CHAPTER 3
SOLVING PROBLEMS BY SEARCHING

Presented by: Dr. R. Bentrcia
Department of Computer Science
University of M'sila

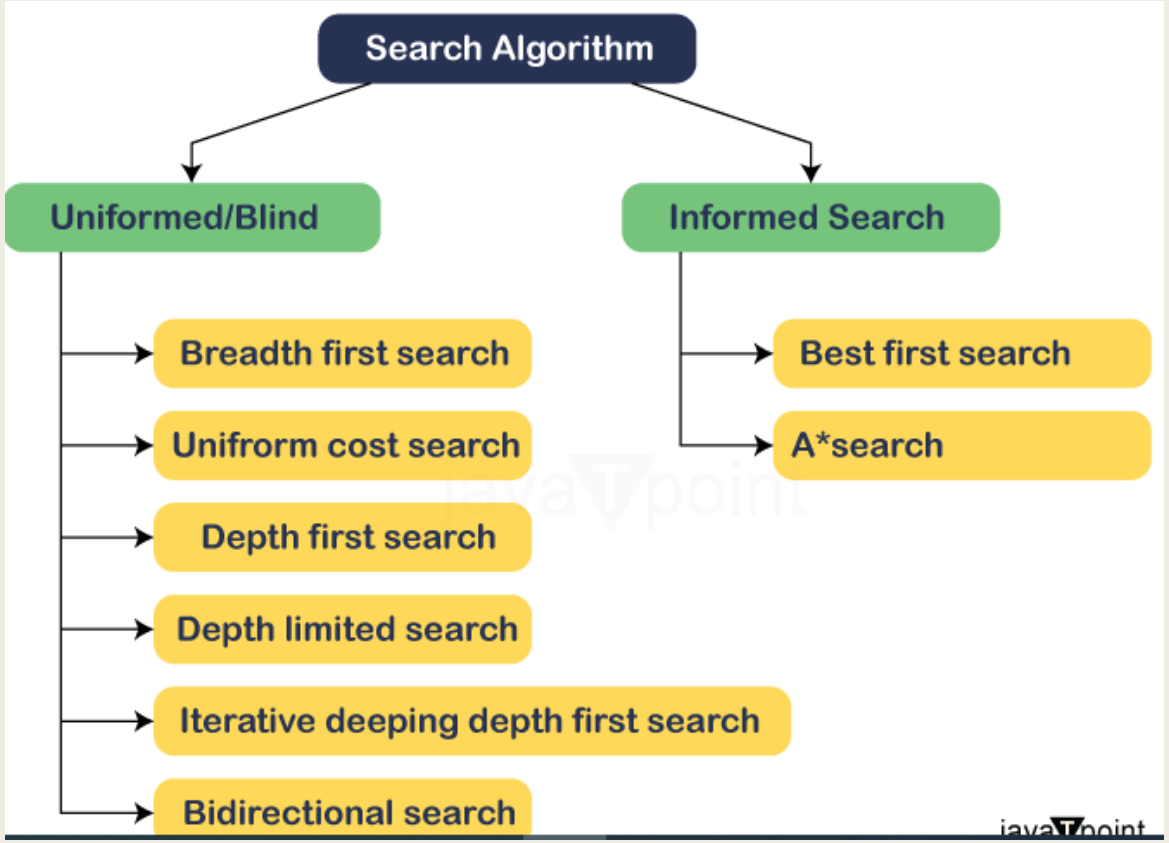
Outline

- Overview
- Tree Search and Graph Search Algorithms
- Problems Complexity
- Breadth-First Search
- Depth First Search
- Uninform Cost Search
- Best First Search
- A* Search

Overview

- Problem solving may be characterized as a systematic **search** through a range of possible actions to reach some predefined goal or solution.
- Problem space is represented by a graph or a tree
- A search algorithm takes a problem as input and returns a solution in form of an action sequence
- It can be divided into two types:
  - *uninformed search (Blind search)*
  - *informed search (Heuristic search)*

Overview



Source: https://www.javatpoint.com/search-algorithms-in-ai

Overview

- The uninformed search does not contain any domain knowledge such as the location of the goal state
- It examines each node of the tree/ graph until it achieves the goal node.
- The search operation begins from the initial state and providing all possible next steps arrangement until goal is reached. These are mostly the simplest search strategies, but they may not be suitable for complex paths
- In an informed search, problem information is available which can guide the search
- Guaranteed to find a good solution in reasonable time but not the best solution.
- Informed search can solve much complex problem

Tree Search and Graph Search Algorithms

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

Problems Complexity

- The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer,
- P class
  - It stands for **Polynomial Time.** It is the collection of decision that can be solved by a deterministic machine in polynomial time.
  - The polynomial time algorithm class P are algorithms that on inputs of size n have a worst case running time of O (n^k) for some constant k.
  - Easily solvable in polynomial time such as sorting numbers, greatest common divisor, prime numbers
- NP class
  - NP class stands for **Non-deterministic Polynomial Time.** It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Problems Complexity

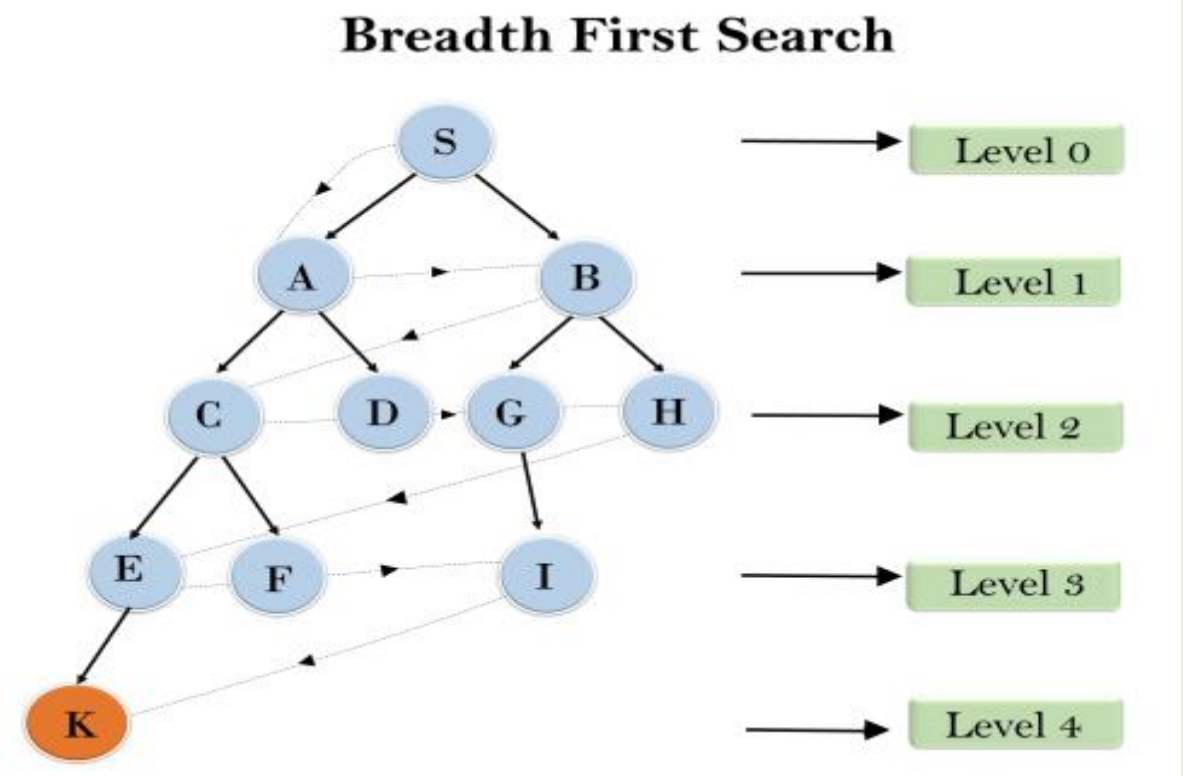
- A problem is called NP (nondeterministic polynomial) if its solution can be verified in polynomial time; nondeterministic means that no particular rule is followed to find the solution.
- Answers can be checked in polynomial time like traveling-salesman problem

- EXP class
  - An algorithm has exponential complexity if its resource usage can be expressed as an exponential function of the input size, typically $O(2^n)$ or $O(c^n)$ for some constant $c>1$,
  - Exponential complexity refers to a computational problem that grows rapidly as the input size increases, leading to a significant increase in the time or resources required to solve it.
  - It includes many problems that are too difficult to solve in practice, such as finding the perfect game for Chess,

Breadth-First Search BFS

- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.
- **Initialization:** Enqueue the given source node into a queue and mark it as visited.
- **Exploration:** While the queue is not empty:
  - Dequeue a node from the queue and visit it (e.g., print its value).
  - For each unvisited neighbor of the dequeued node:
    - Enqueue the neighbor into the queue.
    - Mark the neighbor as visited.
- **Termination:** Repeat step 2 until the queue is empty.

Breadth-First Search BFS



Source: https://www.javatpoint.com/ai-uninformed-search-algorithms

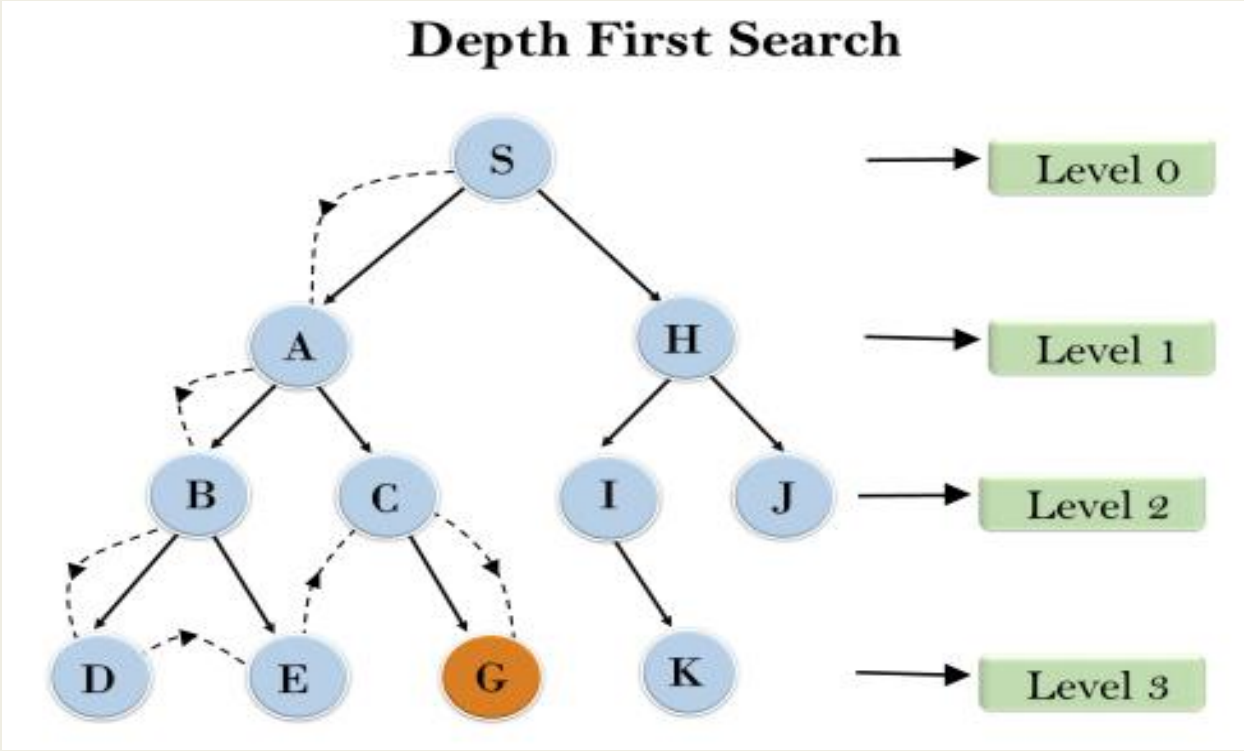
Breadth-First Search BFS

- BFS is complete (always finds goal if one exists)
- BFS finds the shallowest path to any goal node. If multiple goal nodes exist, BFS finds the shortest path
- If tree/graph edges have **weights**, BFS does not find the shortest length path.
- Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest node
- It is **O(n+e)**, where n is the number of nodes and e is the number of edges in the graph.
- The space complexity of the DFS algorithm is O(n)

Depth First Search DFS

- It is a recursive algorithm to search all the nodes of a tree data structure or a graph
- The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.
- Stack data structure can be used to implement the DFS algorithm
- First, create a stack with the total number of vertices in the graph.
- Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
- After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
- Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
- If no vertex is left, go back and pop a vertex from the stack.
- Repeat steps 2, 3, and 4 until the stack is empty.

Depth First Search DFS



Source: https://www.javatpoint.com/ai-uninformed-search-algorithms

Depth First Search DFS

- DFS search algorithm is complete
- The time complexity of the DFS algorithm is **O(n+e)**, where n is the number of nodes and e is the number of edges in the graph.
- The space complexity of the DFS algorithm is O(n)
- DFS search algorithm is non-optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high

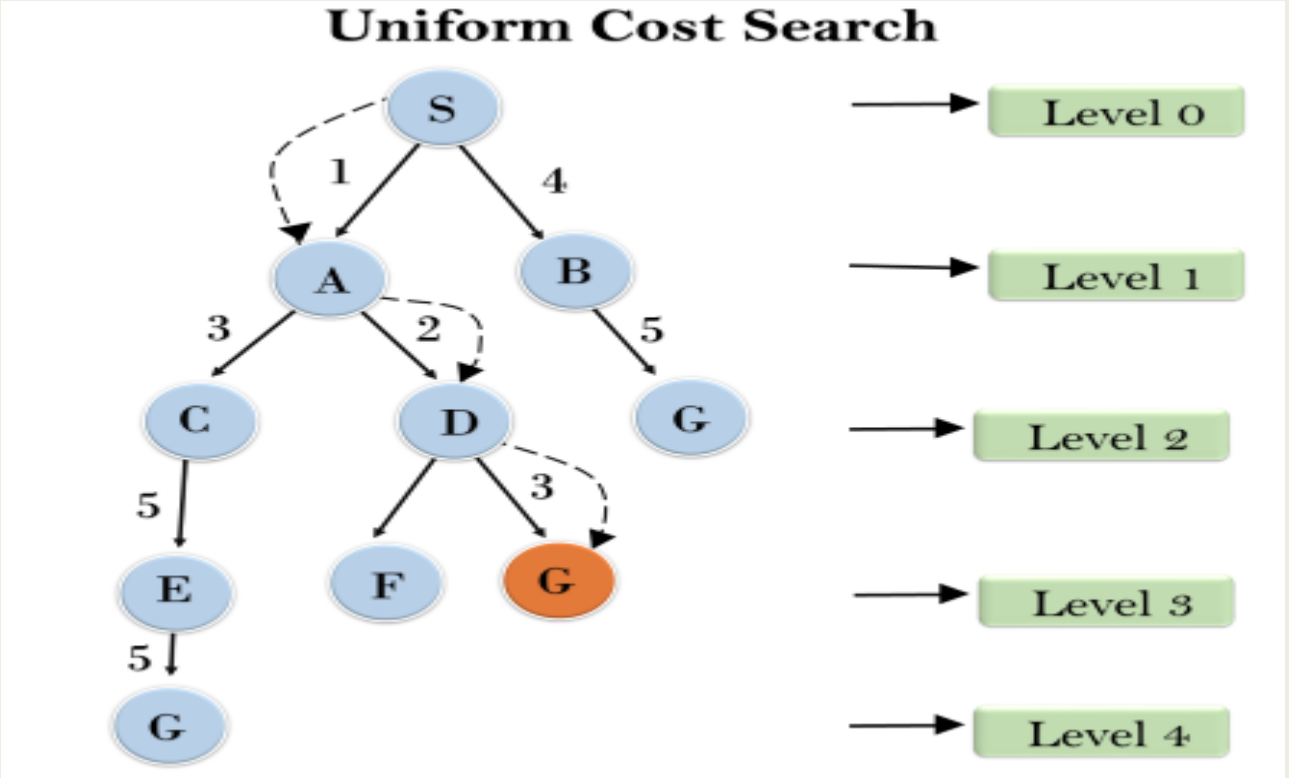
Uninform Cost Search UCS

- Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph
- Uniform Cost Search expands the least cost node first, ensuring that the path to the goal node has the minimum cost
- It uses a priority queue to store nodes. The node with the lowest cumulative cost is expanded first
- The algorithm terminates when the goal node is expanded, ensuring that the first time the goal node is reached, the path is the optimal one

Uninform Cost Search UCS

- **Initialization:** UCS starts with the root node. It is added to the priority queue with a cumulative cost of zero since no steps have been taken yet.
- **Node Expansion:** The node with the lowest path cost is removed from the priority queue. This node is then expanded, and its neighbors are explored.
- **Exploring Neighbors:** For each neighbor of the expanded node, the algorithm calculates the total cost from the start node to the neighbor through the current node. If a neighbor node is not in the priority queue, it is added to the queue with the calculated cost. If the neighbor is already in the queue but a lower cost path to this neighbor is found, the cost is updated in the queue.
- **Goal Check:** After expanding a node, the algorithm checks if it has reached the goal node. If the goal is reached, the algorithm returns the total cost to reach this node and the path taken.
- **Repetition:** This process repeats until the priority queue is empty or the goal is reached.

Uninform Cost Search UCS



Source: https://www.javatpoint.com/ai-uninformed-search-algorithms

Uninform Cost Search UCS

- It is complete and optimal
- Time and space complexity are exponential

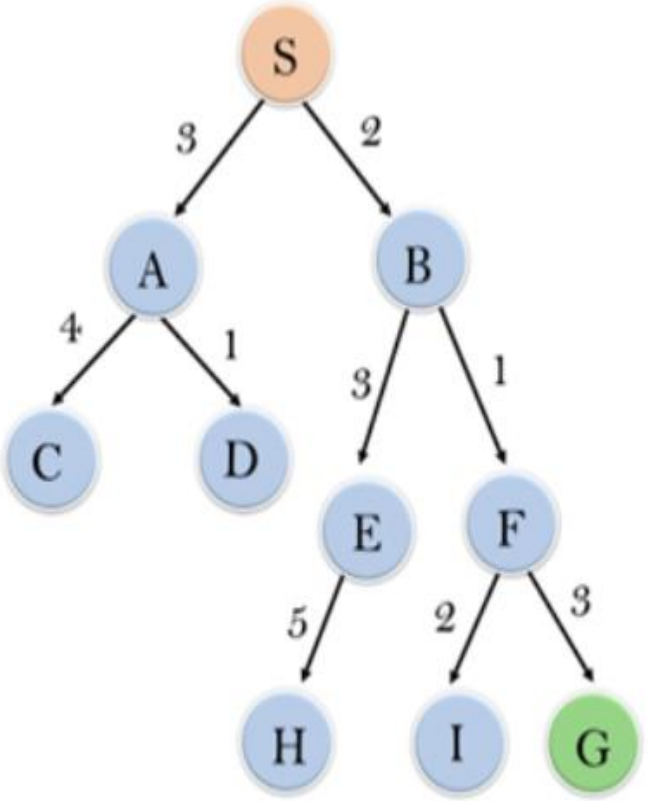
Best First Search

- Best-First search is a searching algorithm used to find the shortest path which uses distance as a heuristic.
- The distance between the starting node and the goal node is taken as heuristics. It defines the evaluation function for each node **n** in the graph as:
  - ***f(n) = h(n)*** *where* ***h(n)*** *is heuristics function*
- It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal
- The idea of **Best First Search** is to use an evaluation function to decide which adjacent is most promising and then explore
- We use a priority queue to store the costs of nodes that have the lowest evaluation function value.

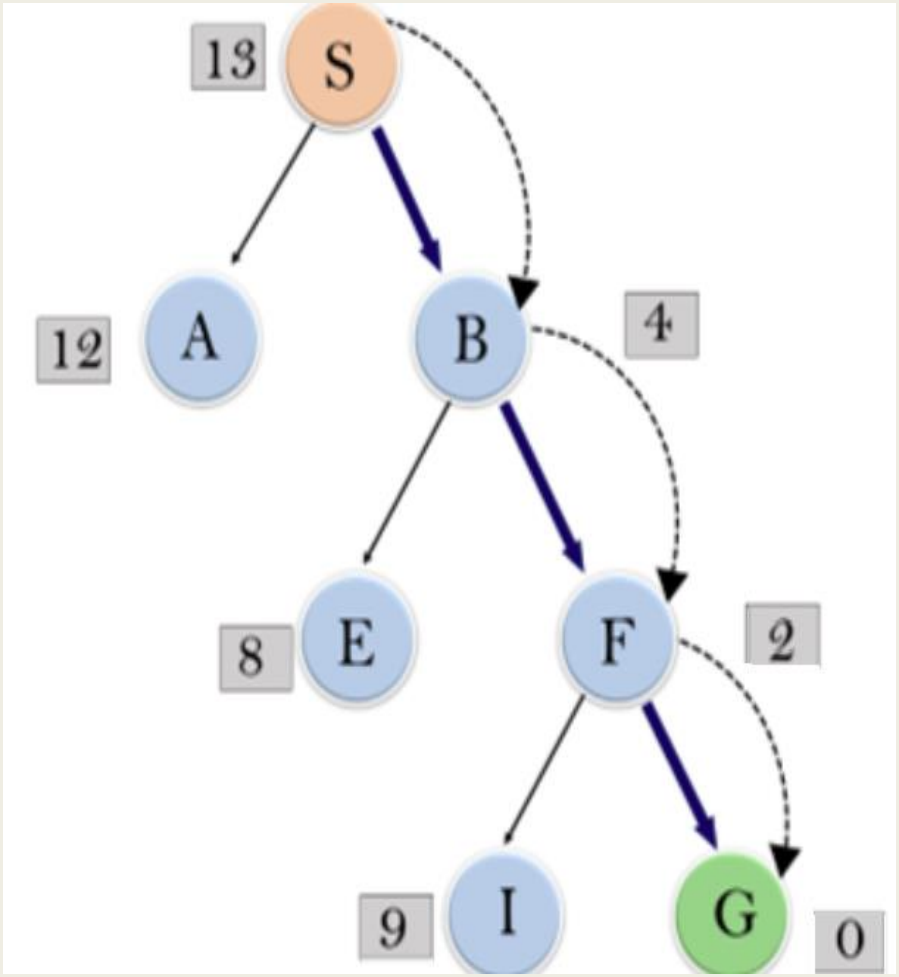
Best First Search

- Create an empty PriorityQueue pq
- Insert "start node" in pq.
- Until pq is empty
- select u node with minimum distance (heuristic)
- If u is the goal
- Exit
- Else
- For each neighbor v of u
- If v "Unvisited"
- insert v in pq
- End

Best First Search



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0



Source: https://iq.opengenus.org/best-first-search/

Best First Search

- It is not complete
- It is not optimal
- This algorithm will traverse the shortest path first in the queue. The time complexity of the algorithm is given by O(n*logn)

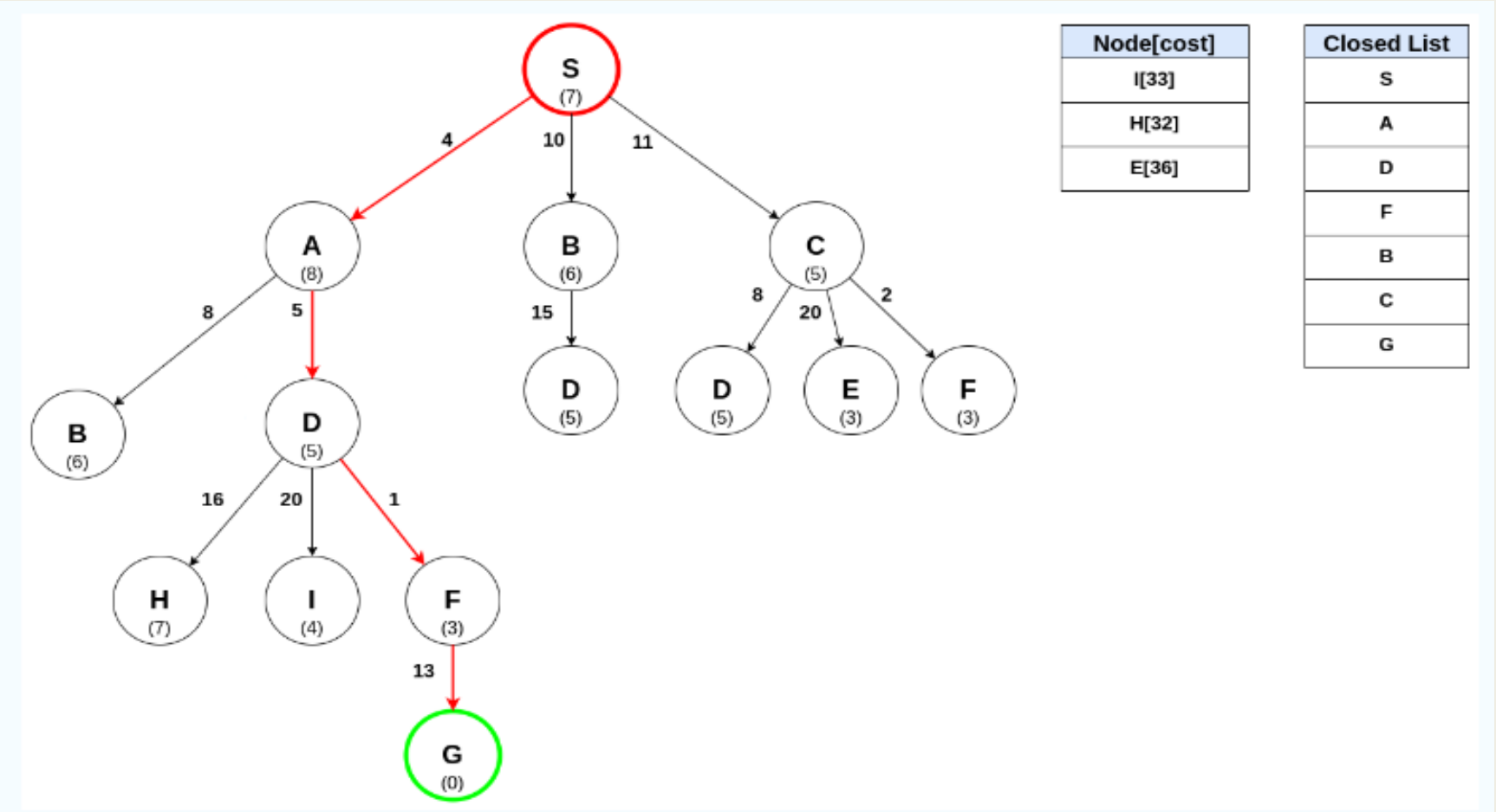
A Star Search A*

- A* is like Best-First-Search algorithm in that it can use a heuristic to guide itself
- It uses g(n) which represents the *exact cost* of the path from the starting node to any node, and h(n) which represents the heuristic *estimated cost* from node n to the goal,
- A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the node n that has the lowest f(n) = g(n) + h(n)
- To find the lowest-cost path from a start node **S** to a goal node **G**, two lists are used:
  - An **open list**, implemented as a priority queue, which stores the next nodes to be explored. Because this is a priority queue, the most promising candidate node (the one with the lowest value from the evaluation function) is always at the top. Initially, the only node in this list is the start node **S**.
  - A **closed list** which stores the nodes that have already been evaluated. When a node is in the closed list, it means that the lowest-cost path to that node has been found.

A Star Search A*

- Initialize a tree with the root node being the start node **S**.
- Remove the top node from the open list for exploration.
- Add the current node to the closed list.
- Add all nodes that have an incoming edge from the current node as child nodes in the tree.
- Update the lowest cost to reach the child node.
- Compute the evaluation function for every child node and add them to the open list.
- Repeat step 2 until you reach the goal or the open list is empty

A Star Search A*



Source: https://www.codecademy.com/resources/docs/ai/search-algorithms/a-star-search

A Star Search A*

- Complete
- It guarantees finding the optimal path when used with appropriate heuristics
- Time complexity is exponential
- Space complexity: it keeps all nodes in memory

References

- Andreas Zell slides, After the Textbook: Artificial Intelligence After the Textbook: Artificial Intelligence, A Modern Approach by Stuart Russel and Peter Norvig (3rd Edition),