BIRZEIT UNIVERSITY

Computer Science Department

Second Semester 2017/2018

Comp338 – Artificial Intelligence

Instructor: Mustafa Jarrar

Notes + Study guide

Prepared by: Maryam Shaheen

Introduction to AI – Chapter1

What is the definition of AI?

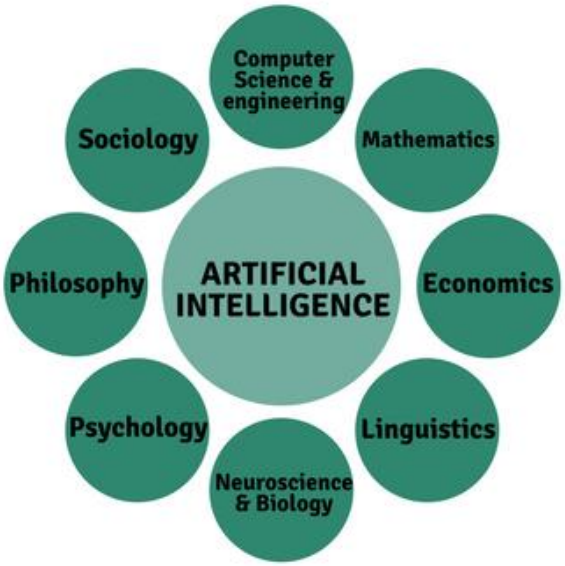
- "Intelligence: The ability to learn and solve problems" Webster's Dictionary
- "Artificial intelligence (AI) is the intelligence exhibited by machines or software" Wikipedia
- "The science and engineering of making intelligent machines" McCarthy
- "The study and design of intelligent agents, where an intelligent agent is a system that perceives its environment and takes actions that maximize its chances of success." Russel and Norvig AI book

What is AI?

Four schools of thoughts (Russel & Norvig)

Thinking humanly	**Thinking rationally**
"The exciting new effort to make computers think... machines with minds, in the full and literal sense." (Haugeland, 1985)	**"The study of mental faculties through the use of computational models." (Charniak & McDermott,1985)**
Acting humanly	**Acting rationally**
"The study of how to make computers do things which, at the moment, people are better." (Rich & Knight, 1991)	**"Computational Intelligence is the study of the design of intelligent agents." (Poole et al., 1998)**

What are the foundations of AI?



-Philosophy
Philosophers (going back to 400 B.C.) made AI conceivable by considering the ideas that the mind is in some ways like a machine, that operates on knowledge encoded in some internal language, and that thought can be used to choose what actions to take.

-Mathematics
Mathematicians provided the tools to manipulate statements of logical certainty as well as uncertain, probabilistic statements. They also set the groundwork for understanding computation and reasoning about algorithms.

-Economics
Economists formalized the problem of making decisions that maximize the expected outcome to the decision maker.

-Neuroscience
Neuroscientists discovered some facts about how the brain works and the ways in which it is similar to and different from computers.

-Psychology
Psychologists adopted the idea that humans and animals can be considered information-processing machines.

-Linguistics
Linguists showed that language use fits into this model.

-Computer engineering
Computer engineers provided the ever-more-powerful machines that make AI applications possible.

-Control theory and cybernetics
Control theory deals with designing devices that act optimally on the basis of feedback from the environment. Initially, the mathematical tools of control theory were quite different from AI, but the fields are coming closer together.

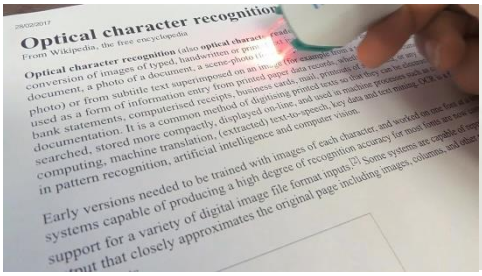
What are AI Applications?

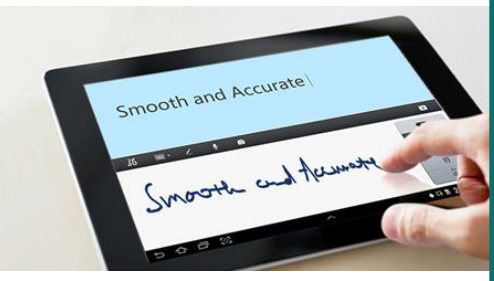
- ***Smart Search Algorithms***
  - **Games**
  - **Route finding**
  - **Transportation/scheduling**
  - **Traveling salesperson**
  - **VLSI layout**
  - **Automatic assembly**

…. Many more!

- ***NLP Applications***
  - **Search engines**
  - **OCR**
  - **Speech recognition**
  - **Machine translation**
  - **Spam fighting**
  - **Information extraction**
  - **Summarization**
  - **Spelling checkers**
  - **Grammar Cheekers**
  - **Sentiment analysis**

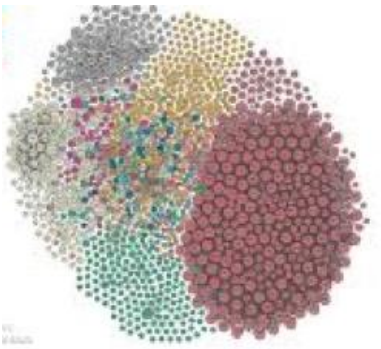
…. Many more!

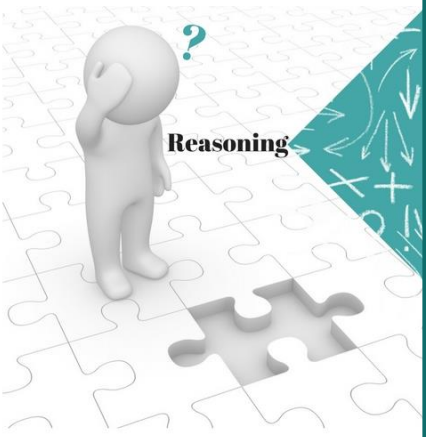




- ***Knowledge base Applications***
  - **Semantic Web**
  - **Expert Systems**
  - **Reasoning**
  - **Logic based games**
  - **System interoperability**
  - **Semantic search**
  - **Medical diagnosis**

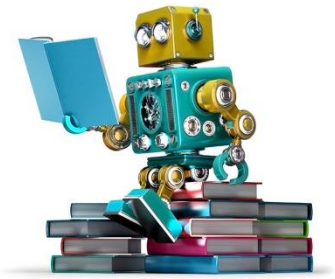
…. Many more!

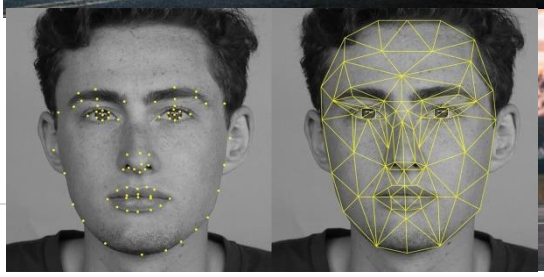




- ***Machine learning***
  - **Face Recognition**
  - **Autonomous cars**
  - **Social network analysis**
  - **Recommendation systems**
  - **Fraud detection**
  - **Financial forecasting**

…. Many more!



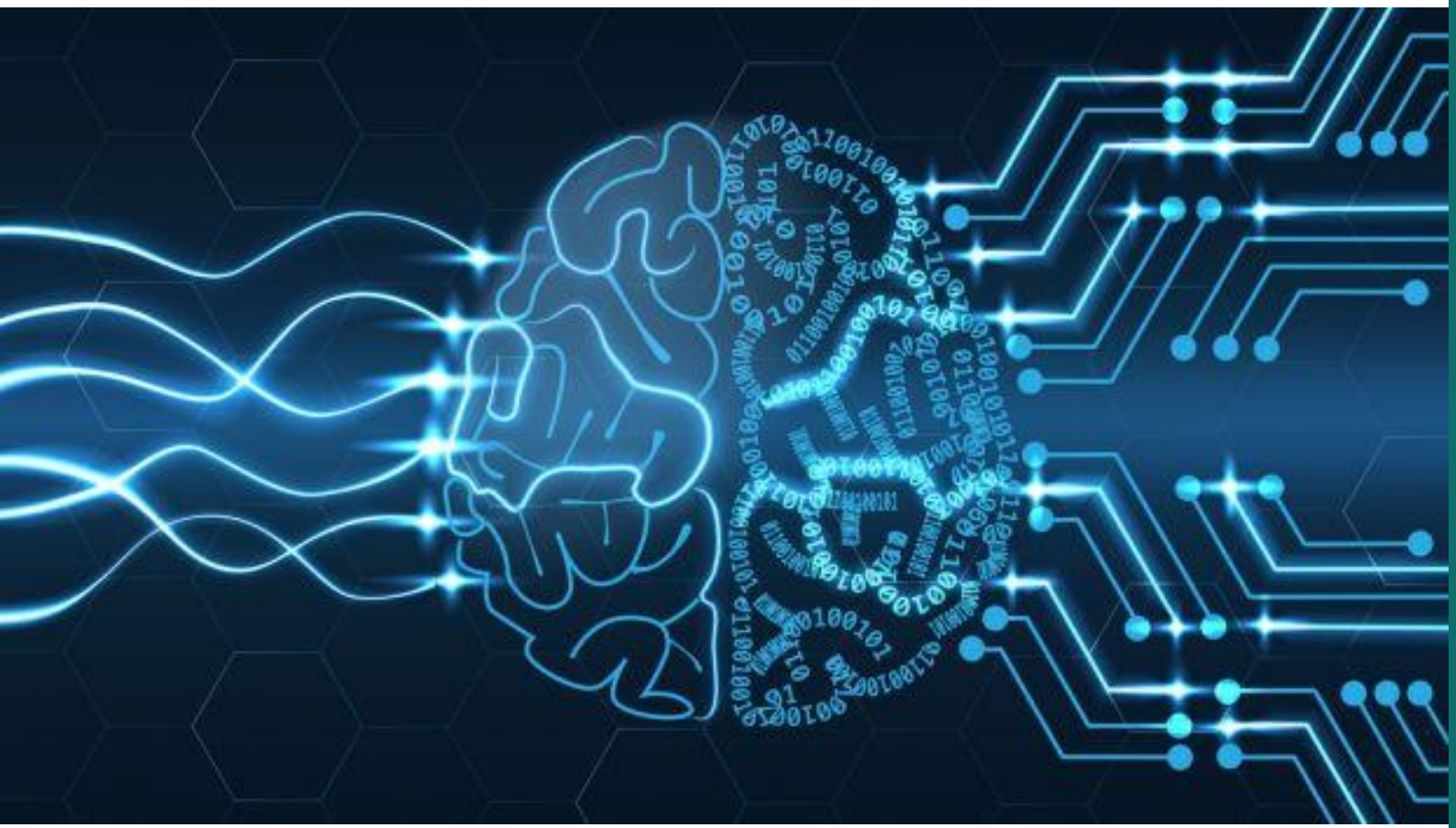


Should artificial intelligence simulate natural intelligence?

No, because artificial intelligence (AI) is intelligence demonstrated by machines, in contrast to the natural intelligence (NI) that is displayed by humans and other animals.

What are the criticisms on the AI research?

The AI field was founded on the claim that human intelligence "can be so precisely described that a machine can be made to simulate it". This raises philosophical arguments about the nature of the mind and the ethics of creating artificial beings endowed with human-like intelligence, issues which have been explored by myth, fiction and philosophy since antiquity. Some people also consider AI to be a danger to humanity if it progresses unabatedly. Others believe that AI, unlike previous technological revolutions, will create a risk of mass unemployment.



Intelligent Agents – Chapter2

What is the definition of Intelligent Agent?

It's an autonomous entity which observes through sensors and acts upon an environment using actuators (i.e. it is an agent) and directs its activity towards achieving goals (i.e. it is "rational", as defined in economics). Intelligent agents may also learn or use knowledge to achieve their goals. They may be very simple or very complex. A reflex machine, such as a thermostat, is considered an example of an intelligent agent.

- An **agent** is something that perceives and acts in an environment.
  The **agent function** for an agent specifies the action taken by the agent in response to any percept sequence.

- The **performance measure** evaluates the behavior of the agent in an environment.
  A **rational agent** acts so as to maximize the expected value of the performance measure, given the percept sequence it has seen so far.

- A **task environment** specification includes the performance measure, the external environment, the actuators, and the sensors. In designing an agent, the first step must always be to specify the task environment as fully as possible.
  Task environments vary along several significant dimensions. They can be fully or partially observable, single-agent or multi-agent, deterministic or stochastic, episodic or sequential, static or dynamic, discrete or continuous, and known or unknown.

- The **agent program** implements the agent function. There exists a variety of basic agent-program designs reflecting the kind of information made explicit and used in the decision process. The designs vary in efficiency, compactness, and flexibility. The appropriate design of the agent program depends on the nature of the environment.

- **Simple reflex agents** respond directly to percepts, whereas **model-based reflex agents** maintain internal state to track aspects of the world that are not evident in the current percept.
  Goal-based agents act to achieve their goals, and **utility-based agents** try to maximize their own expected "happiness."
- All **agents** can improve their performance through **learning**.

Problem Solving by Search

- ***Uninformed Search – Chapter 3***
- ***Informed and heuristic Search – Chapter 4***
- ***Constraints Satisfaction – Chapter 5***
- ***Games and Adversarial Search – Chapter 6***

Uninformed Search – Chapter 3

This chapter describes one kind of goal-based agent called a **problem-solving agent**. /

Problem-solving agents use **atomic** representations

• Before an agent can start searching for solutions, a **goal** must be identified and a well-defined **problem** must be formulated.

A problem consists of **five** parts:

- the **initial state**
- a set of **actions**
- a **transition model** describing the results of those actions
- a **goal test** function
- And a **path cost** function.

The environment of the problem is represented by a **state space**.

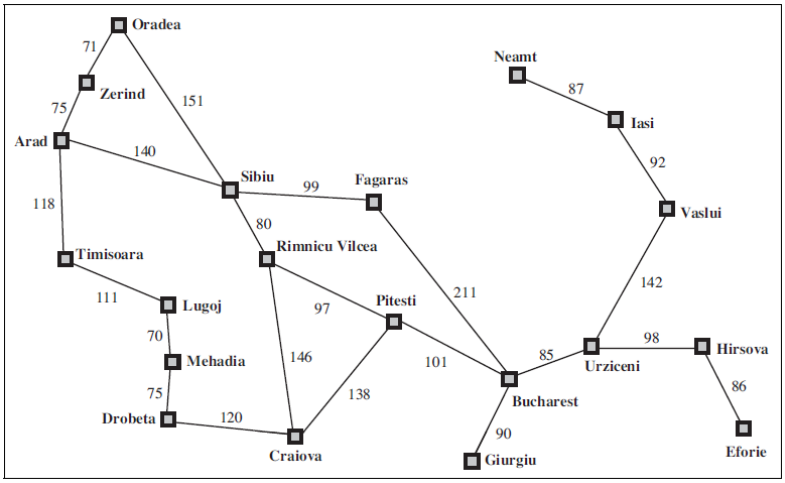
A **path** through the state space from the initial state to a goal state is a **solution**.

The following schema to help us formulate problems

1. State
2. Initial state
3. Actions or Successor Function
4. Goal Test
5. Path Cost
6. Solution

Examples of problem formulating:

EXP#1: The Romania Example
(From the book)



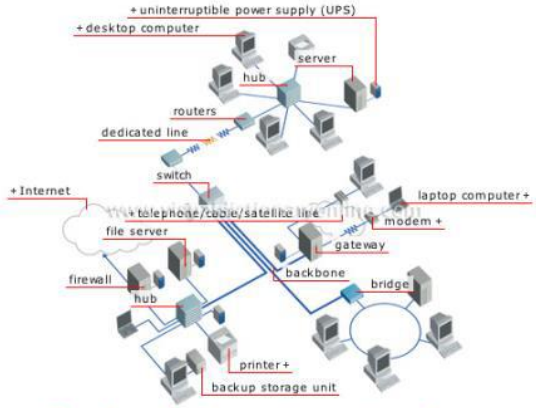
- **State**: We regard a problem as state space here a state is a City
- **Initial State**: the state to start from in(Arad)
- **Successor Function:** description of the possible actions, give state x, S(X) returns a set of <action, successor> ordered pairs.
  S(x)={ <Go(Sibiu), In(Sibiu)>, <Go(Timisoara), In(Timisoara)>, <Go(Zerind),In(Zerind)> }
- **Goal Test:** determine a given state is a goal state.
  in(Sibiu) →No. In(Zerind) →No….. in(Bucharest)→Yes!
- **Path Cost:** a function that assigns a numeric cost to each path.
  - e.g., sum of distances, number of actions executed, etc.
  - c(x, a, y) is the step cost, assumed to be ≥ 0
- **Solution:** a sequence of actions leading from the initial state to a goal state
  {Arad →Sibiu →RimnicuVilcea→Pitesti →Bucharest}

Real-life Applications

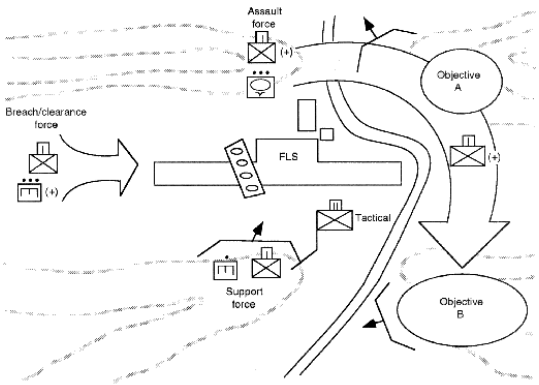
EXP#2: Route Finding Problem

Car Navigation

Airline travel planning



Routing in Computer networks



Military operation planning

State: locations

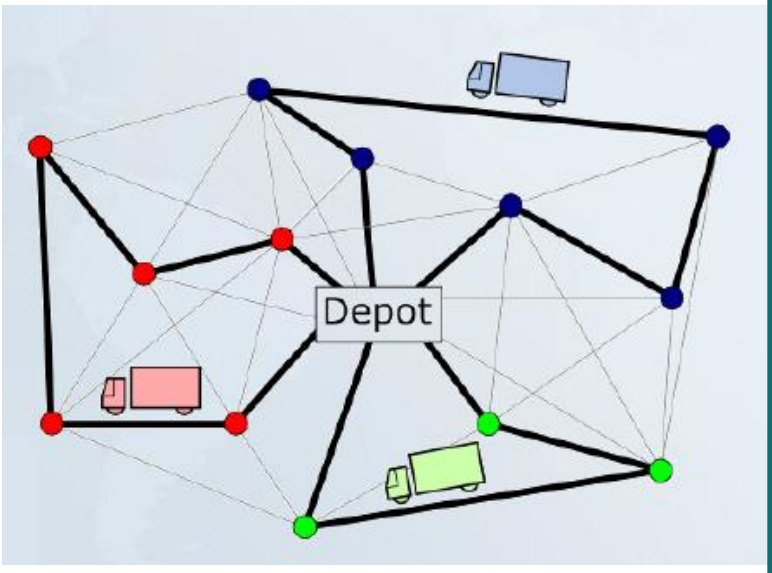
Initial state: starting point

Successor function (operators): move from one location to another

Goal test: arrive at a certain location

Path cost: may be quite complex money, time, travel comfort, scenery,

Routing Problem



What is the state space for each of them?

➔ A set of places with links between them, which have been visited

Travel Salesperson Problem

State:

- locations / cities
- illegal states
  - each city may be visited only once
  - visited cities must be kept as state information

Initial state:

- starting point:
- no cities visited

Successor function (operators):

- move from one location to another one

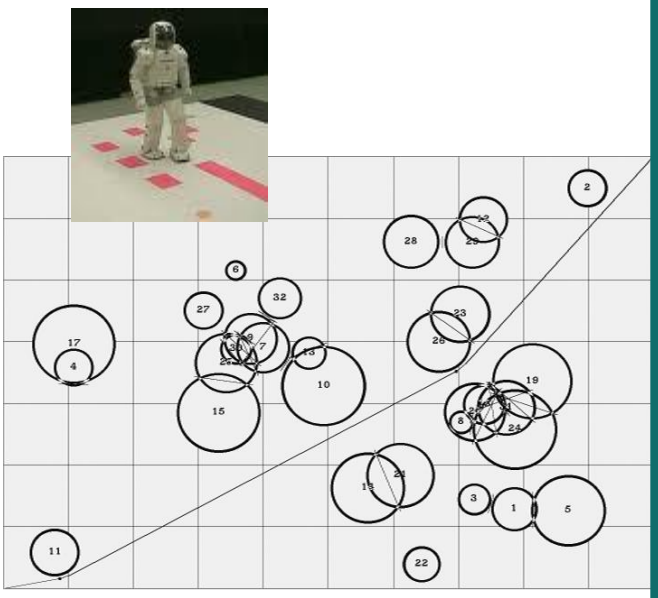
Goal test:

- all locations visited
- agent at the initial location

Path cost:

- distance between locations

Robot Navigation



States:
–locations
–position of actuators

Initial state:
–start position (dependent on the task)

Successor function (operators):
–movement, actions of actuators

Goal test:
–task-dependent

Path cost:
–maybe very complex
•distance, energy consumption

Automatic Assembly Sequencing

State:
–location of components

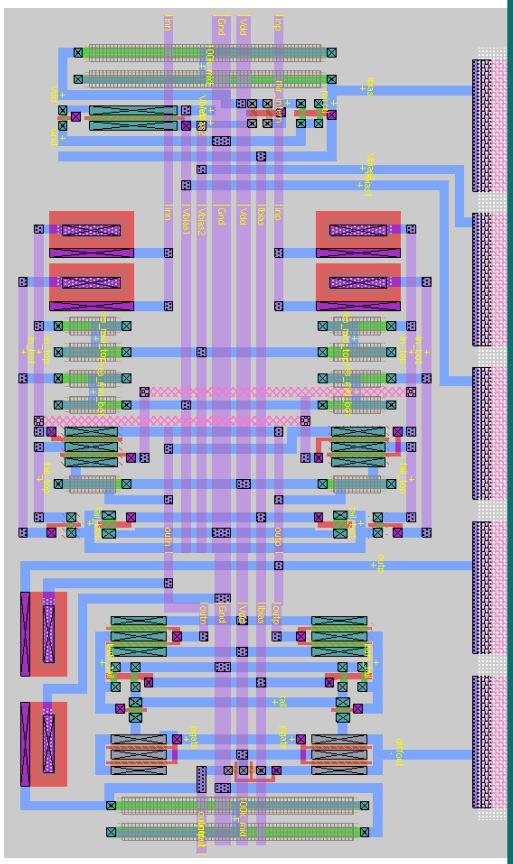
Initial state:
–no components assembled

Successor function (operators):
–place component

Goal test:
–system fully assembled

Path cost:
–number of moves

VLSI layout Problem



State:
–positions of components, wires on a chip

Initial state:
–incremental: no components placed
–Complete-state: all components placed (e.g. randomly, manually)

Successor function (operators):
–incremental: place components, route wire
–Complete-state: move component, move wire

Goal test:
–all components placed
–components connected as specified

Path cost:
–maybe complex
•distance, capacity, number of connections per component

Searching for Solutions

- Traversal of the search space
  - From the initial state to a goal state.
  - Legal sequence of actions as defined by successor function.

- *General procedure*
  - *Check for goal state*
  - *Expand the current state*

- *Determine the set of reachable states*

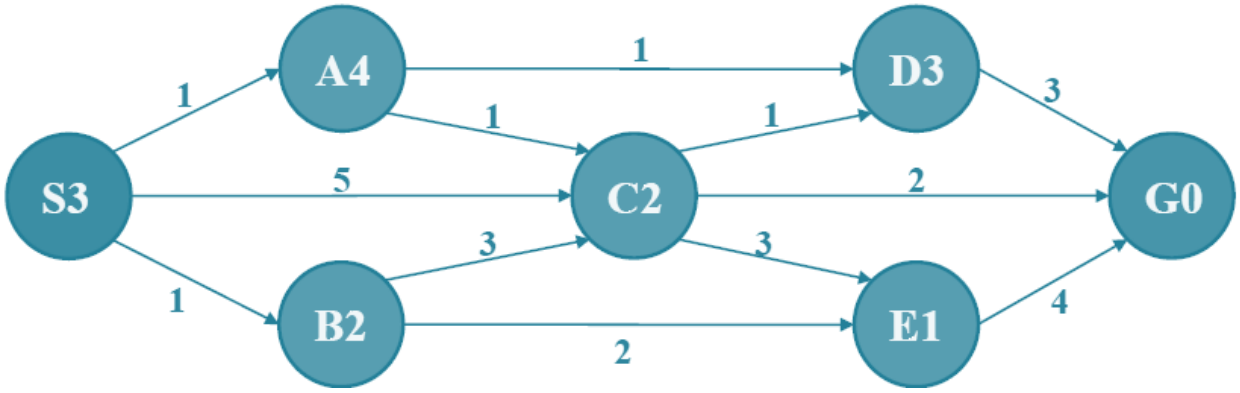
- *Return "failure" if the set is empty*
  - *Select one from the set of reachable states*
  - *Move to the selected state*

- *A search tree is generated*
  - *Nodes are added as more states are visited*

Search Terminology

- ***Search Tree***
  - Generated as the search space is traversed
    - The search space itself is not necessarily a tree, frequently it is a graph
    - The tree specifies possible paths through the search space
- ***Expansion of nodes***
  - As states are explored, the corresponding nodes are expanded by applying the successor function
    - this generates a new set of (child) nodes
- ***The fringe(frontier/queue) is the set of nodes not yet visited***
  - newly generated nodes are added to the fringe
- ***Search strategy***
  - Determines the selection of the next node to be expanded
  - Can be achieved by ordering the nodes in the fringe
    - e.g. queue (FIFO), stack (LIFO), "best" node w.r.t. some measure (cost)

Example: Graph Search



The graph describes the search (state) space

–Each node in the graph represents one state in the search space

- e.g. a city to be visited in a routing or touring problem

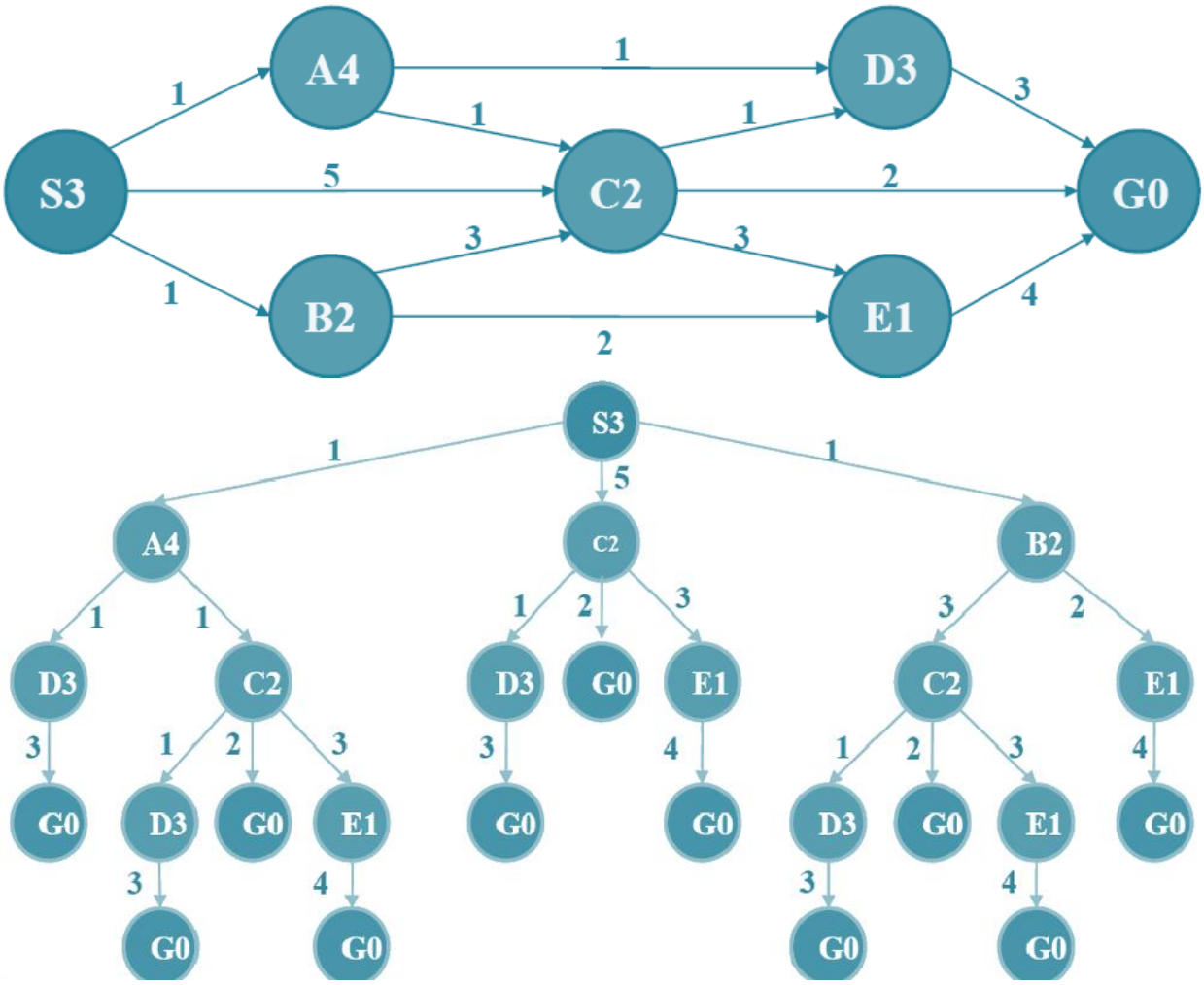
This graph has additional information

–Names and properties for the states (e.g. S3)

–Links between nodes, specified by the successor function

- properties for links (distance, cost, name …)

• Search algorithms treat states and actions as **atomic**: they do not consider any internal structure they might possess.
• A general **TREE-SEARCH** algorithm considers all possible paths to find a solution, whereas a **GRAPH-SEARCH** algorithm avoids consideration of redundant paths.



- A tree is generated by traversing the graph.
- The same node in the graph may appear repeatedly in the tree.
- The arrangement of the tree depends on the traversal strategy (search method)
- The initial state becomes the root node of the tree
- In the fully expanded tree, the goal states are the leaf nodes.
- Cycles in graphs may result in infinite branches.

- Search algorithms are judged on the basis of **completeness**, **optimality**, **time complexity**, and **space complexity**. Complexity depends on (b: the branching factor in the state space), and (d: the depth of the shallowest solution.)
- **Uninformed search** (blind search) methods have access only to the problem definition.
    - Number of steps and path cost are unknown
    - Agent knows when it reaches a goal

The basic algorithms are as follows:

1. **Breadth-first search** expands the shallowest nodes first; it is complete, optimal for unit step costs, but has exponential space complexity.

2. **Uniform-cost search** expands the node with lowest path cost, g(n), and is optimal for general step costs.

3. **Depth-first search** expands the deepest unexpanded node first. It is neither complete nor optimal, but has linear space complexity.

4. **Depth-limited search** adds a depth bound.

5. **Iterative deepening search** calls depth-first search with increasing depth limits until a goal is found. It is complete, optimal for unit step costs, has time complexity comparable to breadth-first search, and has linear space complexity.

6. **Bidirectional search** can enormously reduce time complexity, but it is not always applicable and may require too much space.

Evaluation of Search Strategies

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

- Strategies are evaluated along the following dimensions:
    - Completeness: if there is a solution, will it be found
    - Time complexity: How long does it takes to find the solution
    - Space complexity: memory required for the search
    - Optimality: will the best solution be found

- Time and space complexity are measured in terms of
    - b: maximum branching factor of the search tree
    - d: depth of the least-cost solution
    - m: maximum depth of the state space (may be ∞)

1- Breadth-First-Search (BFS)

- ***Completeness:*** Yes (if b is finite), a solution will be found if exists.
- **Time Complexity:** 1+b+b^2+b^3+… + b^d+ (bd+1-b) = b^{d+1} (nodes until the solution)
- **Space Complexity:** b^{d+1}(keeps every generated node in memory)
- ***Optimality:*** Yes (if cost = 1 per step)

→Suppose the branching factor b = 10 and the goal is at depth d=12

b: Branching Factor
d: the depth of the goal

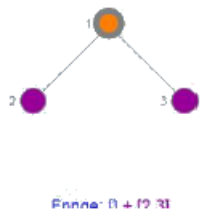
–Then we need Operation 10^{12} time to finish. If Operation is 0.001 second, then we need 1 billion seconds (31 year). And if each Operation costs 10 bytes to store, then we also need 1 terabytes.

→ Not suitable for searching large graphs

Applications/ Uses

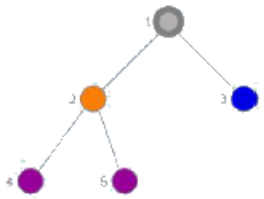
Breadth-first search can be used to solve many problems in graph theory, for example:

- Copying garbage collection, Cheney's algorithm
- Finding the shortest path between two nodes *u* and *v*, with path length measured by number of edges (an advantage over depth-first search)
- (Reverse) Cuthill–McKee mesh numbering
- Ford–Fulkerson method for computing the maximum flow in a flow network
- Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.
- Constructions of the failure function of the Aho-Corasick pattern matcher.
- Testing bipartiteness of a graph.

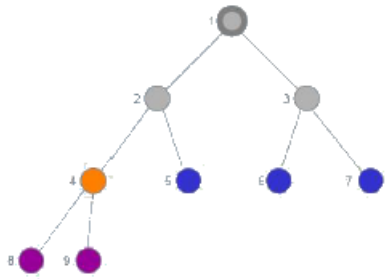


Fringe: [] + [2,3]

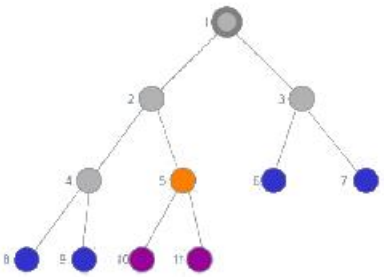
Fringe: [4,5] + [6,7]



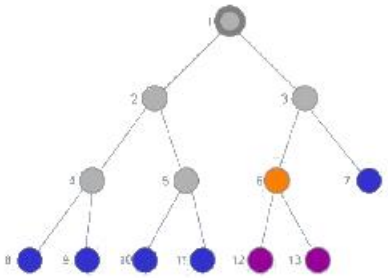
Fringe: [3] + [4,5]



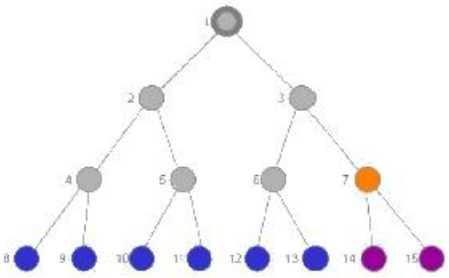
Fringe: [5,6,7] + [8,9]



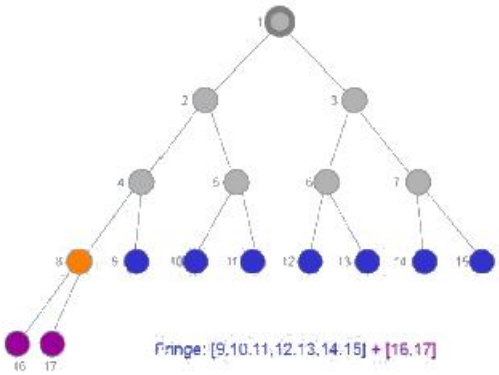
Fringe: [6,7,8,9] + [10,11]



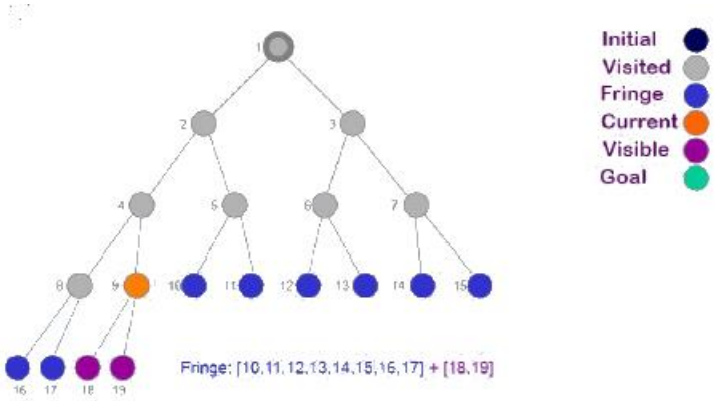
Fringe: [7,8,9,10,11] + [12,13]

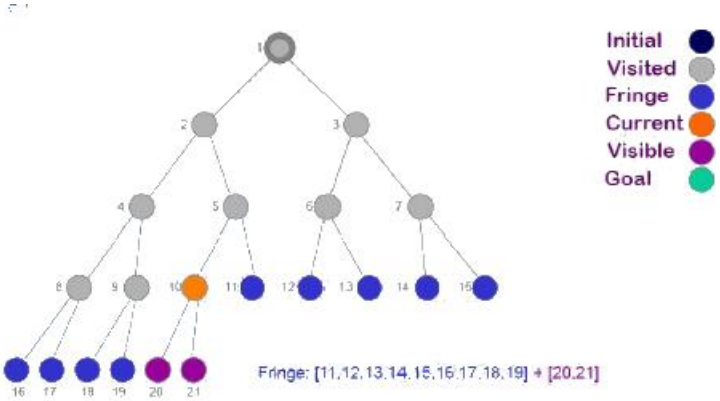


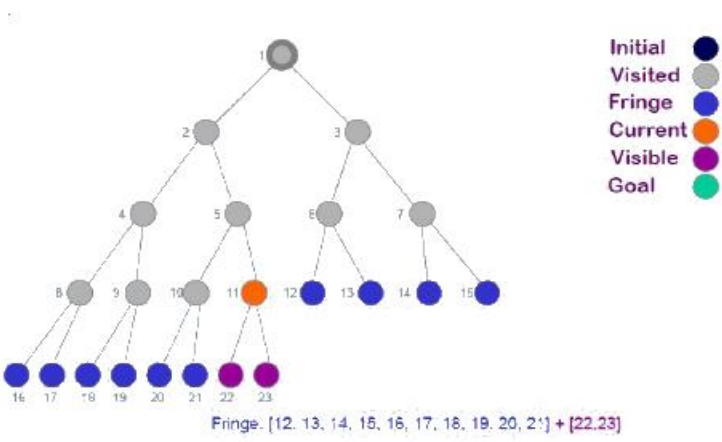
Fringe: [8,9,10,11,12,13] + [14,15]

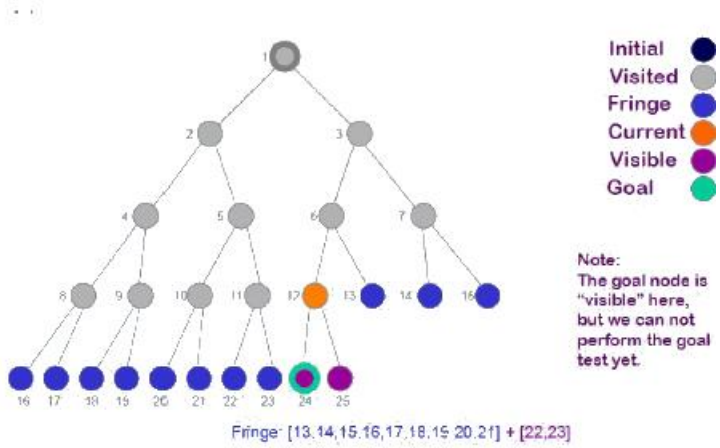


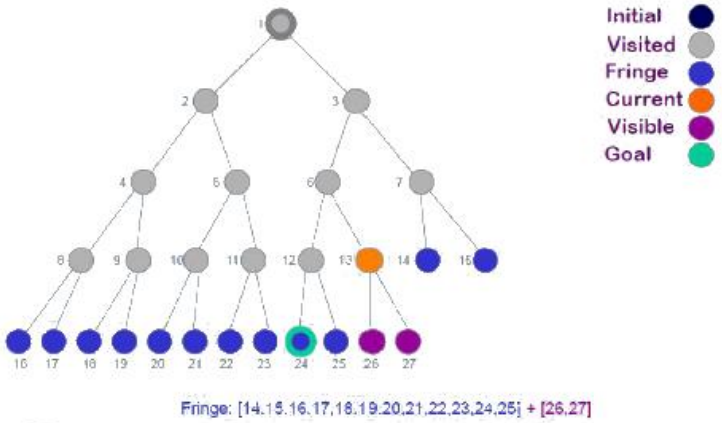
Fringe: [9,10,11,12,13,14,15] + [16,17]

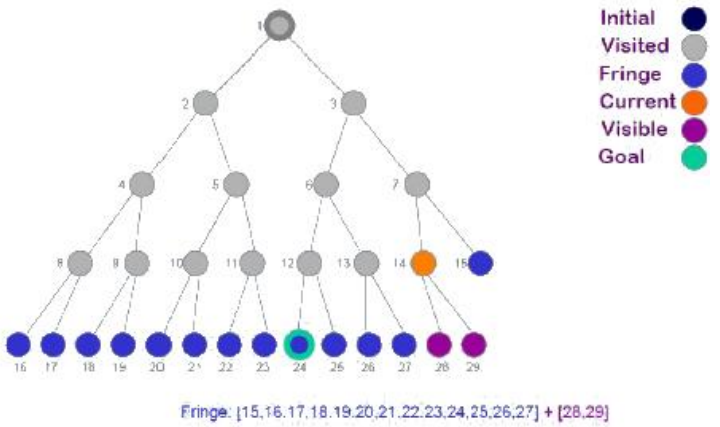


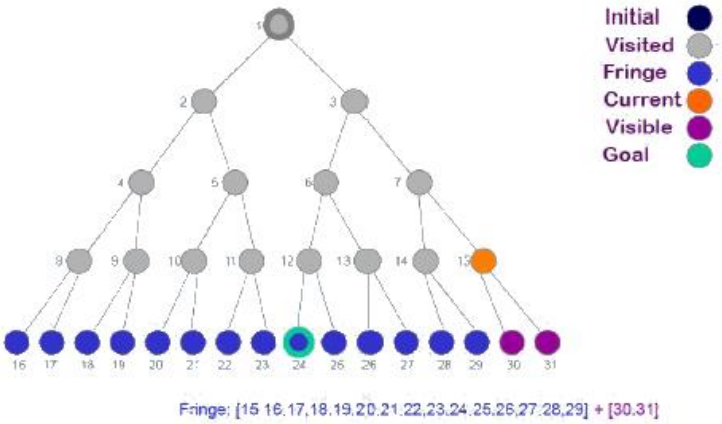


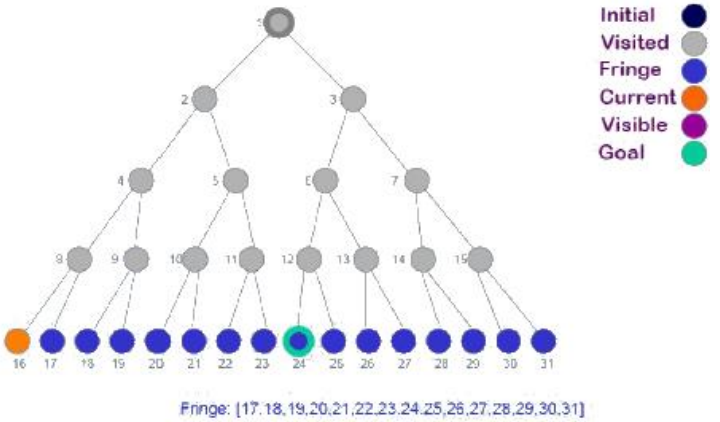


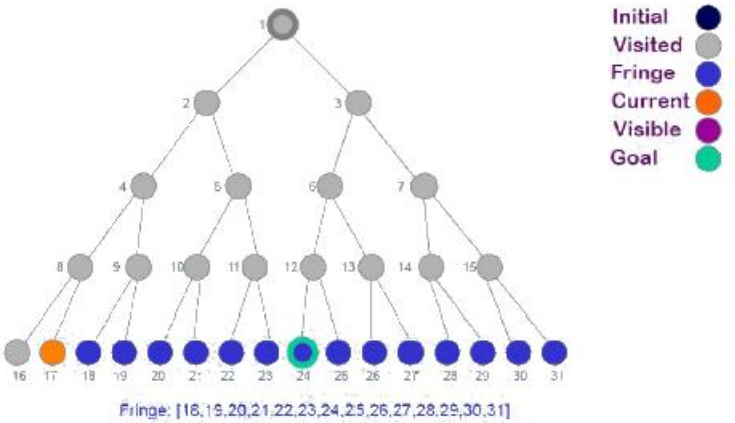


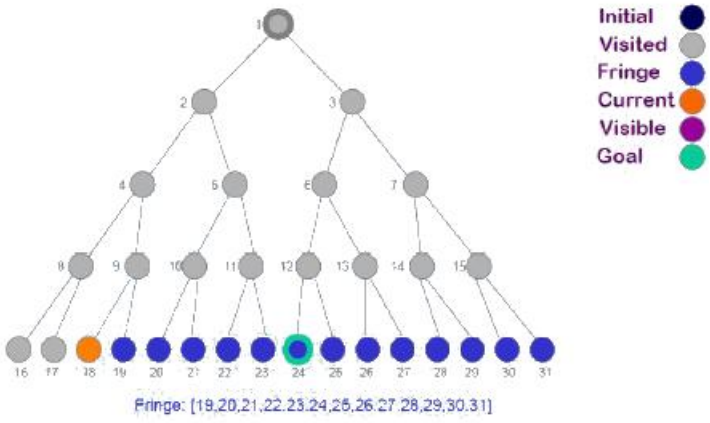


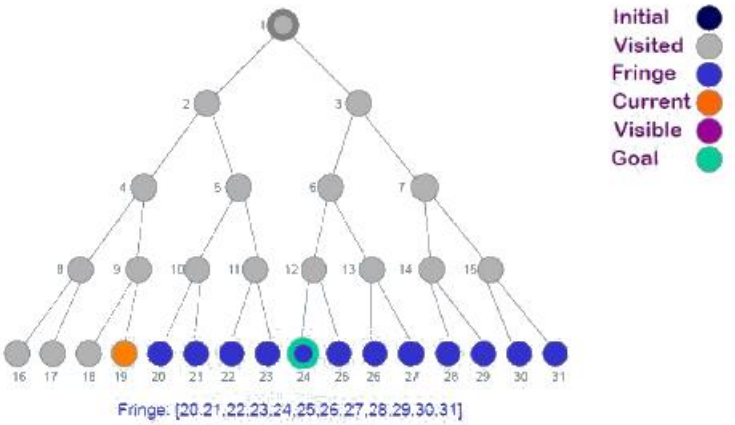


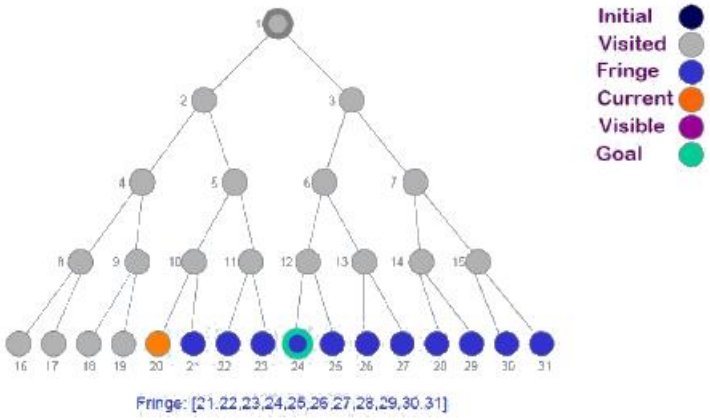


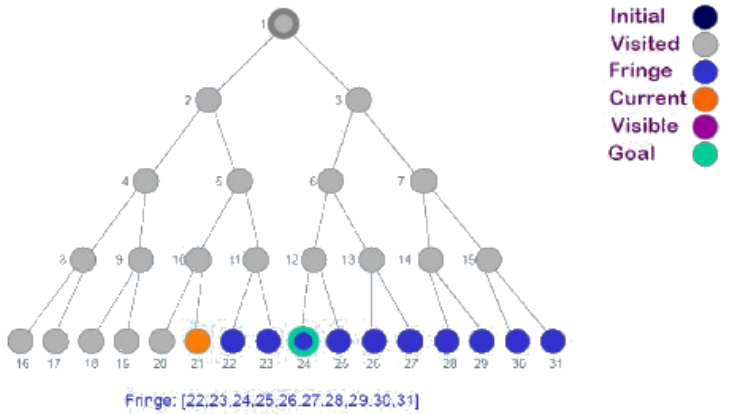


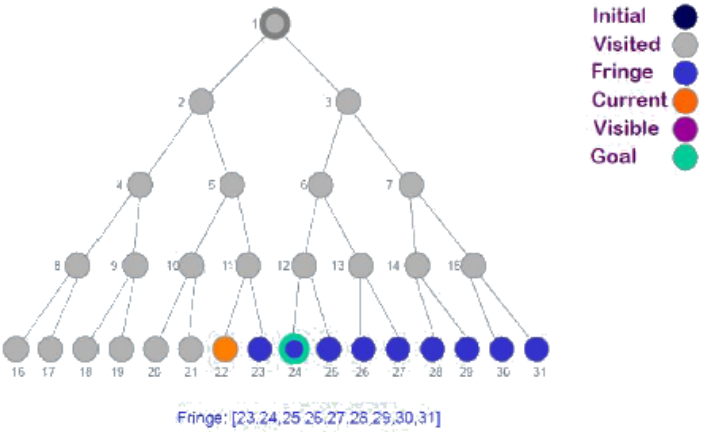


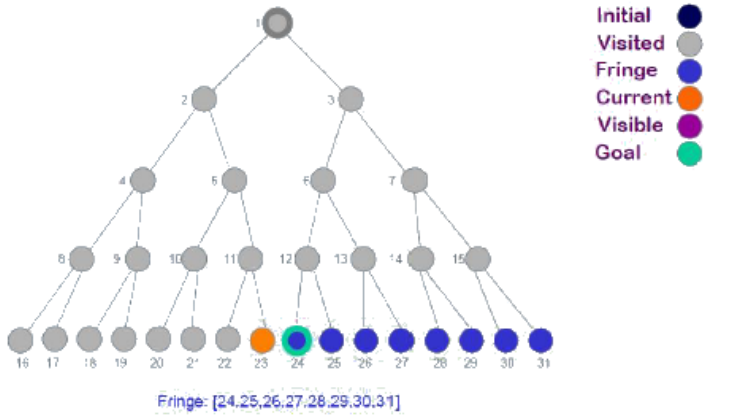


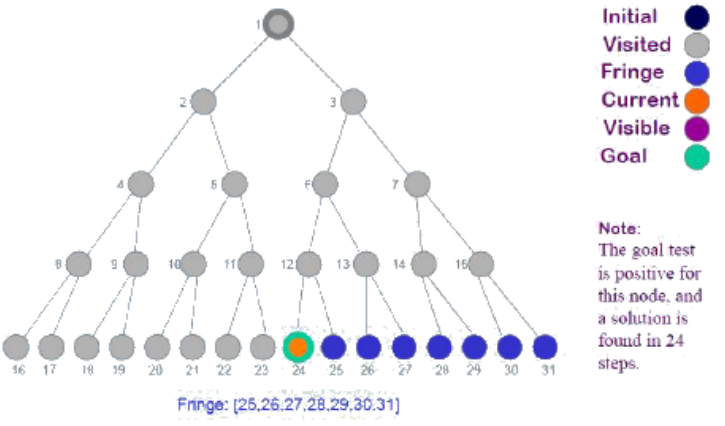












Note:
The goal test is positive for this node, and a solution is found in 24 steps.

Breadth-First-Search (BFS) Algorithm:

1. en-queue the root/initial node.
2. de-queue a node and examine it.
    1. If the element sought is found in this node, quit the search and return a result.
    2. Otherwise enqueuer any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined –quit the search and return "not found".
4. Repeat from Step 2.

Java Code:

```
// prints BFS traversal from a given source s
       void BFS(int s){

         // Mark all the vertices as not visited(By default
         // set as false)
         boolean visited[] = new boolean[V];

         // Create a queue for BFS
         LinkedList<Integer> queue = new LinkedList<Integer>();

         // Mark the current node as visited and enqueue it
         visited[s]=true;
         queue.add(s);

         while(queue.size() != 0){
             // Dequeue a vertex from queue and print it
             s = queue.poll();
             System.out.print(s+" ");

             // Get all adjacent vertices of the dequeued vertex s
             // If a adjacent has not been visited, then mark it
             // visited and enqueue it
             Iterator<Integer> i = adj[s].listIterator();
             while(i.hasNext()){
                 int n = i.next();
                 if(!visited[n]){
                     visited[n] = true;
                     queue.add(n);
                 }
             }
         }
    }
www.geeksforgeeks.org
```

Video: https://goo.gl/xkE1yf

Tracing Program: https://goo.gl/wRFjub

2-Uniform-Cost –Search (UCS):

In some fields, artificial intelligence in particular, **Dijkstra's algorithm** or a variant of it, is known as **uniform cost search** and formulated as an instance of the more general idea of best-first search.
BUT, the difference between **uniform cost search** and **Dijkstra's algorithm** is that UCS has a **goal**

→ Visits the next node which has the least total cost from the root, until a goal state is reached.

- Similar to BREADTH-FIRST, but with an evaluation of the cost for each reachable node.
- g(n) = path cost(n) = sum of individual edge costs to reach the current node.

- ***Completeness:*** Yes (if b is finite, and step cost is positive)
- **Time Complexity:** much larger than b^d, and just b^d if all steps have the same cost.
- **Space Complexity:** as above
- ***Optimality:*** Yes

b: Branching Factor
d: the depth of the goal

→ Requires that the goal test being applied when a node is removed from the nodes list rather than when the node is first generated while its parent node is expanded.

Video: https://goo.gl/RfZaxb

Video: https://www.youtube.com/watch?v=0u78hx-66Xk

Java Code:

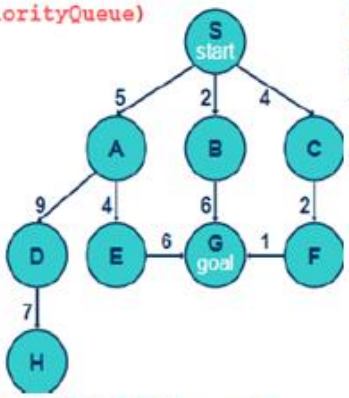
```
public static void UniformCostSearch(Node source, Node goal) {
	source.pathCost = 0;
	PriorityQueue<Node> queue = new PriorityQueue<Node>(20, new Comparator<Node>() {
	// override compare method
	public int compare(Node i, Node j) {
		if (i.pathCost > j.pathCost) {
			return 1;
		}
		else if (i.pathCost < j.pathCost) {
			return -1;
		}
		else {
			return 0;
		    }
		  }
		 }
		);
		queue.add(source);
		Set<Node> explored = new HashSet<Node>();
		boolean found = false;
		// while frontier is not empty
		do {Node current = queue.poll();
			explored.add(current);
			if (current.value.equals(goal.value)) {
				found = true;
			}for (Edge e : current.adjacencies) {
				Node child = e.target;
				double cost = e.cost;
				child.pathCost = current.pathCost + cost;
				if (!explored.contains(child) && !queue.contains(child)) {
					child.parent = current;
					queue.add(child);
					System.out.println(child);
					System.out.println(queue);
					System.out.println();
				  }
			      else if ((queue.contains(child)) && (child.pathCost > current.pathCost + cost)) {
					child.parent = current;
					child.pathCost = current.pathCost + cost;
					queue.remove(child);
					queue.add(child);
				}
			}
		} while (!queue.isEmpty());
	}
```

stackoverflow.com

generalSearch(problem, priorityQueue)

# of nodes tested: 0, expanded: 0

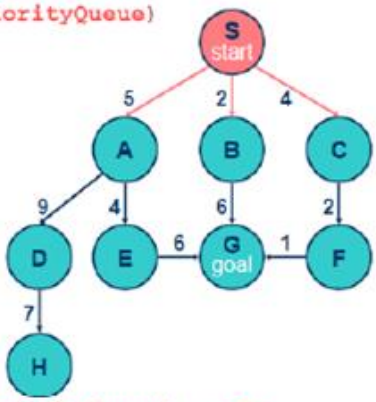
expnd. node	nodes list
	{S}



generalSearch(problem, priorityQueue)

# of nodes tested: 1, expanded: 1

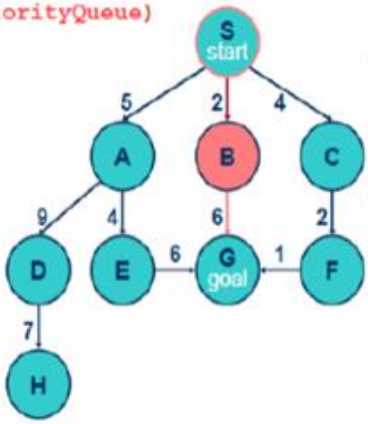
expnd. node	nodes list
	{S:0}
S not goal	{B:2,C:4,A:5}



generalSearch(problem, priorityQueue)

# of nodes tested: 2, expanded: 2

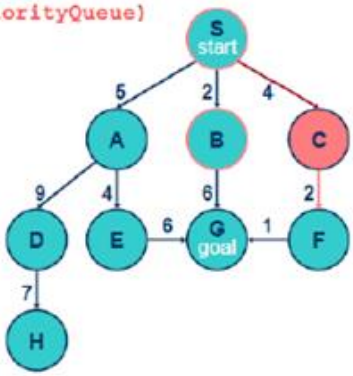
expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B not goal	{C:4,A:5,G:2+6}



generalSearch(problem, priorityQueue)

# of nodes tested: 3, expanded: 3

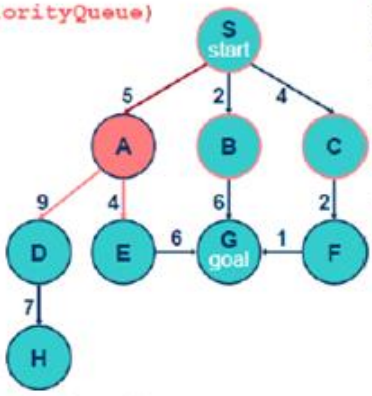
expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C not goal	{A:5,F:4+2,G:8}



generalSearch(problem, priorityQueue)

# of nodes tested: 4, expanded: 4

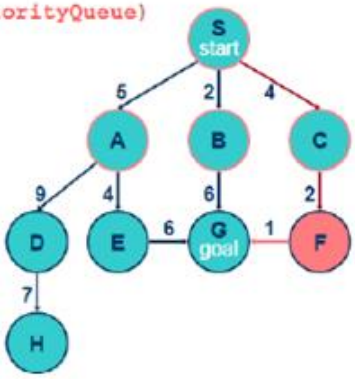
expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A not goal	{F:6,G:8,E:5+4, D:5+9}



generalSearch(problem, priorityQueue)

# of nodes tested: 5, expanded: 5

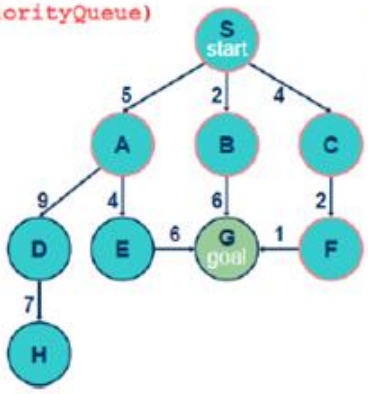
expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F not goal	{G:4+2+1,G:8,E:9, D:14}



generalSearch(problem, priorityQueue)

# of nodes tested: 6, expanded: 5

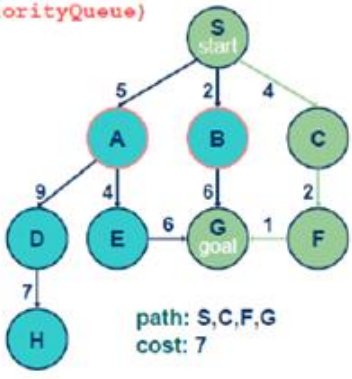
expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G goal	{G:8,E:9,D:14} no expand



generalSearch(problem, priorityQueue)

# of nodes tested: 6, expanded: 5

expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G	{G:8,E:9,D:14}



path: S,C,F,G
cost: 7

Breadth-First vs. Uniform-Cost

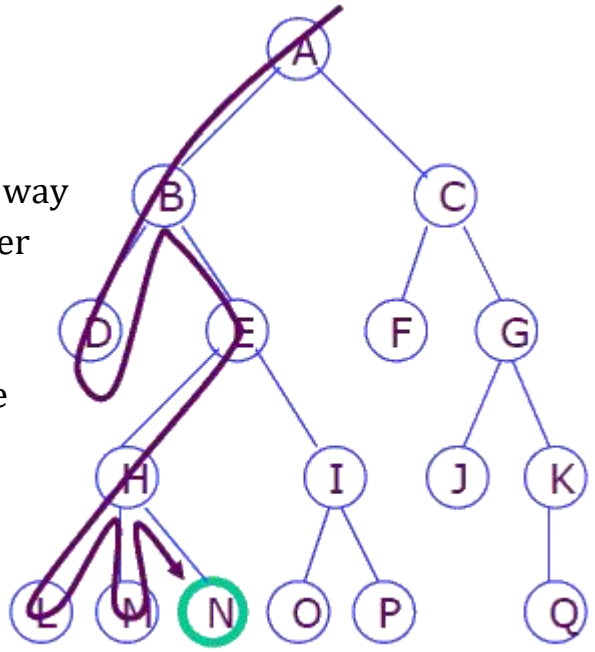
- Breadth-first search (BFS) is a special case of uniform-cost search when all edge costs are positive and identical.

- Breadth-first always expands the shallowest node
  - Only optimal if all step-costs are equal

- Uniform-cost considers the overall path cost
  - Optimal for any (reasonable) cost function
    - non-zero, positive
  - Gets stuck down in trees with many fruitless, short branches
    - low path cost, but no goal node

- Both are complete for non-extreme problems
  - Finite number of branches
  - Strictly positive search function

3-Depth-First Search (DFS):



- A depth-first-search (DFS) explores a path all the way to a leaf before backtracking and exploring another path.
- For example, after searching A, then B, then D, the search backtracks and tries another path from B.
- Node are explored in the order
  A B D E H L M N I O P C F G J K Q
  (root-left-right)

- ***Completeness:*** No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
  - complete in finite spaces
- **Time Complexity:** O(b^m): terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- **Space Complexity:** $O(bm), i.e., linear\ space$!
- ***Optimality:*** No.

b: maximum branching factor of the search tree
d: depth of the least-cost solution
m: maximum depth of the state space (may be ∞)

Depth-First-Search Algorithm:

Algorithm1:

```
Put the root node on a stack;
      while (stack is not empty) {
            remove a node from the stack;
        if (node is a goal node)
                  return success;
    put all children of node onto the stack;
 } return failure;
```

- At each step, the stack contains some nodes from each of a number of levels
  - The size of stack that is required depends on the branching factor b
  - While searching level n, the stack contains approximately (b-1)*n nodes
- When this method succeeds, it doesn't give the path

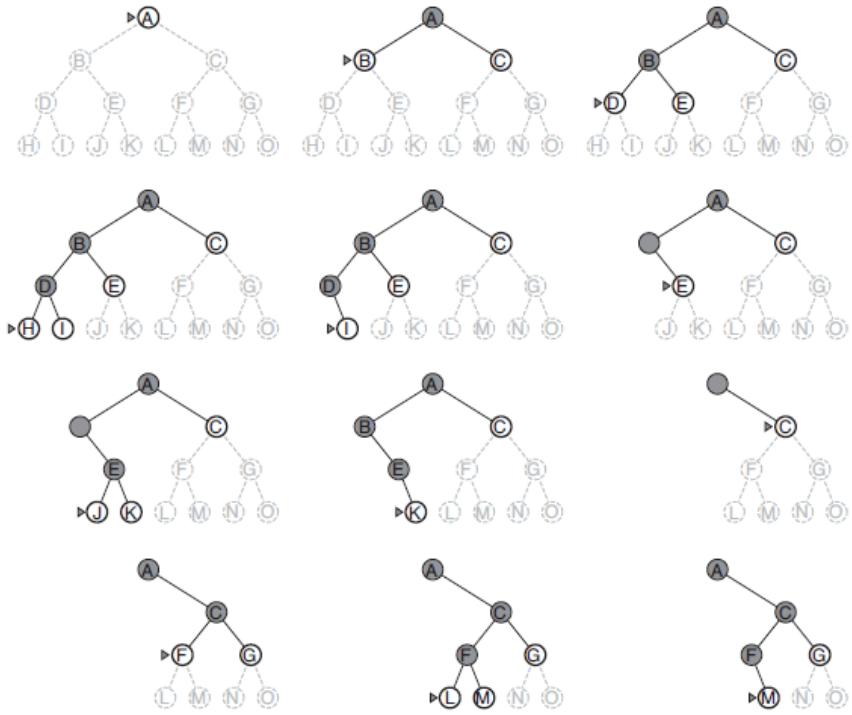
Algorithm2: **(recursive)**

```
Search (node);
   if node is a goal, return success;
   for each child c of node {
        if search(c) is successful, return success;
} return failure;
```

- The (implicit) stack contains only the nodes on a path from the root to a goal
  - The stack only needs to be large enough to hold the deepest search path
  - When a solution is found, the path is on the (implicit) stack, and can be extracted as the recursion "unwinds"

Video: https://goo.gl/kWrqNH

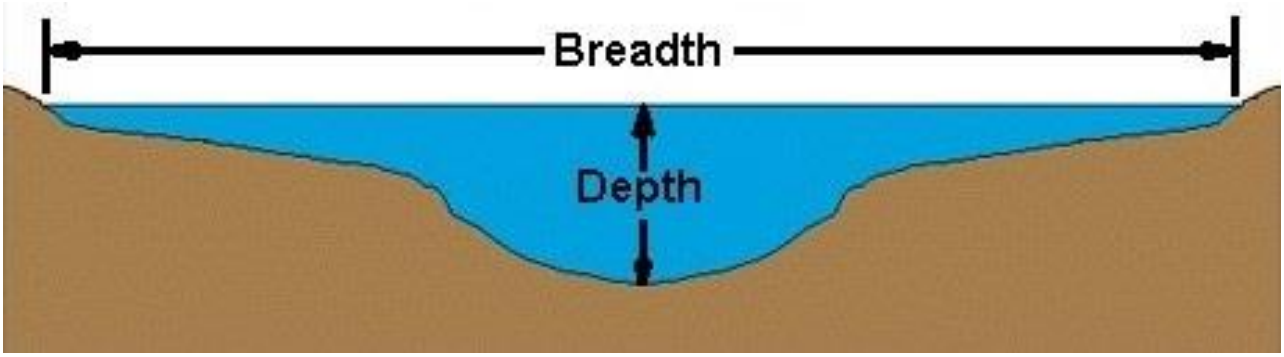
Example:



Java Code:

```
    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS()
    {
        // Mark all the vertices as not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        // starting from all vertices one by one
        for (int i=0; i<V; ++i)
            if (visited[i] == false)
                DFSUtil(i, visited);
    }
```

www.geeksforgeeks.org



Depth-First vs. Breadth-First

- Depth-first goes off into one branch until it reaches a leaf node
  - Not good if the goal is on another branch
  - Neither complete nor optimal
  - Uses much less space than breadth-first
    - Much fewer visited nodes to keep track, smaller fringe
- Breadth-first is more careful by checking all alternatives
  - Complete and optimal (Under most circumstances)
  - Very memory-intensive
- For a large tree, breadth-first search memory requirements maybe excessive
- For a large tree, a depth-first search may take an excessively long time to find even a very nearby goal node.

How can we combine the advantages (and avoid the disadvantages) of these two search techniques?

→ by Iterative-Deepening-Search

	Advantages	*Disadvantages*
Breadth-First-Search	- **will never get trapped exploring the useless path forever.** - **If there is a solution, BFS will definitely find it out.** - **If there is more than one solution then BFS can find the minimal one that requires less number of steps.**	- **its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional number of nodes stored, the space complexity of BFS is O(b^d). As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.** - **If the solution is farther away from the root, breath first search will consume lot of time.**
Depth-First-Search	**memory requirement is only linear with respect to the search graph**	- **there is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree** - **not guaranteed to find the solution** - **no guarantee to find a minimal solution, if more than one solution exists.**

4-Depth-limited Search (DLS):

Similar to Depth-First-Search, but with a limit

- i.e., nodes at depth / have no successors
- Overcomes problems with infinite paths
- Sometimes a depth limit can be inferred or estimated from the problem description
  - In other cases, a good depth limit is only known when the problem is solved
- must keep track of the depth

- ***Completeness:*** no (if goal beyond l (l<d), or infinite branch length)
- **Time Complexity:** b^L
- **Space Complexity:** b*L
- ***Optimality:*** No (if L<d).

b: Branching Factor
d: the depth of the goal
L: the depth Limit

Depth-limited Search Algorithm:

```
function IDDFS(root)
    for depth from 0 to ∞
        found ← DLS(root, depth)
        if found ≠ null
            return found

function DLS(node, depth)
    if depth = 0 and node is a goal
        return node
    if depth > 0
        for each child of node
            found ← DLS(child, depth−1)
            if found ≠ null
                return found
    return null
```

Java Code:

```
public void depthLimitedSearch(int adjacencyMatrix[][], int source) {
    int visited[] = new int[numberOfNodes + 1];
    int element, destination;
    int depth = 0;

        System.out.println(source + " at depth " + depth);
        stack.push(source);
         visited[source] = 1;
        depth = 0;

         while (!stack.isEmpty()){
             element = stack.peek();
             destination = element;
             while (destination <= numberOfNodes){
                if (depth < MAX_DEPTH){
                  if (adjacencyMatrix[element][destination] == 1 && visited[destination] == 0){
                            stack.push(destination);
                            visited[destination] = 1;
                            depth++;
                            System.out.println(destination + " at depth " + depth);
                            element = destination;
                            destination = 1;
                        }
                    }
                    else{
                        return;
                    }
                    destination++;
                }
                stack.pop();
                depth--;
            }
        }
http://www.sanfoundry.com
```

Video: https://goo.gl/ufJNxM

5-Iterative-Deepening-Search(IDS):

→Applies **LIMITED-DEPTH** with increasing depth limits

- Combines advantages of BREADTH-FIRST and DEPTH-FIRST
- It searches to depth 0 (root only), then if that fails it searches to depth 1, then depth 2, etc.

Iterative-Deepening-Search Algorithm:

function ITERATIVE-DEEPENING-SEARCH(problem) **returns** a solution, or failure
for depth = 0 **to**∞ **do**
result ←DEPTH-LIMITED-SEARCH(problem, depth)
if result ≠ cutoff **then return** result

Java Code:

```
public void iterativeDeeping(int adjacencyMatrix[][], int destination){
  numberOfNodes = adjacencyMatrix[1].length - 1;
   while (!goalFound){
    depthLimitedSearch(adjacencyMatrix, 1, destination);
    maxDepth++;
   }
  System.out.println("nGoal Found at depth " + depth);
 }
```

http://code-library.net/

Video: https://goo.gl/CzRhmt

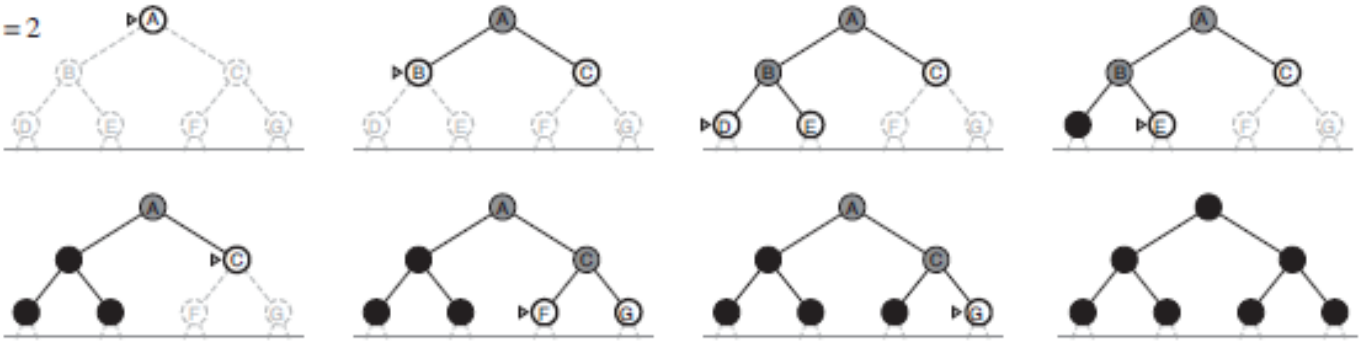
Tracing Program: https://goo.gl/T82Usf

Example:

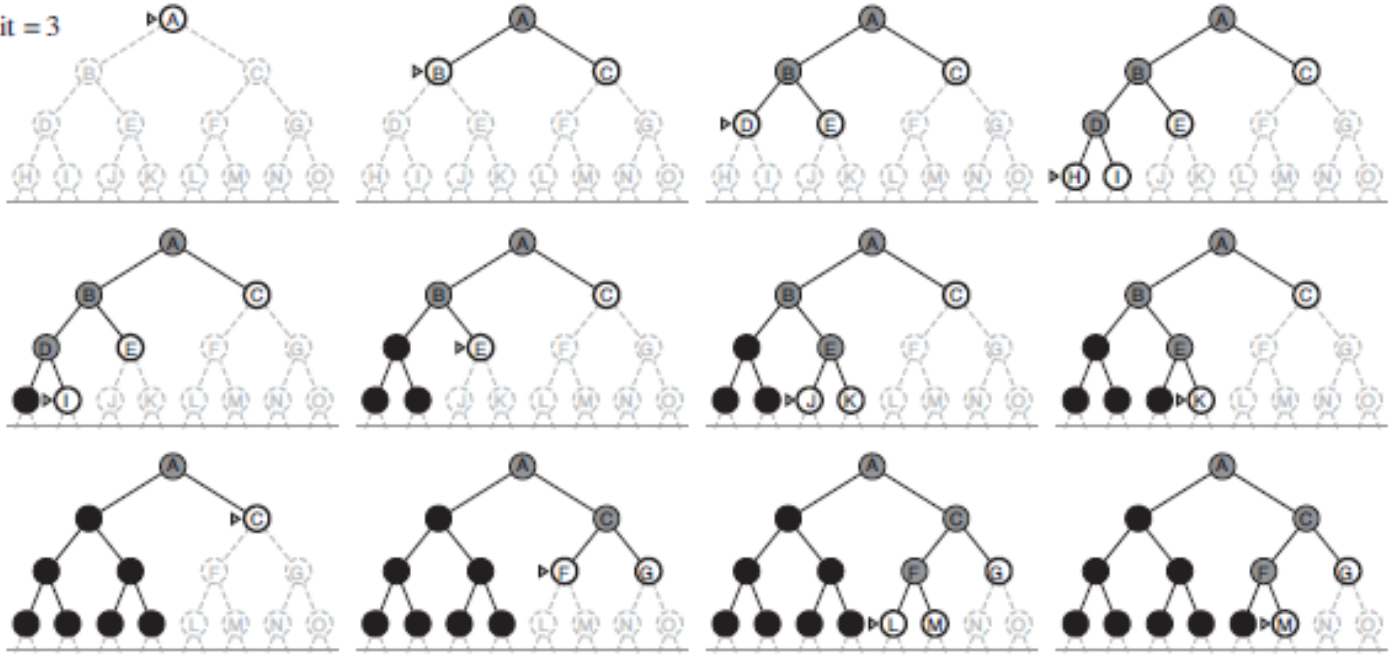
Limit = 0

Limit = 1

Limit = 2



Limit = 3



Iterative Deepening Depth-First Search (IDDFS)

- If a goal node is found, it is a nearest node and the path to it is on the stack.
- Required stack size is limit of search depth (plus 1).
- Many states are expanded multiple times
- doesn't really matter because the number of those nodes is small
- In practice, one of the best uninformed search methods
- for large search spaces, unknown depth

- ***Completeness:*** Yes (if b is finite)
- **Time Complexity:** (d+1)b^0+ d b^1+ (d-1) b^2+ …… + b^d= O(b^d)
- **Space Complexity:** O(b*d)
- ***Optimality:*** Yes, if step cost =1

b: Branching Factor
d: the depth of the goal

- The nodes in the bottom level (level d) are generated once, those on the next bottom level are generated twice, and so on:

  N_{IDS}= (d)b + (d-1) b^2+ … + (1) b^d

  Time complexity = b^d
- Compared with BFS:

  N_{BFS}= b + b^2 … + b^d+ (b^{d+1} –b)
- Suppose b = 10, d = 5,

  N_{IDS} = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

  N_{BFS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111

➔ IDS behaves better in case the search space is large and the depth of goal is unknown.

- When searching a binary tree to depth 7:
  - DFS requires searching 255 nodes
  - Iterative deepening requires searching 502 nodes
  - Iterative deepening takes only about twice as long
- When searching a tree with branching factor of 4 (each node may have four children):
  - DFS requires searching 21845 nodes
  - Iterative deepening requires searching 29124 nodes
  - Iterative deepening takes about 4/3 = 1.33 times as long
- The higher the branching factor, the lower the relative cost of iterative deepening depth first search

6-Bi-directional-Search (BDS):

- Search simultaneously from two directions
  - Forward from the initial and backward from the goal state, until they meet in the middle (i.e., if a node exists in the fringe of the other).
  - The idea is to have ($b^{d/2}$+ $b^{d/2}$) instead of bd, which much less

- May lead to substantial savings (if it is applicable), but is has several limitations
  - Predecessors must be generated, which is not always possible
  - Search must be coordinated between the two searches
  - One search must keep all nodes in memory

- ***Completeness:*** Yes (if b is finite, breadth-first for both directions)

- **Time Complexity:** $b^{d/2}$

- **Space Complexity:** $b^{d/2}$

b: Branching Factor
d: tree depth

- ***Optimality:*** yes (all step costs identical, breadth-first for both directions)

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.

- Variety of uninformed search strategies

Criterion	Breadth-First	Depth-First	Depth-Limited	Uniform-Cost	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes	No	No	Yes	Yes	Yes
Time	$O(b^d)$	$O(b^m)$	$O(b^L)$	$O(b^1 + [C^*/€)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(bm)$	O(bL)	$O(b^1 + [C^*/€)$	O(bd)	$O(b^{d/2})$
Optimal?	Yes	No	No	Yes	Yes	Yes

→Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

- Breadth-first search (BFS) and depth-first search (DFS) are the foundation for all other search techniques.
  - We might have a weighted tree, in which the edges connecting a node to its children have differing "weights"

- We might therefore look for a "least cost" goal

- The searches we have been doing are blind searches, in which we have no prior information to help guide the search

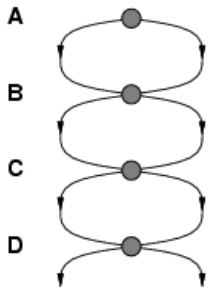
Summary (When to use what)

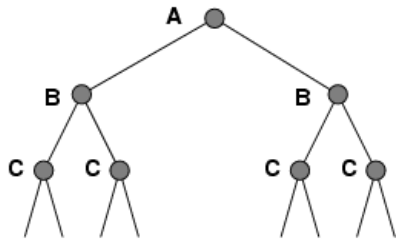
- **Breadth-First Search:**
  - Some solutions are known to be shallow
- **Uniform-Cost Search:**
  - Actions have varying costs
  - Least cost solution is the required

  This is the only uninformed search that worries about costs.
- **Depth-First Search:**
  - Many solutions exist
  - Know (or have a good estimate of) the depth of solution
- **Iterative-Deepening Search:**
  - Space is limited and the shortest solution path is required

Improving Search Methods

- Make algorithms more efficient
  - avoiding repeated states





- Use additional knowledge about the problem
  - properties ("shape") of the search space
    - more interesting areas are investigated first
  - pruning of irrelevant areas
    - areas that are guaranteed not to contain a solution can be discarded.

Informed Search – Chapter 4

• **Informed search** (Heuristic search) methods may have access to a **heuristic** function h(n) that estimates the cost of a solution from n.

• Agent has background information about the problem

– map, costs of actions

The basic algorithms are as follows:

- The generic **best-first search** algorithm selects a node for expansion according to an **evaluation function**.

- **Greedy best-first search** expands nodes with minimal h(n). It is not optimal but is often efficient.

- **A∗ search** expands nodes with minimal f(n) = g(n) + h(n). A∗ is complete and optimal, provided that h(n) is admissible (for TREE-SEARCH) or consistent (for GRAPH-SEARCH). The space complexity of A∗ is still prohibitive.

- **RBFS** (recursive best-first search) and **SMA**∗ (simplified memory-bounded A∗)
- are robust, optimal search algorithms that use limited amounts of memory; given enough time, they can solve problems that A∗ cannot solve because it runs out of memory.

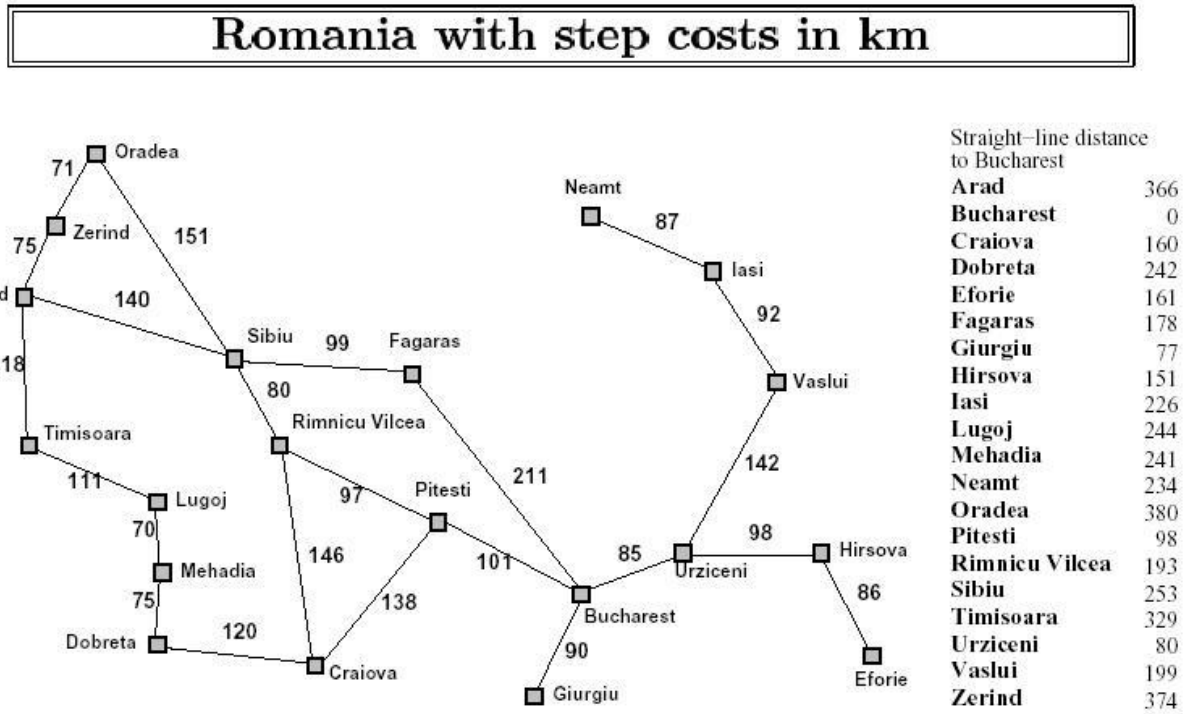
• The performance of heuristic search algorithms depends on the quality of the heuristic function. One can sometimes construct good heuristics by relaxing the problem definition, by storing precomputed solution costs for sub-problems in a pattern database, or by learning from experience with the problem class.

1-Best-First search (BFS):

- Idea: use an evaluation function f(n) for each node
  - family of search methods with various evaluation functions (estimate of "desirability")
  - usually gives an estimate of the distance to the goal
  - often referred to as heuristics in this context

  →Expand most desirable unexpanded node.

  →A heuristic function ranks alternatives at each branching step based on the available information (heuristically) in order to make a decision about which branch to follow during a search.
- Implementation:
  Order the nodes in fringe in decreasing order of desirability.
- Special cases:
  - greedy best-first search
  - A*search



Greedy best-first search

• Greedy best-first search expands the node that appears to be closest to goal.

• Estimate of cost from n to goal, e.g., hSLD(n) = straight-line distance from n to Bucharest.

Utilizes a heuristic function as evaluation function

– f (n) = h (n) = estimated cost from the current node to a goal.

– Heuristic functions are problem-specific.

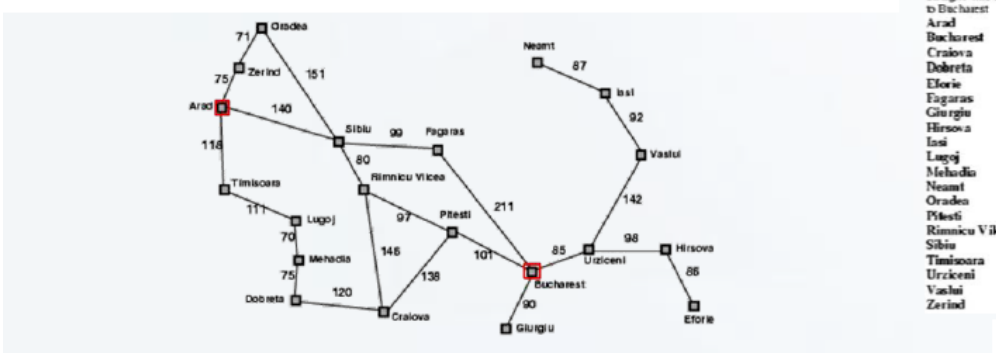
– Often straight-line distance for route-finding and similar problems.

– Often better than depth-first, although worst-time complexities are equal or worse (space).

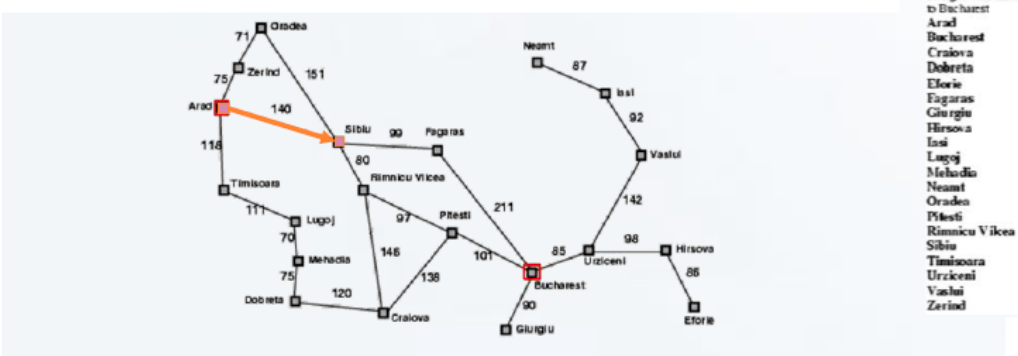
Algorithm:

```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

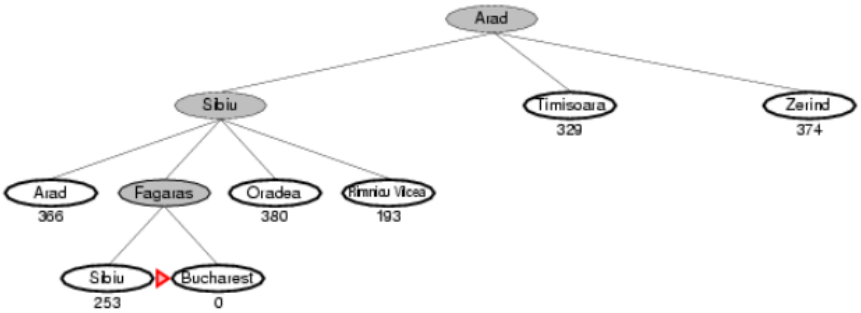
Example:

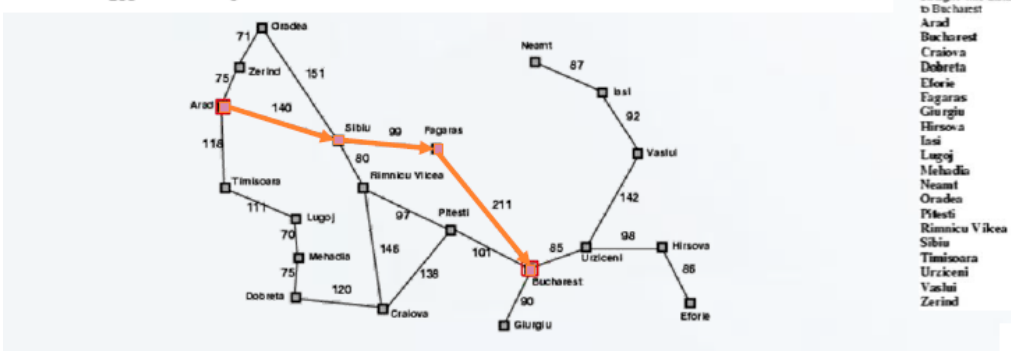


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

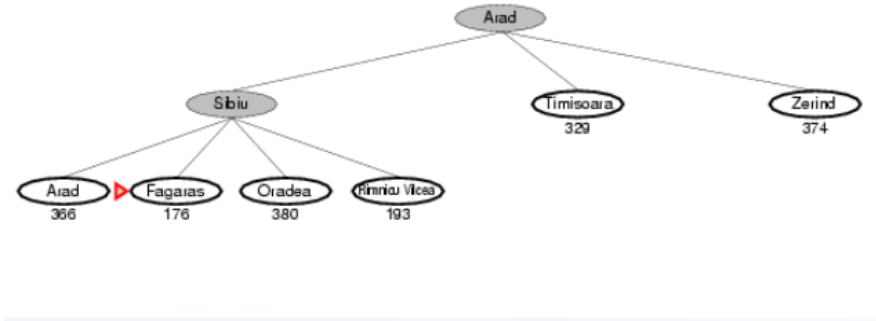


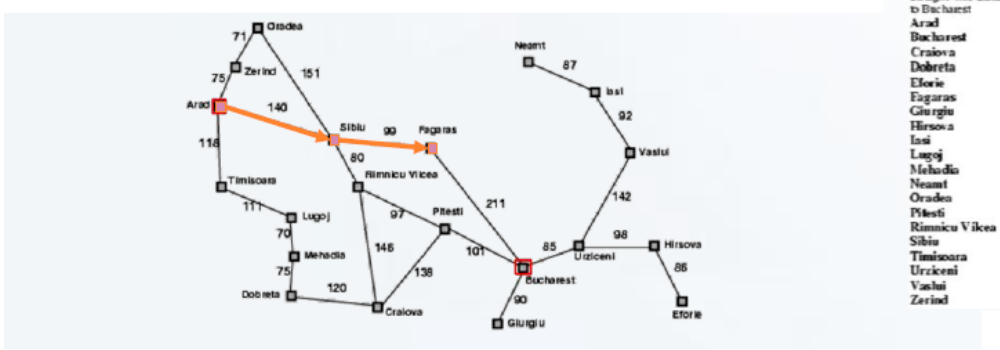
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374





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





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

Properties of greedy best-first search

Complete: No – can get stuck in loops (e.g., Iasi → Neamt → Iasi → Neamt → ....)

Time: O(b^m), but a good heuristic can give significant improvement

Space: O(b^m), -- keeps all nodes in memory

Optimal: No

b: Branching Factor
m: maximum depth of the source tree

A search*

Idea: avoid expanding paths that are already expensive.

Evaluation function = path cost + estimated cost to the goal

$$f(n) = g(n) + h(n)$$

- g(n) = cost so far to reach n
- h(n) = estimated cost from n to goal
- f(n) = estimated total cost of path through n to goal

Combines greedy and uniform-cost search to find the (estimated) cheapest path through the current node

– Heuristics must be admissible

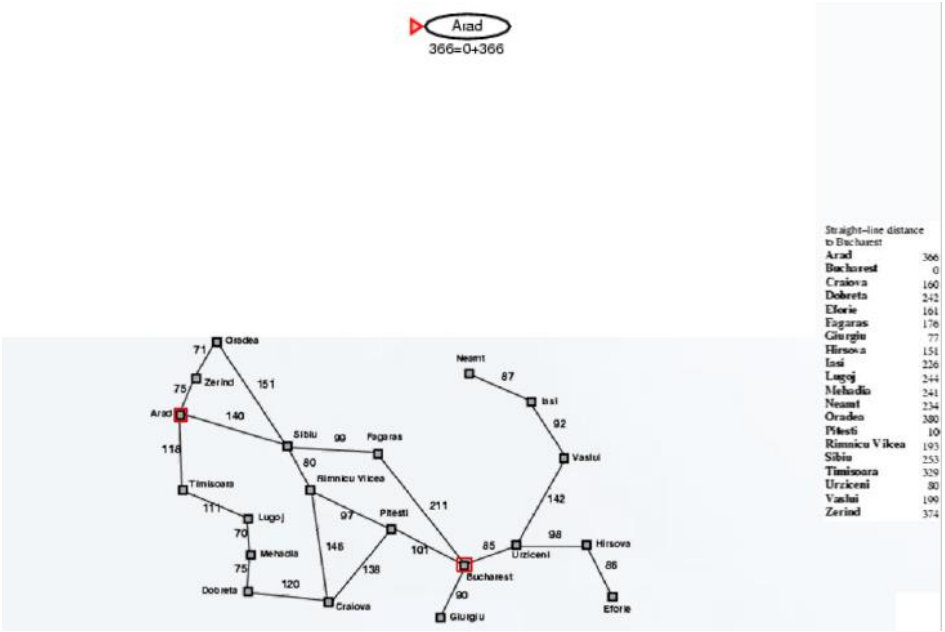
- Never overestimate the cost to reach the goal

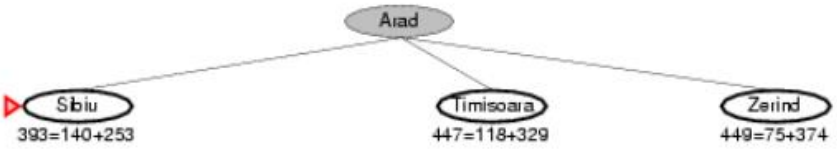
– Very good search method, but with complexity problems

Algorithm:

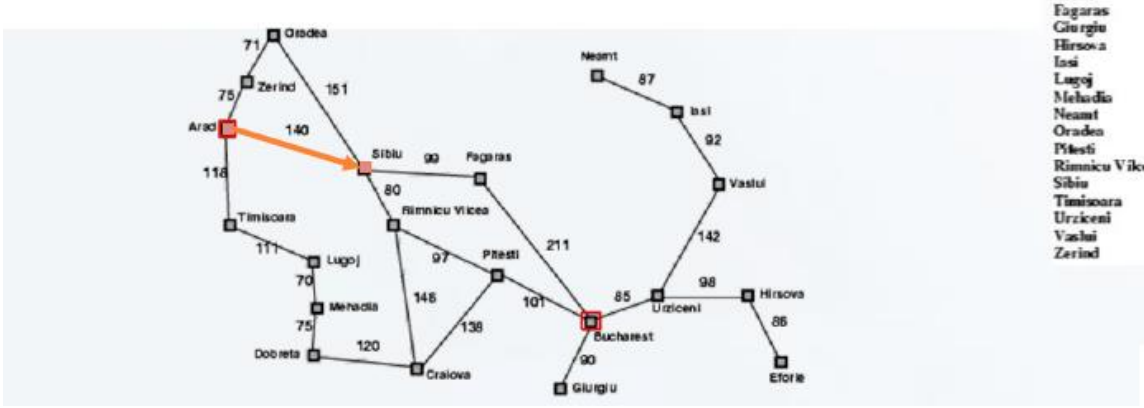
```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

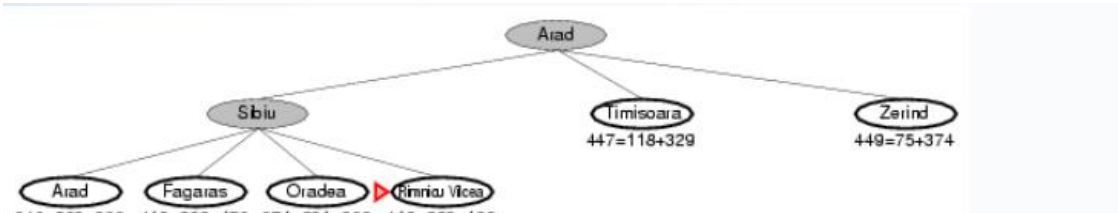
Example:



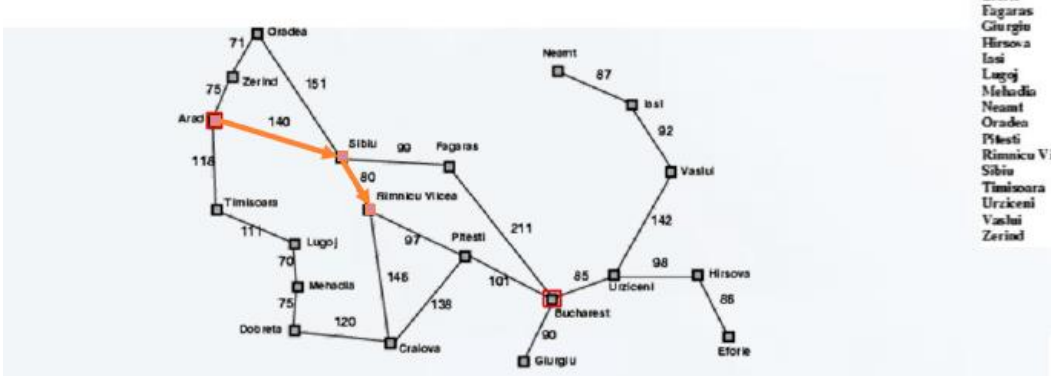


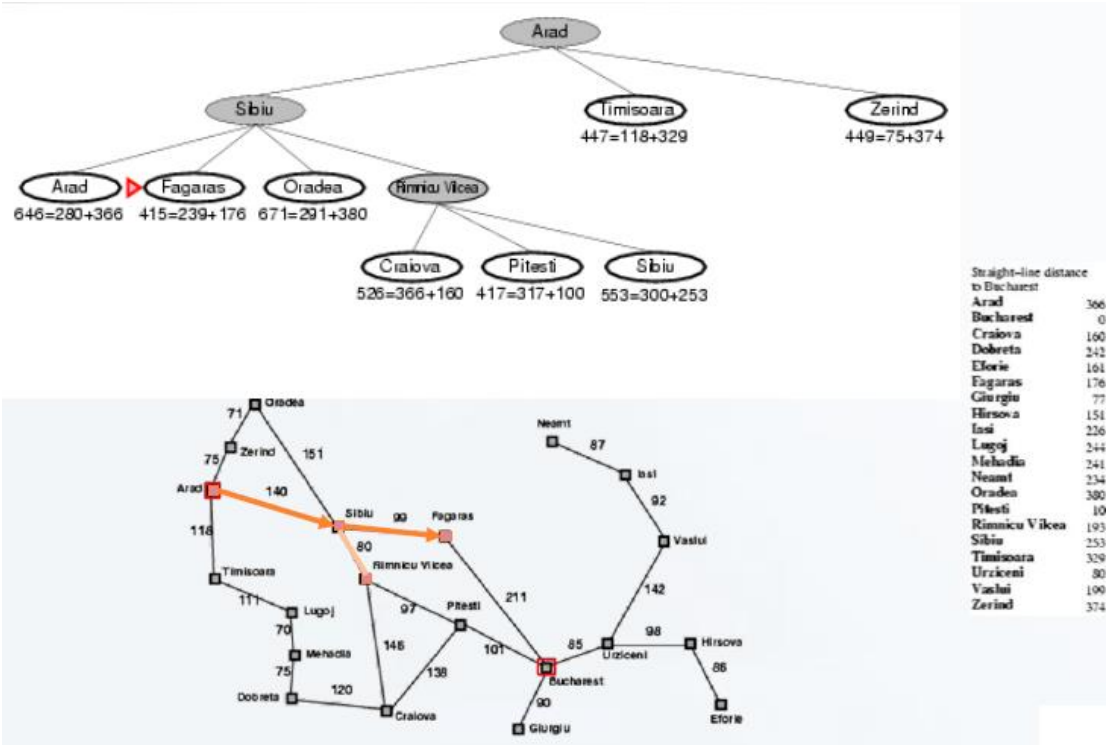
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

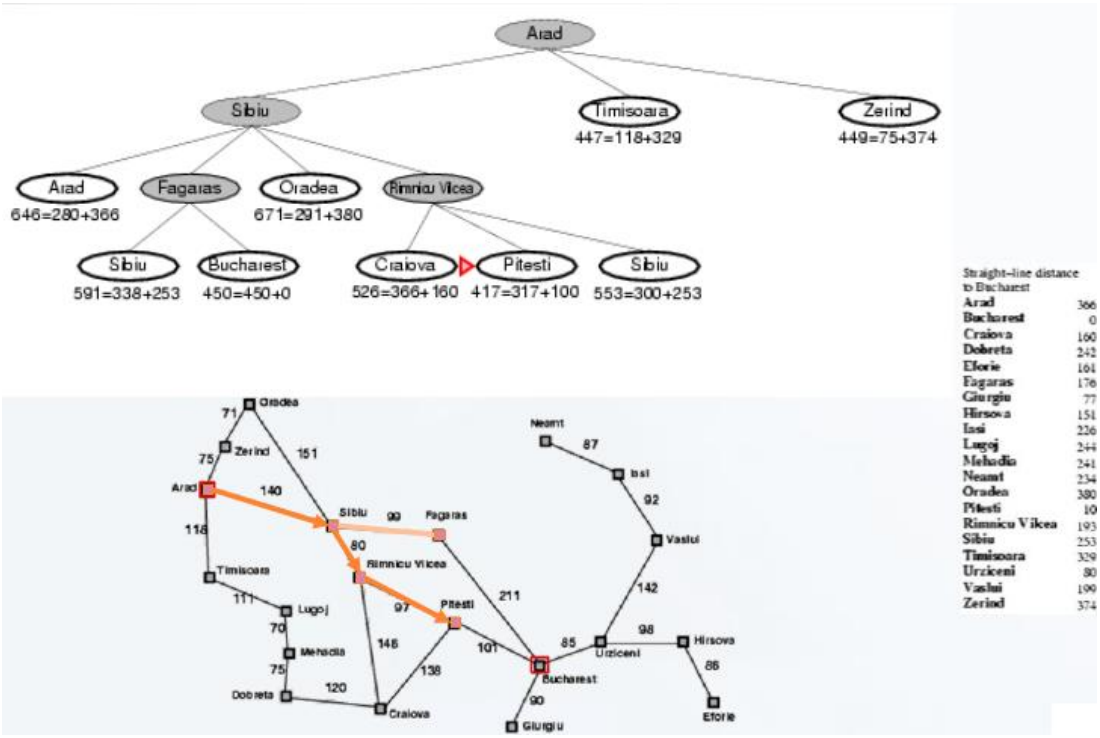


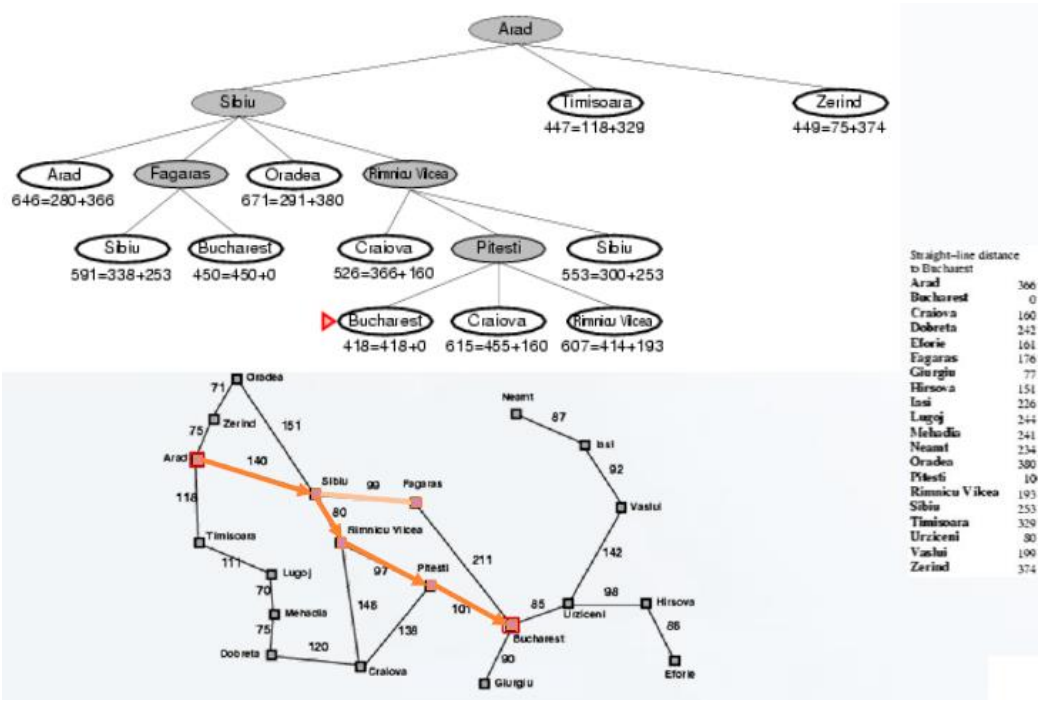


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374









Admissible Heuristics

A heuristic h(n) is admissible if for every node n, h(n) ≤ h*(n), where h*(n) is the true cost to reach the goal state from n.

An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic.

The heuristic function hSLD(n) is admissible because it never overestimates the actual road distance)

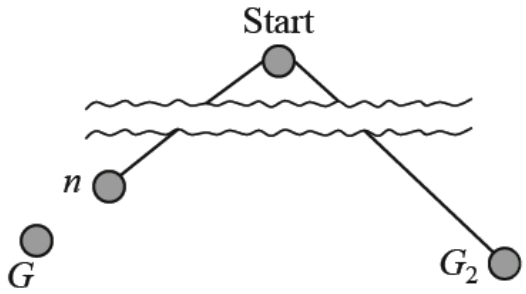
Theorem-1: If h(n) is admissible, A* using TREE-SEARCH is optimal.

Optimality of A (proof)*

Recall that f(n) = g(n) + h(n)

Now, suppose some suboptimal goal G2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.

We want to prove:



f(n) < f(G2)

(Then A* will prefer n over G2)

f(G2) = g(G2) since h(G2) = 0

g(G2) > g(G) since G2 is suboptimal

f(G) = g(G) since h(G) = 0

Then f(G2) > f(G) from above

h(n) ≤ h*(n) since h is admissible

g(n) + h(n) ≤ g(n) + h*(n)

Then f(n) ≤ f(G)

Thus, A* will never select G2 for expansion

In other words:

F (G2) = g(G2) + h(G2) = g(G2) > C*,
since G2 is a goal on a non-optimal path (C* is the optimal cost)

f (n) = g(n) + h(n) ≤ C*, since h is admissible

f (n) ≤ C* < f (G2), so G2 will never be expanded

→A* will not expand goals on sub-optimal paths

*Properties of A**

- **Complete:** Yes,
  unless there are infinitely many nodes with f ≤ f (G)
- **Time:** Exponential,
  because all nodes such that f (n) ≤ C* are expanded!
- **Space:** Keeps all nodes in memory fringe is exponentially large
- **Optimal:** Yes

Memory Bounded Heuristic Search

How can we solve the memory problem for A* search?

Idea: Try something like Iterative Deeping Search, but the cutoff is f-cost ($g+h$) at each iteration, rather than depth first.

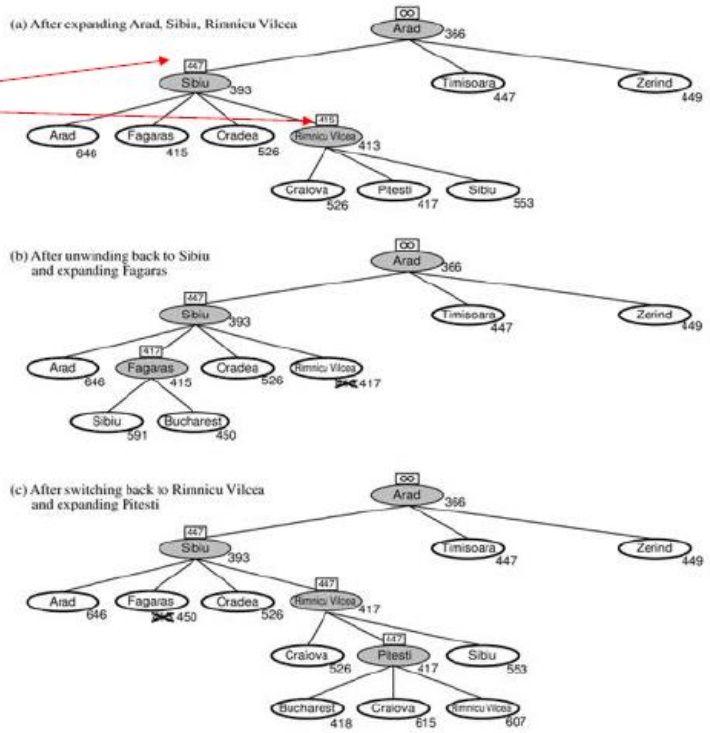
Two types of memory bounded heuristic searches:

→ Recursive BFS

→SMA*

Recursive Best First Search (RBFS)

best alternative over fringe nodes, which are not childrem: do i want to back up?



RBFS changes its mind very often in practice.

This is because the *f=g+h* become more accurate (less optimistic) as we approach the goal. Hence, higher level nodes have smaller f-values and will be explored first.

Problem? If we have more memory we cannot make use of it.

Ay idea to improve this?

→ Simple Memory Bounded A* (SMA*)

Simple Memory Bounded A (SMA*)*

- This is like A*, but when memory is full we delete the worst node (largest f-value).
- Like RBFS, we remember the best descendent in the branch we delete.
- If there is a tie (equal f-values) we first delete the oldest nodes first.
- SMA* finds the optimal reachable solution given the memory constraint.
- But time can still be exponential.

SMA pseudocode*

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost
  Queue ← MAKE-QUEUE ({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
   if Queue is empty then return failure
       n ← deepest least-f-cost node in Queue
   if GOAL-TEST(n) then return success
       s ← NEXT-SUCCESSOR(n)
   if s is not a goal and is at maximum depth then
       f(s) ←∞
   else
      f(s) ← MAX(f(n),g(s)+h(s))
   if all of n's successors have been generated then
      update n's f-cost and those of its ancestors if necessary
   if SUCCESSORS(n) all in memory then remove n from Queue
   if memory is full then
     delete shallowest, highest-f-cost node in Queue
     remove it from its parent's successor list
     insert its parent on Queue if necessary
   insert s in Queue
end
```

SMA* is a shortest path algorithm based on the A* algorithm.

The advantage of SMA* is that it uses a bounded memory, while the A* algorithm might need exponential memory.

All other characteristics of SMA* are inherited from A*.

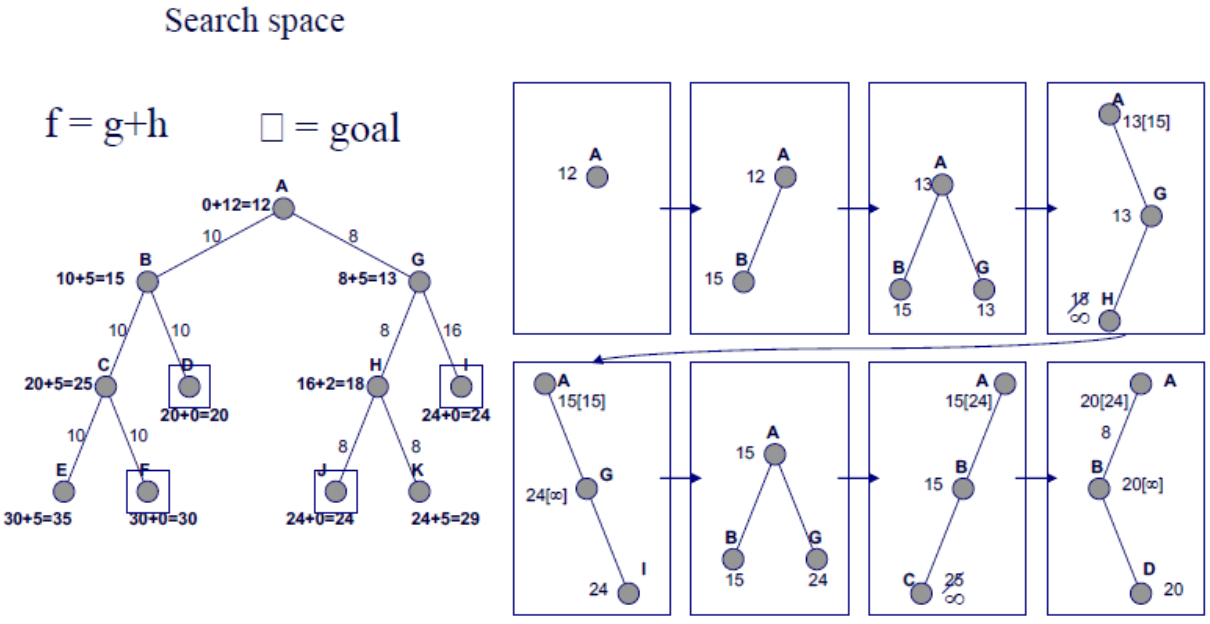
How it works:

- Like A*, it expands the best leaf until memory is full.
- Drops the worst leaf node- the one with the highest f-value.
- Like RBFS, SMA* then backs up the value of the forgotten node to its parent.

Example with 3-node memory:

Progress of SMA*:

Each node is labeled with its current f-cost. Values in parentheses show the value of the best forgotten descendant.



∞ is given to nodes that the path up to it uses all available memory.

Can tell when best solution found within memory constraint is optimal or not.

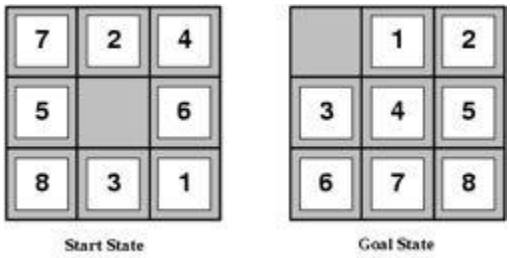
The Algorithm proceeds as follows:

1. At each stage, one successor is added to the deepest lowest-f-cost node that has some successors not currently in the tree. The left child B is added to the root A.
2. Now $f(A)$ is still 12, so we add the right child G (f = 13). Now that we have seen all the children of A, we can update its f-cost to the minimum of its children, that is, 13. The memory is now full.
3. G is now designated for expansion, but we must first drop a node to make room. We drop the shallowest highest-f-cost leaf, that is, B. When we have done this, we note that A's best forgotten descendant has f = 15, as shown in parentheses. We then add H, with $f(H)$ = 18. Unfortunately, H is not a goal node, but the path to H uses up all the available memory. Hence, there is no way to find a solution through H, so we set $f(H) = \infty$.
4. G is expanded again. We drop H, and add I, with $f(I)$ = 24. Now we have seen both successors of G, with values of ∞ and 24, so $f(G)$ becomes 24. $f(A)$ becomes 15, the minimum of 15 (forgotten successor value) and 24. Notice that I is a goal node, but it might not be the best solution because A's f-cost is only 15.
5. A is once again the most promising node, so B is generated for the second time. We have found that the path through G was not so great after all.
6. C, the first successor of B, is a nongoal node at the maximum depth, so $f(C) = \infty$.
7. To look at the second successor, D, we first drop C. Then $f(D)$ = 20, and this value is inherited by B and A.
8. Now the deepest, lowest-f-cost node is D. D is therefore selected, and because it is a goal node, the search terminates.

SMA Properties*

- It works with a heuristic, just as A*
- It is complete if the allowed memory is high enough to store the shallowest solution.
- It is optimal if the allowed memory is high enough to store the shallowest optimal solution, otherwise it will return the best solution that fits in the allowed memory.
- It avoids repeated states as long as the memory bound allows it
- It will use all memory available.
- Enlarging the memory bound of the algorithm will only speed up the calculation.
- When enough memory is available to contain the entire search tree, then calculation has an optimal speed

Admissible heuristics



E.g., for the 8-puzzle:

h1 (n) = number of misplaced tiles

h2 (n) = total Manhattan distance (i.e., no. of squares from desired location of each tile)

h1(S) = 8

h2(S) = 3+1+2+2+2+3+3+2 = 18

Dominance

If h2 (n) ≥ h1(n) for all n, and both are admissible. then h2 dominates h1

h2 is better for search: it is guaranteed to expand less nodes.

Typical search costs (average number of nodes expanded):

d=12
IDS = 3,644,035 nodes
A*(h_1) = 227 nodes
A*(h_2) = 73 nodes

d=24
IDS = too many nodes
A*(h_1) = 39,135 nodes
A*(h_2) = 1,641 nodes

What to do If we have $h_1 \ldots h_m$, but none dominates the other?

→ h(n) = max{h_1 (n), . . . h_m (n)}

Relaxed Problems

A problem with fewer restrictions on the actions is called a **relaxed problem**.

The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.

If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**, then h_1 (n) gives the shortest solution.

If the rules are relaxed so that a tile can move to **any near square**, then h_2 (n) gives the shortest solution.

Admissible Heuristics

How can you invent a good admissible heuristic function?

→Try to relax the problem, from which an optimal solution can be found easily.

→ Learn from experience.

→Can machines invite an admissible heuristic automatically?

References:

1. **S. Russell and P. Norvig: Artificial Intelligence: A Modern Approach Prentice Hall, Third Edition**
2. **Dr. Mustafa Jarrar's Slides & Videos: http://www.jarrar.info/courses/AI/**
3. **Wikipedia: https://en.wikipedia.org/wiki/Artificial_intelligence**
   http://en.wikipedia.org/wiki/SMA*
4. **Geeks For Geeks: https://www.geeksforgeeks.org**
5. **Stack overflow: https://stackoverflow.com**
6. **http://en.wikipedia.org/wiki/SMA***
7. **Moonis Ali: Lecture Notes on Artificial Intelligence: http://cs.txstate.edu/~ma04/files/CS5346/SMA%20search.pdf**
8. **Max Welling: Lecture Notes on Artificial Intelligence: https://www.ics.uci.edu/~welling/teaching/ICS175winter12/A-starSearch.pdf**
9. **Kathleen McKeown: Lecture Notes on Artificial Intelligence: http://www.cs.columbia.edu/~kathy/cs4701/documents/InformedSearch-AR-print.ppt**
10. **Franz Kurfess: Lecture Notes on Artificial Intelligence: http://users.csc.calpoly.edu/~fkurfess/Courses/Artificial-Intelligence/F09/Slides/3-Search.ppt**