g(x), in computing the cost f(x) = h(x) + g(x) for the node.
- The *h*(*x*) part of the *f*(*x*) function must be an admissible heuristic; that is, it must not overestimate the distance to the goal.
  - Thus, for an application like routing, *h*(*x*) might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.

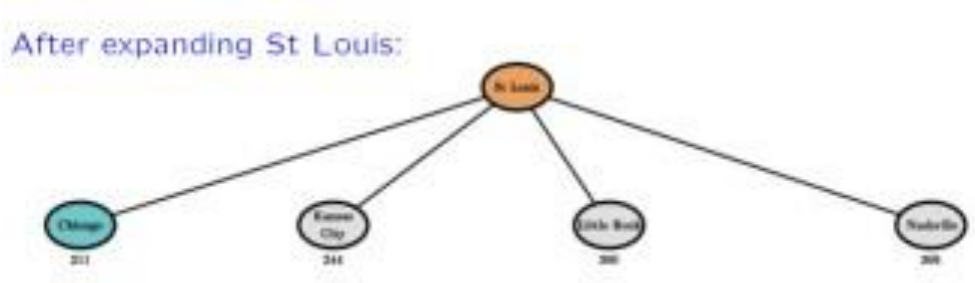
- **It finds a minimal cost-path joining the start node and a goal node for node n.**
- **Evaluation function: f(n) = g(n) + h(n)**
  - g(n) = cost so far to reach n from root
  - h(n) = estimated cost to goal from n
  - f(n) = estimated total cost of path through n to goal
- combines the two by minimizing f(n) = g(n) + h(n);
- is informed and, *under reasonable assumptions*, optimal and complete.

- As A* traverses the graph, it follows a path of the lowest *known* path, keeping a sorted priority queue of alternate path segments along the way.
  - If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead.
- This process continues until the goal is reached

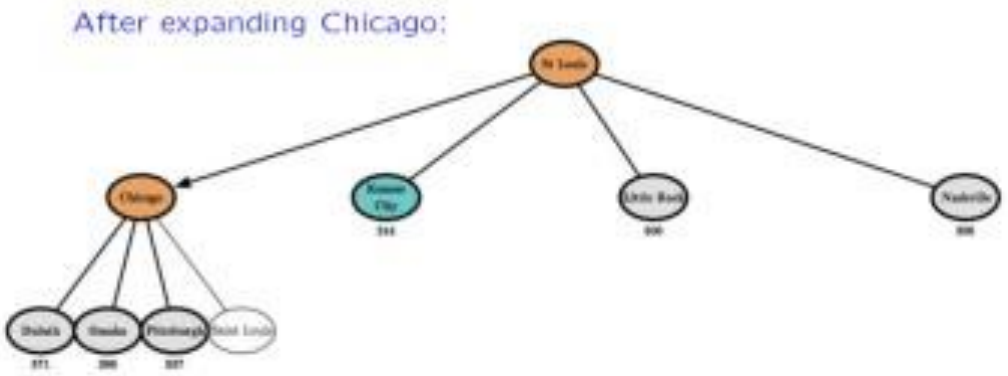
# A* search example

The initial state:

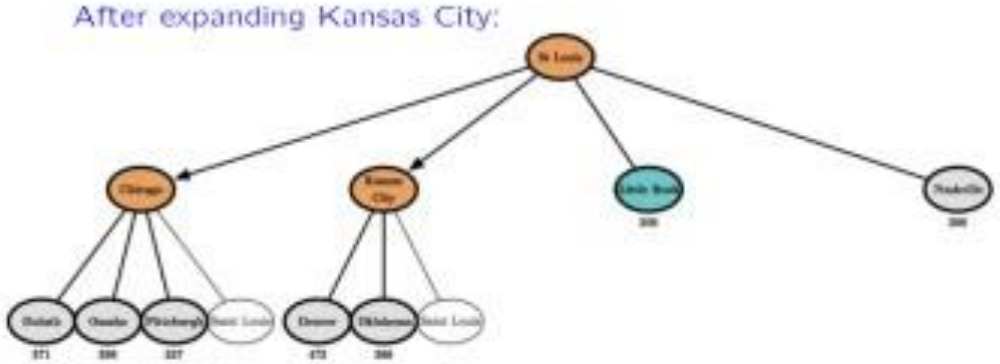
# A* search example



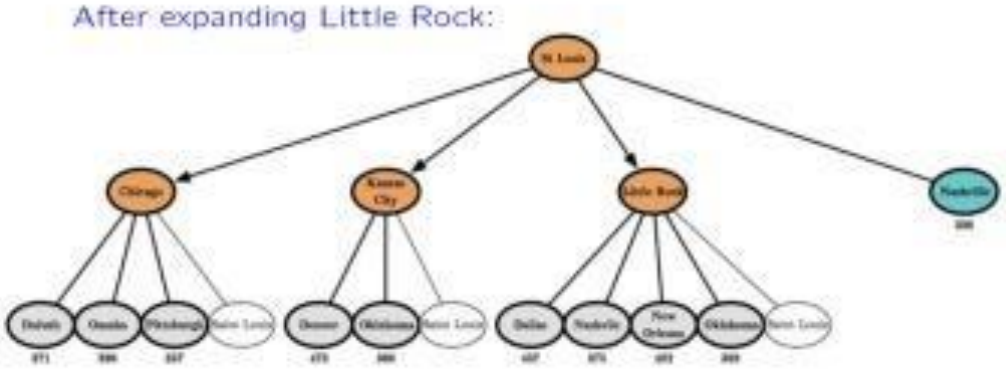
# A* search example



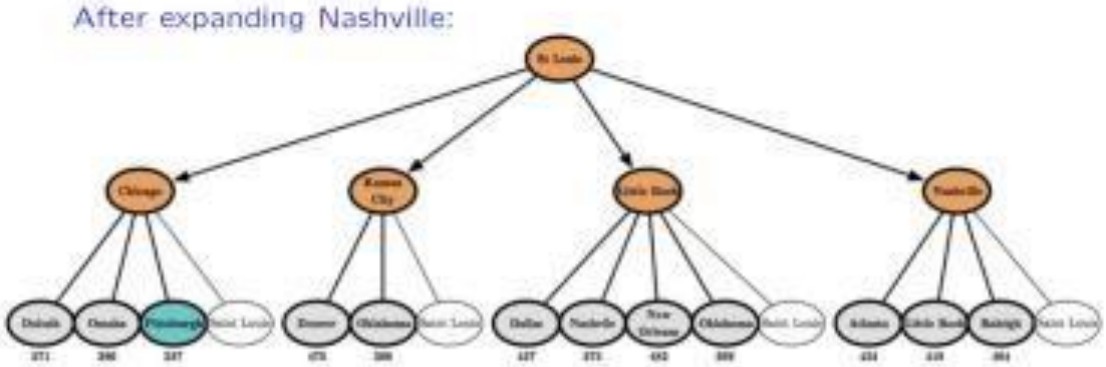
# A* search example



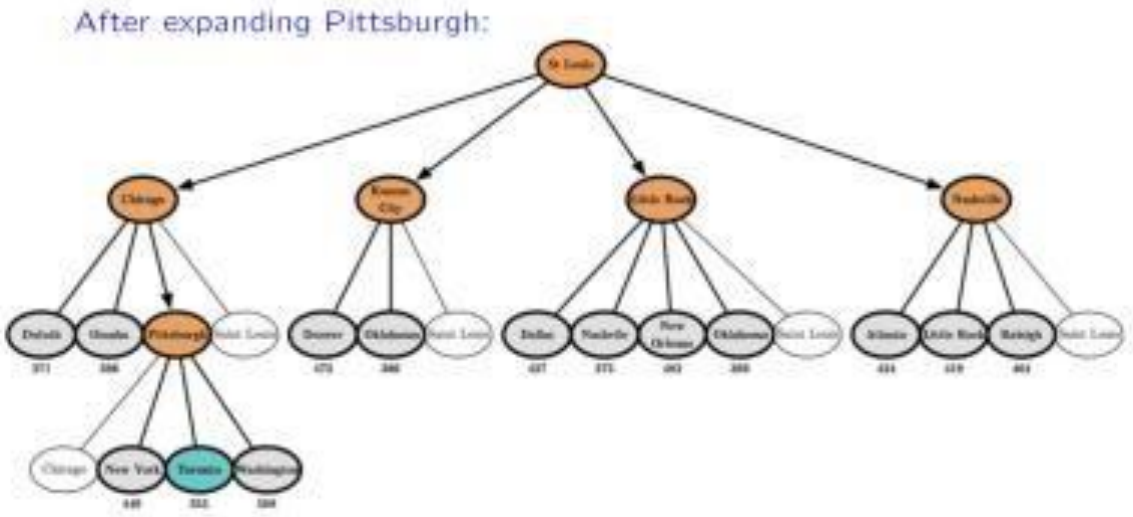
# A* search example



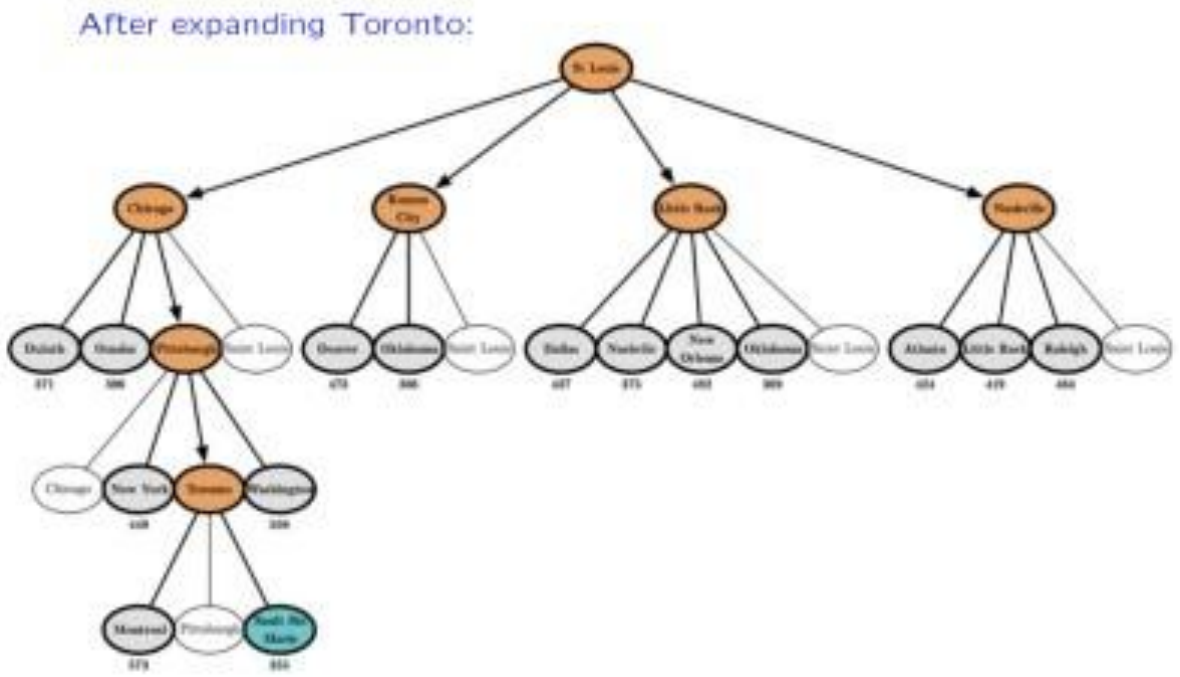
# A* search example



# A* search example

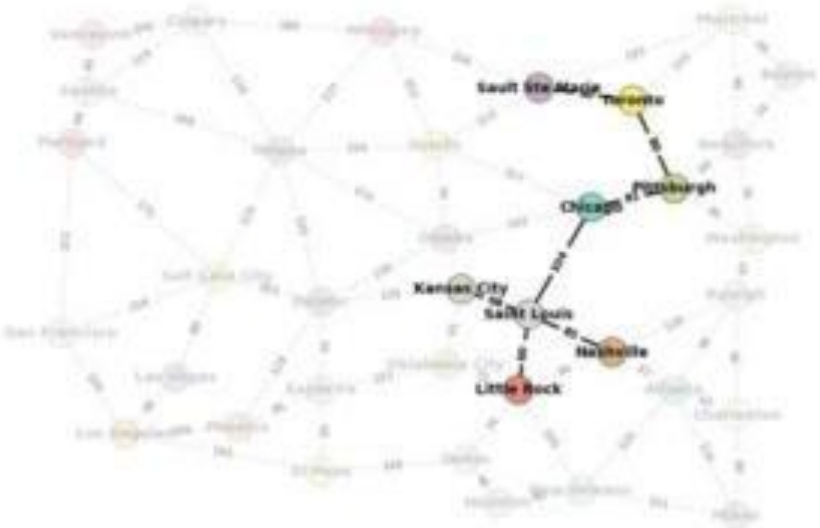


# A* search example



# Example using map

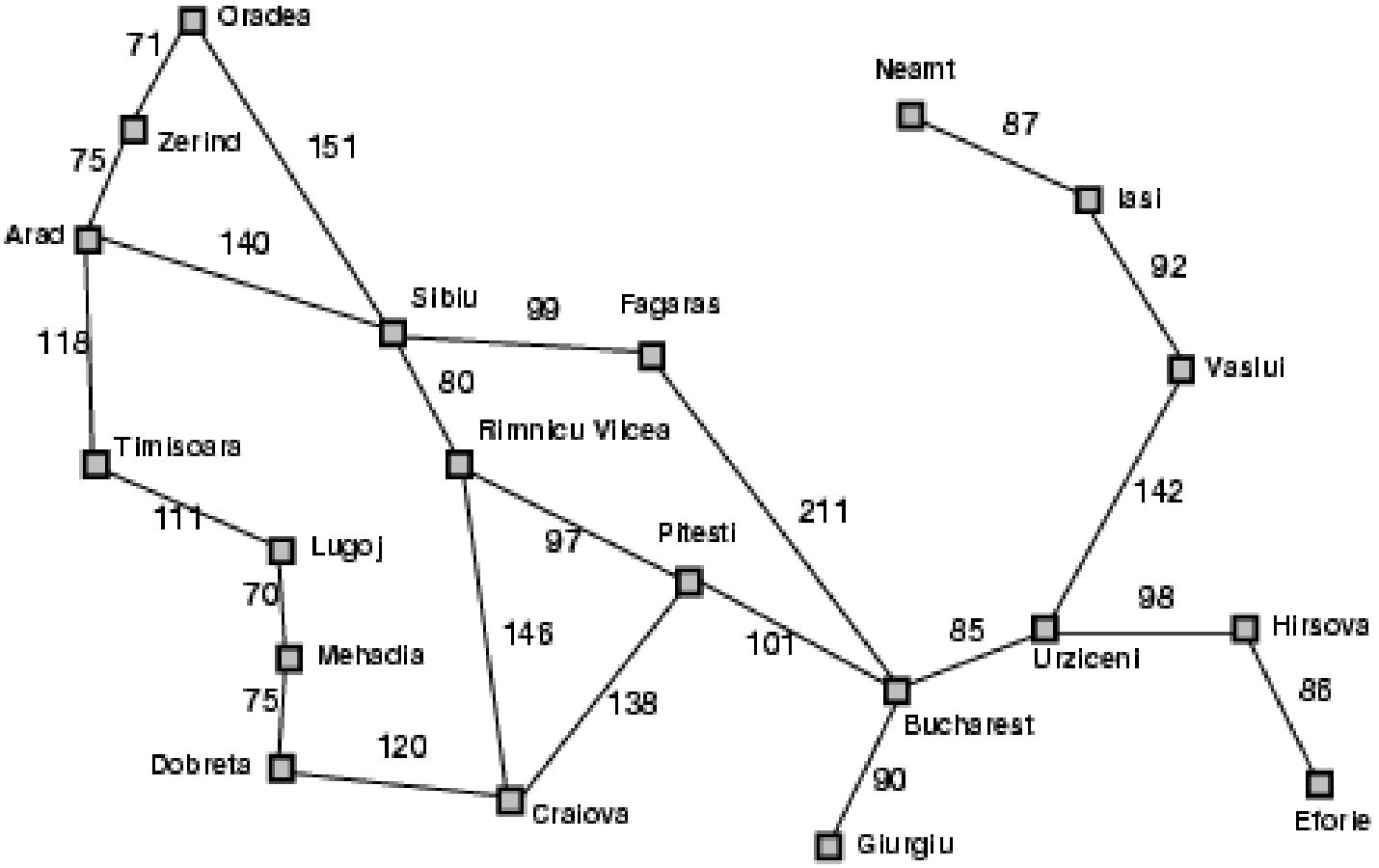
Start: Saint Louis
Goal: Sault Ste Marie



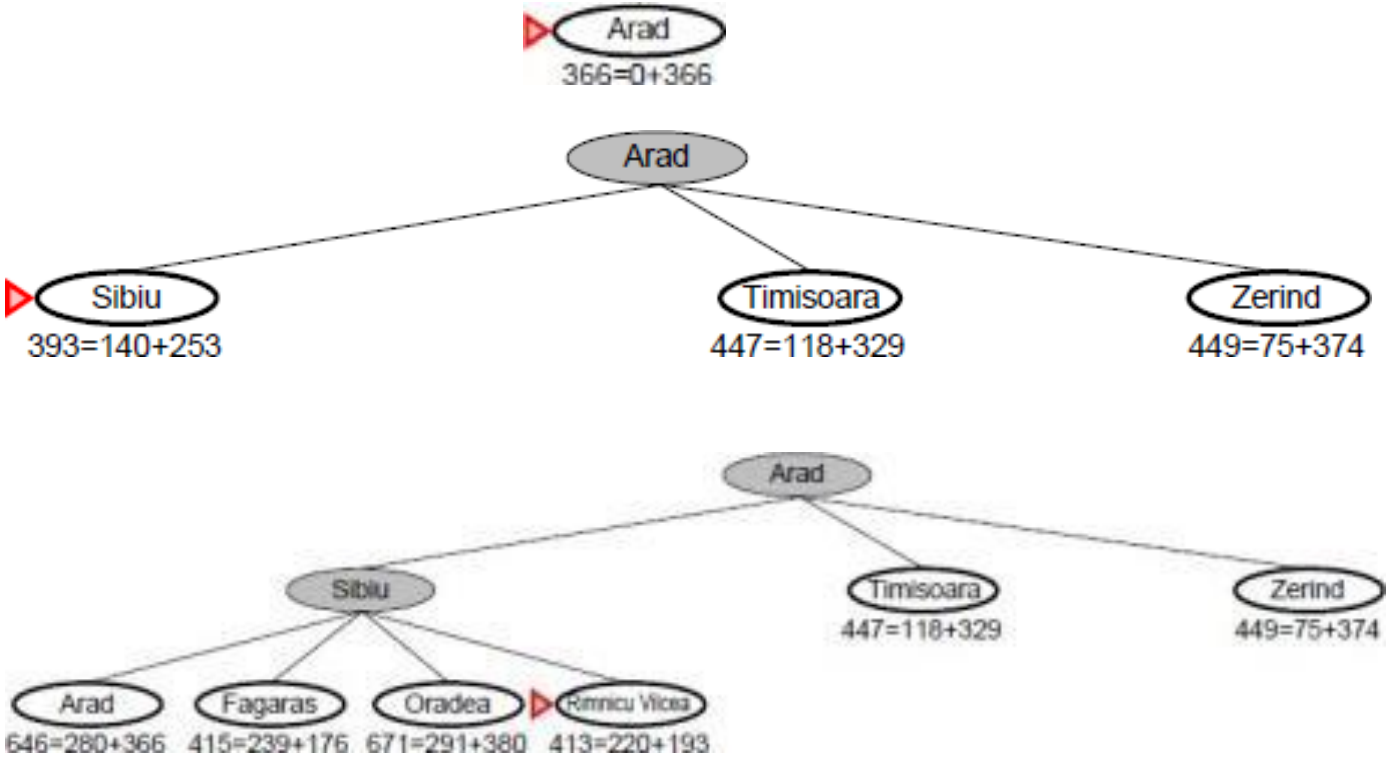

A*

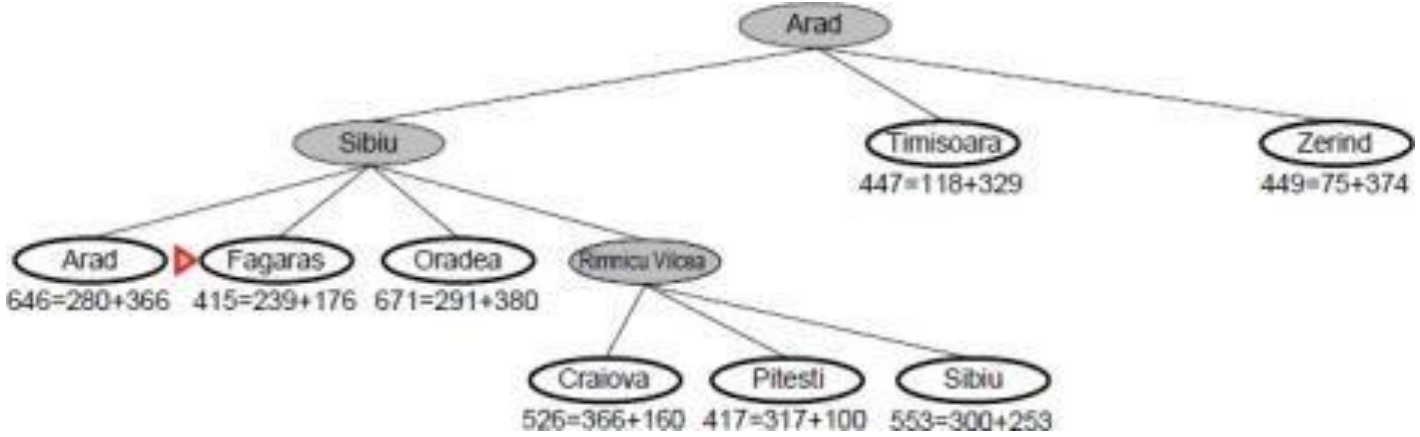
*A* total cost:*
*104 + 81 + 80 + 90 = 355 kms*

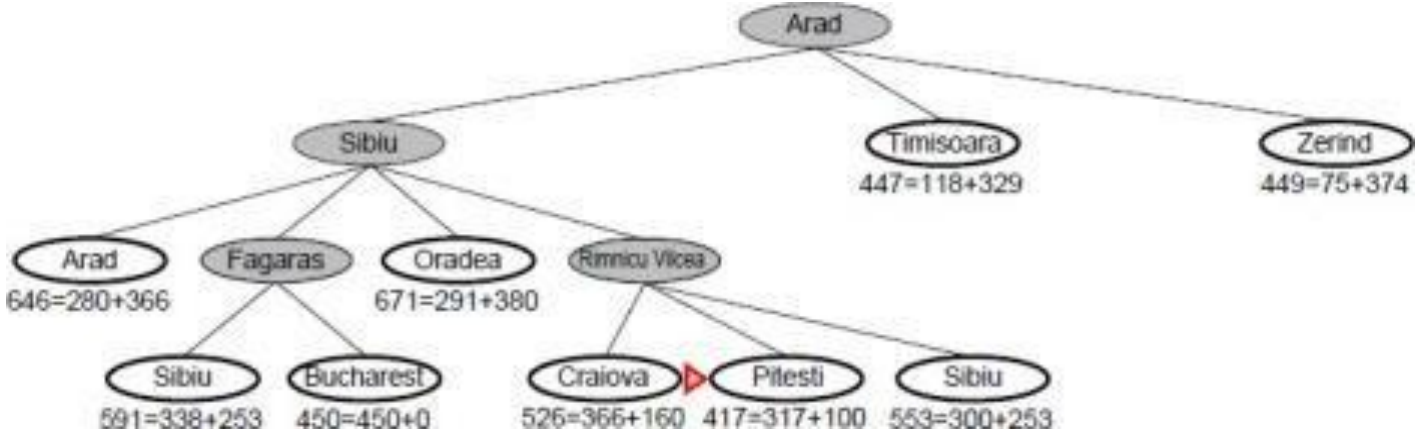
*greedy search total cost:*
*104 + 157 + 110 = 371 kms*

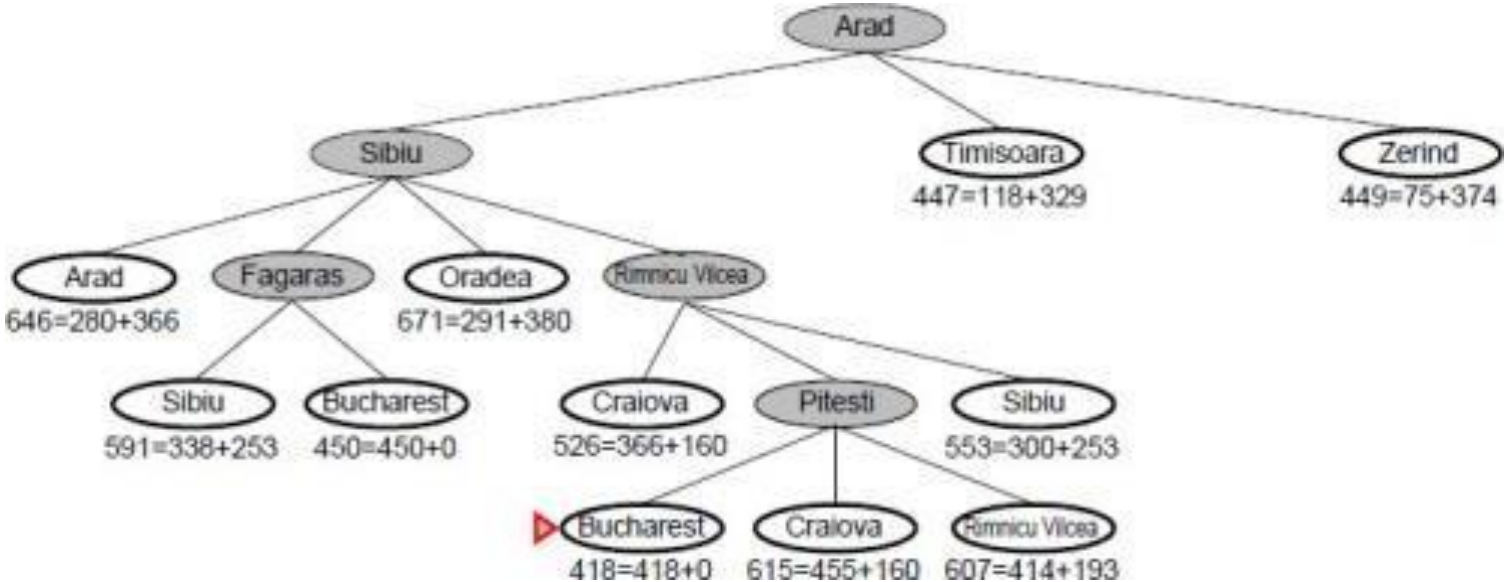



| Straight–line distance to Bucharest | |
|---|---|
| Arad | 366 |
| Bucharest | 0 |
| Craiova | 160 |
| Dobreta | 242 |
| Eforie | 161 |
| Fagaras | 176 |
| Giurgiu | 77 |
| Hirsova | 151 |
| Iasi | 226 |
| Lugoj | 244 |
| Mehadia | 241 |
| Neamt | 234 |
| Oradea | 380 |
| Pitesti | 10 |
| Rimnicu Vilcea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


Arad
366=0+366
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

# Admissible heuristics

- A good heuristic can be powerful, only if it is of a *"good quality"*
- A good heuristic must be *admissible*
- An admissible heuristic never *overestimates* the cost to reach the goal, that is it is *optimistic*
- *A heuristic h is admissible if*

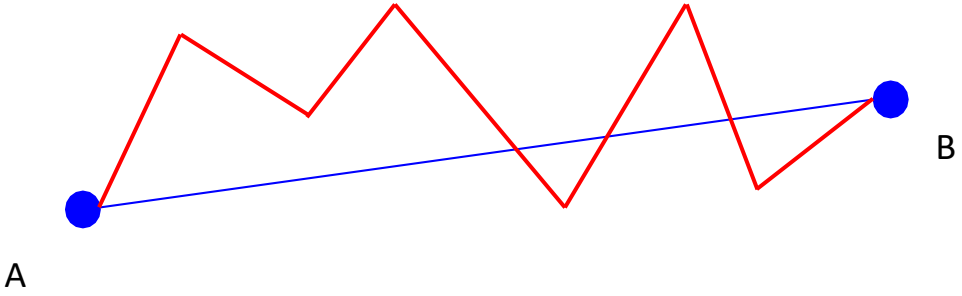
$$\forall\ node\ n,\ h(n) \leq h^*(n)$$

*where $h^*$ is the true cost to reach the goal from $n$*

- A heuristic function is said to be **admissible** if it is no more than the lowest-cost path to the goal.
- In other words, a heuristic is admissible if it never overestimates the cost of reaching the goal.
- An admissible heuristic is also known as an **optimistic heuristic**.

# Admissible heuristics

- $h_{SLD}$ (used as a heuristic in the map example) is admissible because it is by definition the shortest distance between two points.




*blue = SLD*

*red = actual distance*

*SLD <= actual distance*

***so $h_{SLD}$ is an admissibleheuristic***

# A* search criteria

- Complete: Yes
- Time: exponential
- Space: keeps every node in memory, the biggest problem
- Optimal: Yes!

***If $h(n)$ is admissible, A* using tree search is optimal.***

# Hill Climbing Search

- Hill climbing can be used to solve problems that have many solutions, some of which are better than others.
- **It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little.**
- **When the algorithm cannot see any improvement anymore, it terminates.**
- not guaranteed that hill climbing will ever come close to the optimal solution.

- For example, hill climbing can be applied to the traveling salesman problem.
- It is easy to find a solution that visits all the cities but will be very poor compared to the optimal solution.
- The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited.
- Eventually, a much better route is obtained.
- In hill climbing the basic idea is to always head towards a state which is better than the current one.
- So, if you are at town A and you can get to town B and town C (and your target is town D) then you should make a move IF town B or C appear nearer to town D than town A does.

# The hill climbing can be described as follows

- Start with *current-state* = initial-state.
- Until *current-state* = goal-state OR there is no change in *current-state* do:
  - Get the successors of the current state and use the evaluation function to assign a score to each successor.
  - If one of the successors has a better score than the current-state then set the new current-state to be the successor with the best score.
- Hill climbing terminates when there are no successors of the current state which are better than the current state itself.
- ***Hill climbing is depth-first search with a heuristic measurement that orders choices as nodes are expanded.***
- ***It always selects the most promising successor of the node last expanded.***

- The difference between the hill climbing search method and the best first search method is the following one:
  - the best first search method selects for expansion the most promising leaf node of the current search tree;
  - the hill climbing search method selects for expansion the most promising successor of the node last expanded.

# Problems with Hill Climbing

- Gets stuck at **local minima** when we reach a position where there are no better neighbors, it is not a guarantee that we have found the best solution. **Ridge** is a sequence of local maxima.
- Another type of problem we may find with hill climbing searches is finding a ***plateau.*** This is an area where the search space is flat so that all neighbors return the same evaluation

# Simulated Annealing

- It is motivated by the physical annealing process in which material is heated and slowly cooled into a uniform structure.
- Compared to hill climbing the main difference is that SA allows downwards steps.
- Simulated annealing also differs from hill climbing in that a move is selected at random and then decides whether to accept it.
- If the move is better than its current position then simulated annealing will always take it.
- If the move is worse (i.e. lesser quality) then it will be accepted based on some probability.

- The probability of accepting a worse state is given by the equation
- P = exponential(-*c* /*t*) > r
  - *c* = the change in the evaluation function
  - *t* = the current value
  - *r* = a random number between 0 and 1
- The probability of accepting a worse state is a function of both the current value and the change in the cost function.
- The most common way of implementing an SA algorithm is to implement hill climbing with an accept function and modify it for Simulated Annealing

- By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter *T* (called the *temperature*), that is gradually decreased during the process.
- The dependency is such that the current solution changes almost randomly when *T* is large, but increasingly "downhill" as *T* goes to zero.
- The allowance for "uphill" moves saves the method from becoming stuck at local optima—which are the bane of greedier methods.

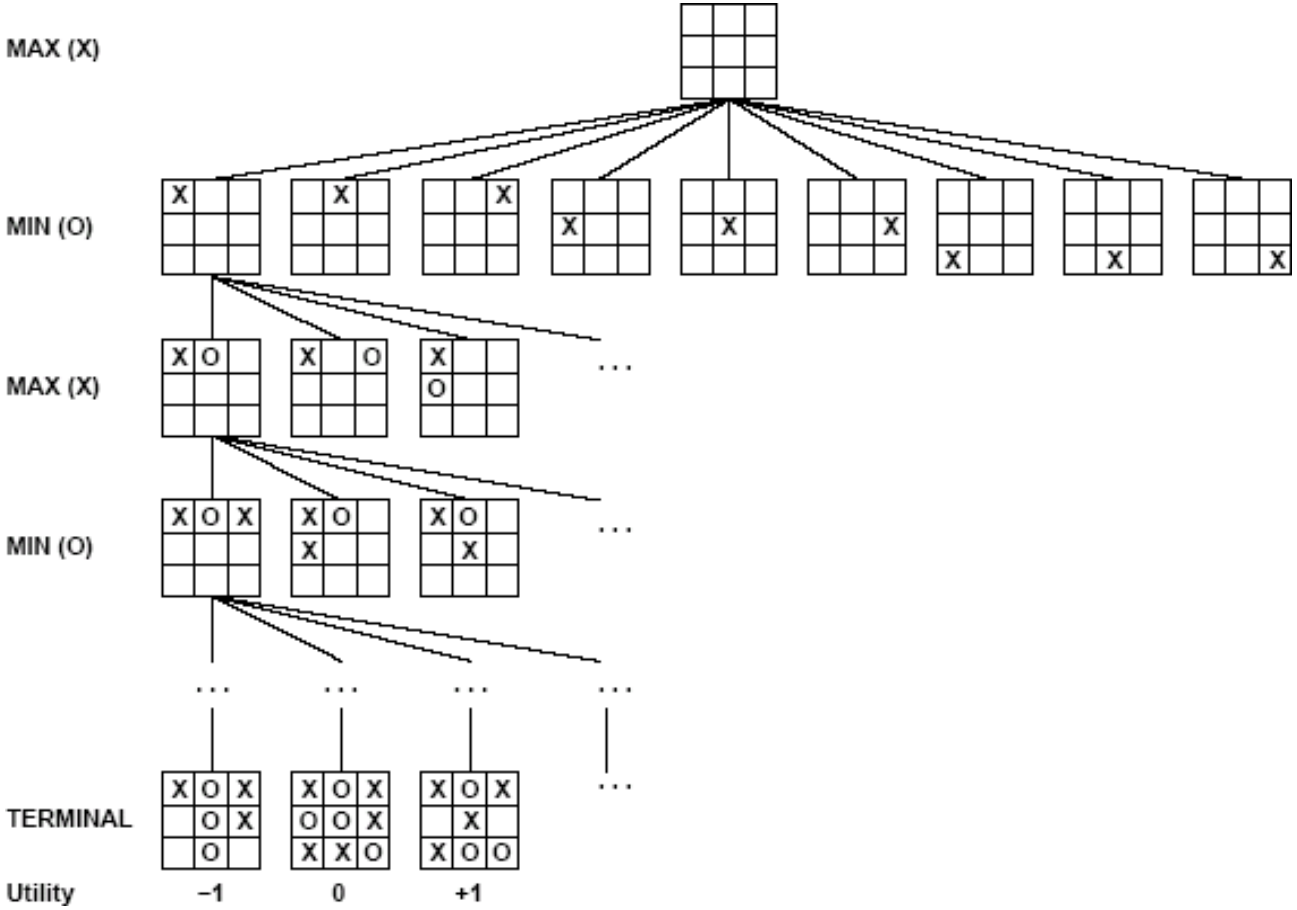
# Game Search

- Games are a form of *multi-agent environment*
  - What do other agents do and how do they affect our success?
  - Cooperative vs. competitive multi-agent environments.
  - Competitive multi-agent environments give rise to adversarial search often known as *games*
- Games – adversary
  - Solution is strategy (strategy specifies move for every possible opponent reply).
  - Time limits force an *approximate* solution
  - Evaluation function: evaluate "goodness" of game position
  - Examples: chess, checkers, Othello, backgammon

- Difference between the search space of a game and the search space of a problem:
- In the first case it represents the moves of two (or more) players, whereas in the latter case it represents the "moves" of a single problem-solving agent.

# Tic-tac-toe

- There are two players denoted by X and O. They are alternatively writing their letter in one of the 9 cells of a 3 by 3 board. The winner is the one who succeeds in writing three letters in line.
- The game begins with an empty board. It ends in a win for one player and a loss for the other, or possibly in a draw.
- A complete tree is a representation of all the possible plays of the game. The root node is the initial state, in which it is the first player's turn to move (the player X).
- The successors of the initial state are the states the player can reach in one move, their successors are the states resulting from the other player's possible replies, and so on.
- Terminal states are those representing a win for X, loss for X, or a draw.
- Each path from the root node to a terminal node gives a different complete play of the game. Figure given below shows the initial search space of Tic-Tac-Toe.


MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
−1
0
+1

# A game can be formally defined as a kind of search problem as below

- Initial state: It includes the board position and identifies the playesr to move.
- Successor function: It gives a list of (move, state) pairs each indicating a legal move and resulting state.
- Terminal test: This determines when the game is over. States where the game is ended are called terminal states.
- Utility function: It gives numerical value of terminal states. E.g. win (+1), loose (- 1) and draw (0). Some games have a wider variety of possible outcomes eg. ranging from +92 to -192.

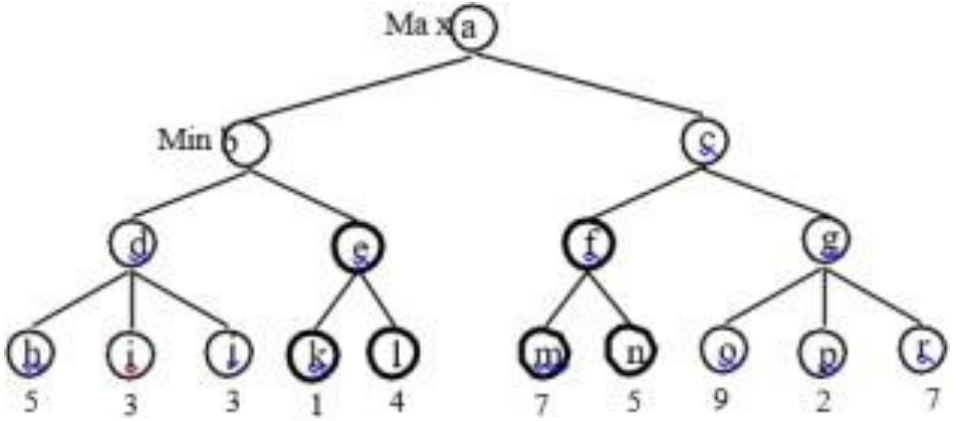
# The Minimax Algorithm

- Let us assign the following values for the game: 1 for win by X, 0 for draw, -1 for loss by X.
- Given the values of the terminal nodes (win for X (1), loss for X (-1), or draw (0)), the values of the non-terminal nodes are computed as follows:
  - the value of a node where it is the turn of player X to move is the maximum of the values of its successors (because X tries to maximize its outcome);
  - the value of a node where it is the turn of player O to move is the minimum of the values of its successors (because O tries to minimize the outcome of X).

- Figure below shows how the values of the nodes of the search tree are computed from the values of the leaves of the tree. The values of the leaves of the tree are given by the rules of the game: ·
  - 1 if there are three X in a row, column or diagonal;
  - -1 if there are three O in a row, column or diagonal;
  - 0 otherwise

# An Example

- Consider the following game tree (drawn from the point of view of the Maximizing player):



- Show what moves should be chosen by the two players, assuming that both are using the mini-max procedure.
- Solution:

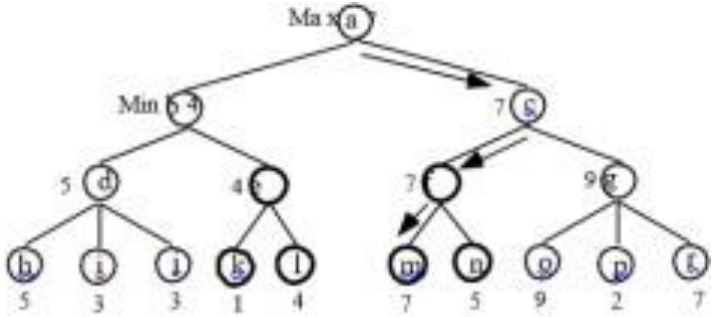



Figure 3.16: The mini-max path for the game tree