# SOLVING PROBLEMS BY SEARCHING

## 1. PROBLEM FORMULATION & SOLVING

## A simple problem-solving agent

**Assumption:** environment is **observable, discrete, known and deterministic**

- **Goal formulation** - based on the current situation and the agent's performance measure, is the first step in problem solving.
  - consider a goal to be a set of world states—exactly those states in which the goal is satisfied. The agent's task is to find out how to act, now and in the future, so that it reaches a goal state.
- **Problem formulation** - the process of deciding what actions and states to consider, given a goal.
- **Search** - the process of looking for a sequence of actions that reaches the goal.
- **Solution** - A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out.
- This EXECUTION is called the **execution** phase.

## A simple problem-solving agent

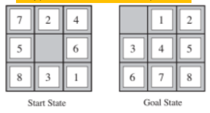
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
     goal ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, goal)
     seq ← SEARCH(problem)
     if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

- It first formulates a goal and a problem.
- searches for a sequence of actions that would solve the problem, and
- then executes the actions one at a time.
- When this is complete, it formulates another goal and starts over.

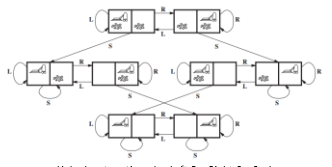
## Well-defined problems and solutions

- The **initial state** that the agent starts in.
- A description of the possible **actions** available to the agent.
  - Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions is **applicable** in s.
- A description of what each action does
  - the formal name for this is the **transition model**, specified by a function RESULT(s, a) that returns the state that results from doing action a in state s.
  - We also use the term **successor** to refer to any state reachable from a given state by a single action.
  - Together, the initial state, actions, and transition model implicitly define the **state space** of the problem—the set of all states reachable from the initial state by any sequence GRAPH of actions.
  - The state space forms a directed network or **graph** in which the nodes are states and the links between nodes are actions.
  - A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.
- A **path cost** function that assigns a numeric cost to each path.
  - The problem-solving agent chooses a cost function that reflects its own performance measure.
  - The **step cost** of taking action a in state s to reach state s is denoted by c(s, a, s)

A typical instance of the 8-puzzle.

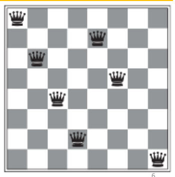


The state space for the vacuum world

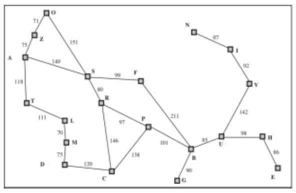


Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

A solution to the 8-queens problem



## Example: Traveling Agent



Our agent has to drive from A to B. The goal of driving to B and is considering where to go from A. Three roads lead out of A, one toward S, one to T, and one to Z.

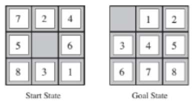
**Assumption:** environment is **observable, discrete, known and deterministic – means the agent has the road map and distance details**

Five components of the problem:

- The **initial state:**
  - **T**he initial state for our agent in a country might be described as In(A)
- A description of the possible **actions** available to the agent.
  - From the state In(A), the applicable actions are {Go(S), Go(T), Go(Z)}.
- A description of what each action does
  - We have RESULT(In(A), Go(Z)) = In(Z)
- The **goal test**
  - The agent's goal in Romania is the singleton set {In(B)}.
- A **path cost** function that assigns a numeric cost to each path.
  - For the agent trying to get to B, time is of the essence, so the cost of a path might be its length in kilometers.
  - The step costs are shown in Figure is route distances.

## Try Out - 8-puzzle

**Consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state**



The 8-puzzle has 9!/2 = 181,440 reachable states and is easily solved.

The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms.

The 24-puzzle (on a 5 × 5 board) has around $10^{25}$ states, and random instances take several hours to solve optimally.

**States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

**Initial state**: Any state can be designated as the initial state.

**Actions**: The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.

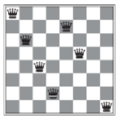
**Transition model**: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in the given figure, the resulting state has the 5 and the blank switched.

**Goal test**: This checks whether the state matches the goal configuration shown in figure (Other goal configurations are possible).

**Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

## Try Out - 8-queens problem

The **8-queens problem** is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal)


Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

**States**: Any arrangement of 0 to 8 queens on the board is a state.

**Initial state**: No queens on the board.

**Actions**: Add a queen to any empty square.

**Transition model**: Returns the board with a queen added to the specified square.

**Goal test**: 8 queens are on the board, none attacked.

**States**: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.

**Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

## Real-world problems

- **Touring problems**
- **Traveling salesperson problem**
- **VLSI layout**
- **Robot navigation**
- **Automatic assembly sequencing**

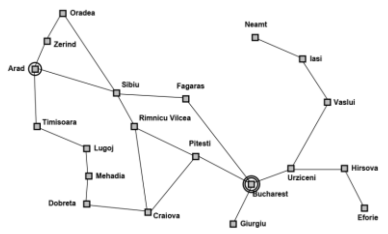
## 2. SEARCHING FOR SOLUTIONS

## General Search Algorithm

Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. *expanding* states)

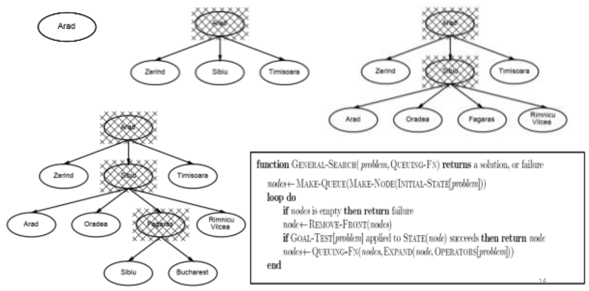
```
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
   end
```

## General Search Algorithm - Example



A simplified road map of part of Romania

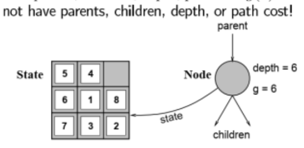
## General Search Algorithm - Example



```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
      if nodes is empty then return failure
      node ← REMOVE-FRONT(nodes)
      if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
      nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

## Implementation contd: states vs. nodes

A *state* is a (representation of) a physical configuration
A *node* is a data structure constituting part of a search tree
includes *parent, children, depth, path cost* $g(x)$
*States* do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the OPERATORS (or SUCCESSORFN) of the problem to create the corresponding states.

## Measuring problem-solving performance

- We can evaluate an algorithm's performance in four ways:
  - **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
  - **Optimality**: Does the strategy find the optimal solution?
  - **Time complexity**: How long does it take to find a solution?
  - **Space complexity**: How much memory is needed to perform the search?

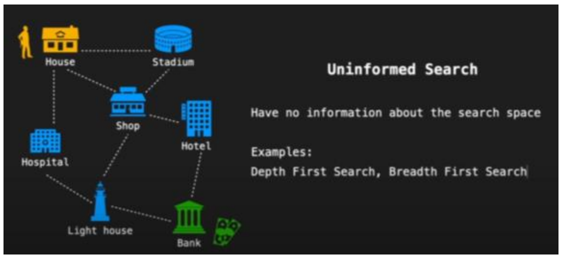
## Complexity &effectiveness of a search algorithm

- Complexity is expressed in terms of three quantities:
  - b - **branching factor** or maximum number of successors of any node
  - d - **depth** of the shallowest goal node (i.e., the number of steps along the path from the root) and
  - m - **maximum length** of any path in the state space
- Time is often measured in terms of the number of nodes generated during the search, and
- Space in terms of the maximum number of nodes stored in memory
- Effectiveness of a search algorithm is assessed using:
  - **search cost** — which typically depends on the time complexity but can also include a term for memory (or)
  - use the **total cost**, which combines the search cost and the path cost of the solution found

## UNINFORMED SEARCH STRATEGIES

## UNINFORMED SEARCH

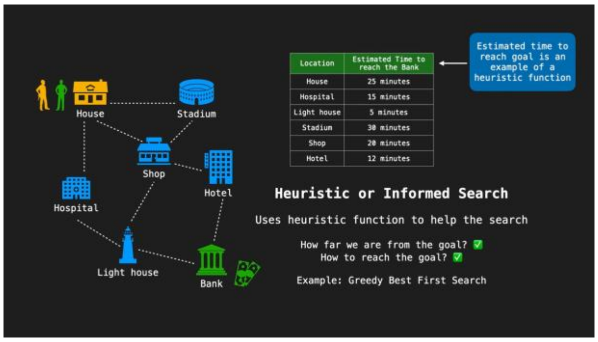
- Uninformed strategies use only the information available in the problem definition
  - a.k.a blind search
- All they can do is generate successors and distinguish a goal state from a non-goal state.
- All search strategies are distinguished by the *order* in which nodes are expanded.
- Strategies that know whether one non-goal state is "more promising" than another are called **informed search** or **heuristic**



**Uninformed Search**

Have no information about the search space

Examples:
Depth First Search, Breadth First Search



| Location | Estimated Time to reach the Bank |
|---|---|
| House | 25 minutes |
| Hospital | 15 minutes |
| Light house | 5 minutes |
| Stadium | 30 minutes |
| Shop | 20 minutes |
| Hotel | 12 minutes |

Estimated time to reach goal is an example of a heuristic function

**Heuristic or Informed Search**

Uses heuristic function to help the search

How far we are from the goal? ✅
How to reach the goal? ✅

Example: Greedy Best First Search

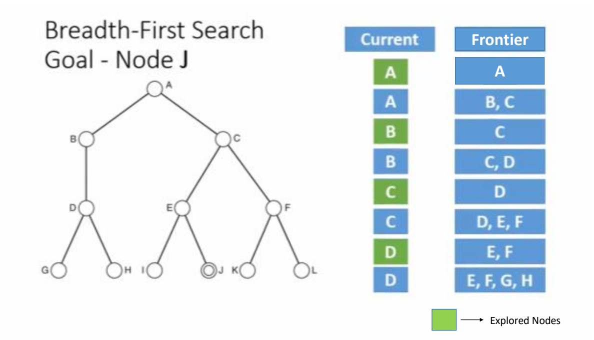
## UNINFORMED SEARCH STRATEGIES

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

## Breadth-first search

- **Breadth-first search** is a simple strategy in which the **root node is expanded first**, **then all the successors of the root node are expanded next**, then *their* successors, and so on.
- In general, <u>all the nodes are expanded at a given depth in the search tree</u> before any nodes at the next level are expanded.
- Is an instance of the general graph-search algorithm in which the *shallowest* unexpanded node is chosen for expansion.
- Achieved using a FIFO queue.
- New nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
- **General graph-search algorithm vs BFS** - the goal test is applied to each node when it is generated rather than when it is selected for expansion.

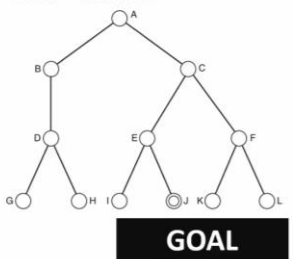
Breadth-First Search
Goal - Node J



| Current | Frontier |
|---|---|
| A | A |
| A | B, C |
| B | C |
| B | C, D |
| C | D |
| C | D, E, F |
| D | E, F |
| D | E, F, G, H |

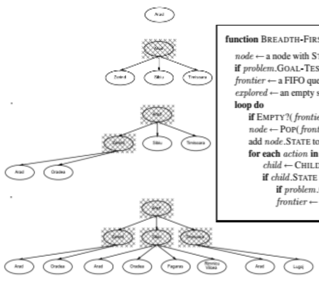
🟩 → Explored Nodes

## Breadth-First Search
## Goal - Node J



| Current | Frontier |
|---|---|
| D | E, F, G, H |
| E | F, G, H |
| E | F, G, H, I, J |
| F | G, H, I, J |
| F | G, H, I, J, K, L |
| G | H, I, J, K, L |
| H | I, J, K, L |
| I | J, K, L |
| J | K, L |

## Breadth-first search on a graph



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

## Properties of BFS

Complete?? Yes (if $b$ is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exponential in $d$

Space?? $O(b^d)$ (keeps every node in memory)

Optimal?? Yes (if cost $= 1$ per step); not optimal in general

If the shallowest goal node is at some finite depth d, BFS will eventually find it after generating all shallower nodes (provided the branching factor b is finite)

- For any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity.
- For BFS, every node generated remains in memory.
- There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier, so the space complexity is $O(b^d)$, i.e., it is dominated by the size of the frontier.

- Assume every state has b successors. Root generates b nodes at the first level, each of which generates b more nodes, and so on.
- Now suppose that the solution is at depth d.
- In the worst case, it is the last node generated at that level

BFS is optimal if the path cost is a non-decreasing function of the depth of the node. The most common such scenario is that all actions have the same cost

## Time and Space Complexity Analysis

- Assume
  - branching factor b = 10;
  - 1 million nodes/second;
  - 1000 bytes/node

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

- Many search problems fit roughly within these assumptions (give or take a factor of 100) when run on a modern personal computer.

## Lessons to Learn

- First, *the memory requirements are a bigger problem for breadth-first search than is the execution time.* One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take. Fortunately, other strategies require less memory.
- The second lesson is that time is still a major factor. If your problem has a solution at depth 16, then (given our assumptions) it will take about 350 years for breadth-first search (or indeed any uninformed search) to find it. In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

## Breadth-first search

**Advantages:**

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

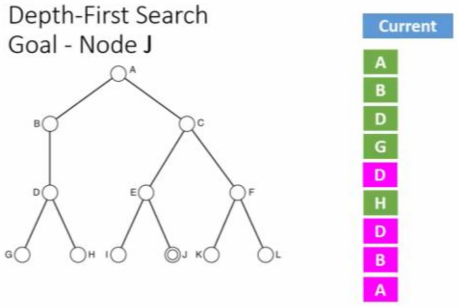
**Disadvantages:**

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

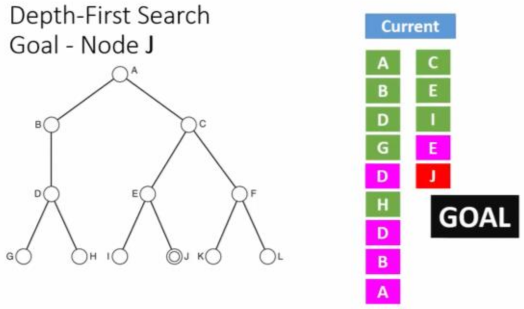
## Depth first Search

- **Depth-first search** always **expands the *deepest* node in the current frontier** of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.
- The depth-first search algorithm is an instance of the graph-search algorithm
- Breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.

## Depth-First Search
Goal - Node J



## Depth-First Search
Goal - Node J



## Properties of DFS

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if $m$ is much larger than $d$ (where m is maximum depth)
but if solutions are dense, may be much faster than breadth-first

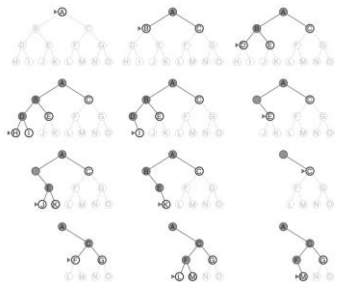
Space?? $O(bm)$, i.e., linear space!

Optimal?? No

The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite, of course).
A depth-first tree search, on the other hand, may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space.
Note that m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.

Depth-first search seems to have no clear advantage over breadth-first search. The reason is the space complexity.

**Depth-first search on a binary tree:** The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.



## Depth first Search

**Advantage:**

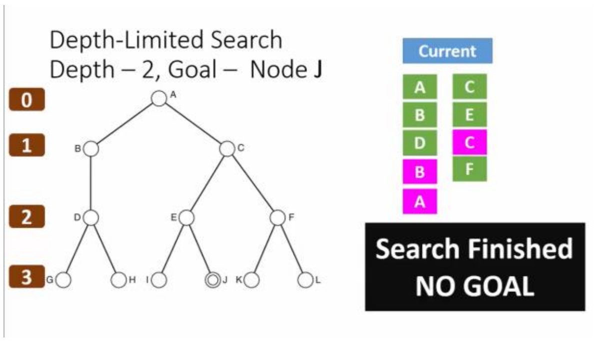
- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

**Disadvantage:**

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

## Depth-limited search

- The failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit ‘*l*’.
- That is, nodes at depth ‘*l*’ are treated as if they have no successors.
- This approach is called **depth-limited search**. The depth limit solves the infinite-path problem.
- Unfortunately, it also introduces an additional source of incompleteness if we choose l < d, that is, the shallowest goal is beyond the depth limit. (This is likely when d is unknown.)
- Depth-limited search will also be non-optimal if we choose l > d.
  - Its time complexity is $O(b^l)$ and its space complexity is O(bl).
- Depth-first search can be viewed as a special case of depth-limited search with l =∞.



## A recursive implementation of depth-limited tree search

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

**Termination condition:**
(1) the standard failure value indicates no solution;
(2) the cutoff value indicates no solution within the depth limit.

## Depth-limited search

**Advantages:**

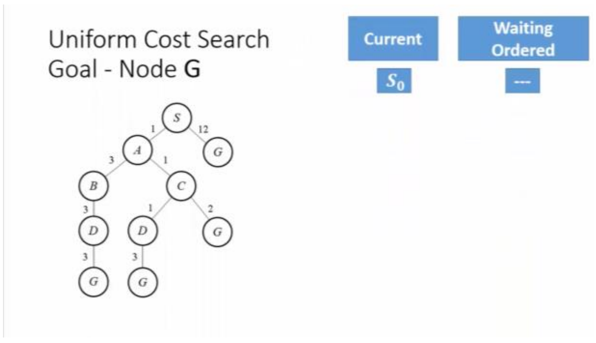
Depth-limited search is Memory efficient.

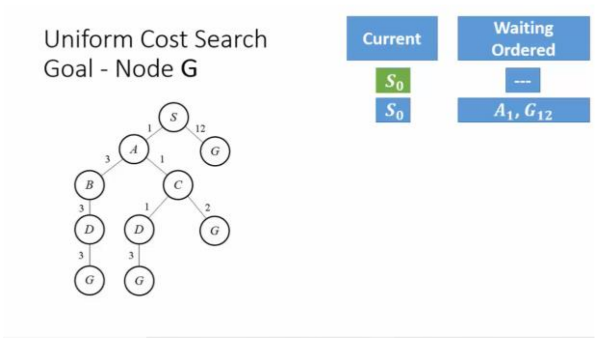
**Disadvantages:**

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

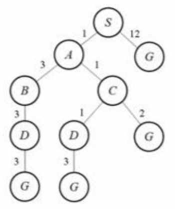
## UNIFORM-COST SEARCH

- When all step costs are equal, breadth-first search is optimal because it always expands the *shallowest* unexpanded node.
- By a simple extension, we can find an algorithm that is optimal UNIFORM-COST SEARCH with any step-cost function.
- Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost* g(n).
- This is done by storing the frontier as a priority queue ordered by g.



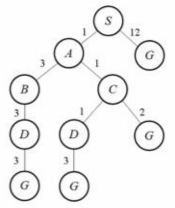


## Uniform Cost Search
Goal - Node **G**



| Current | Waiting Ordered |
|---|---|
| $S_0$ | --- |
| $S_0$ | $A_1, G_{12}$ |
| $A_1$ | $G_{12}$ |
| $A_1$ | $C_2, B_4, G_{12}$ |
| $C_2$ | $B_4, G_{12}$ |
| $C_2$ | $D_3, B_4, G_4, G_{12}$ |
| $D_3$ | $B_4, G_4, G_{12}$ |
| $D_3$ | $B_4, G_4, G_6, G_{12}$ |

## Uniform Cost Search
Goal - Node **G**



| Current | Waiting Ordered |
|---|---|
| $B_4$ | $G_4, G_6, G_{12}$ |
| $B_4$ | $G_4, G_6, D_7, G_{12}$ |
| $G_4$ | $G_6, D_7, G_{12}$ |

**GOAL**

## Uniform Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution

Optimal?? Yes

**General Search vs. UCS**

- The algorithm is identical to the general graph search algorithm, except for the use of a priority queue.
- The goal test is applied to a node when it is *selected for expansion* rather than when it is first generated.
- The second difference is that a test is added in case a better path is found to a node currently on the frontier

## Uniform Cost Search

**Advantages:**

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

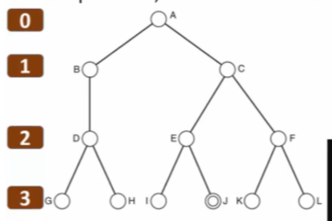
**Disadvantages:**

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

## Iterative deepening depth-first search

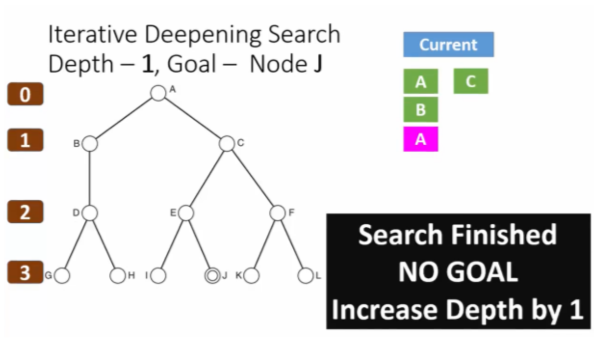
- **Iterative deepening search** (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.
- This will occur when the depth limit reaches d, the depth of the shallowest goal node.

## Iterative Deepening Search
Depth – **0**, Goal – Node **J**

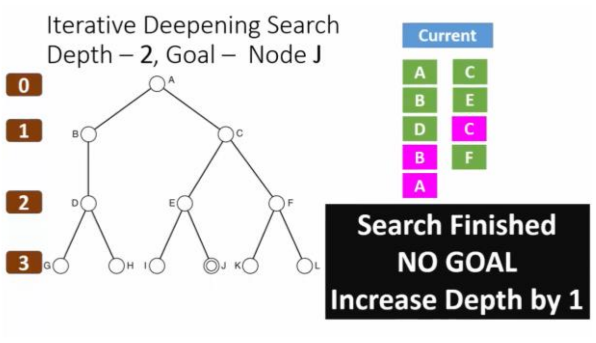


| Current |
|---|
| A |

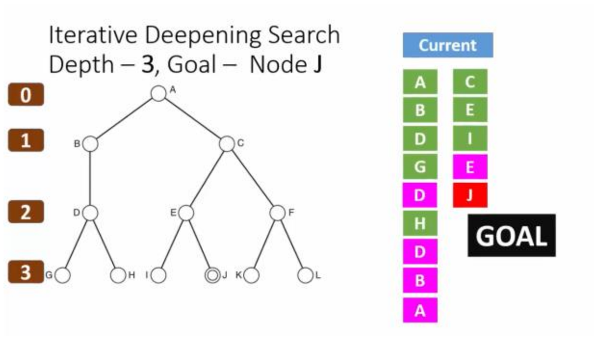
**Search Finished**
**NO GOAL**
**Increase Depth by 1**


Iterative Deepening Search
Depth – 1, Goal – Node J

Search Finished
NO GOAL
Increase Depth by 1


Iterative Deepening Search
Depth – 2, Goal – Node J

Search Finished
NO GOAL
Increase Depth by 1

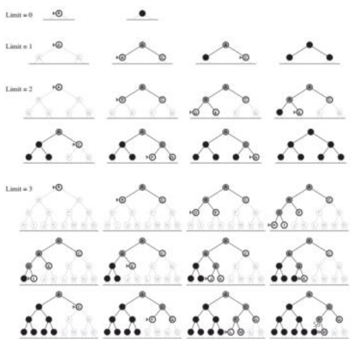

Iterative Deepening Search
Depth – 3, Goal – Node J

## Iterative deepening search algorithm

- The iterative deepening search algorithm, which repeatedly applies depth limited search with increasing limits.

```
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
```

- It terminates when a solution is found or if the depth limited search returns failure, meaning that no solution exists.



Four iterations of iterative deepening search on a binary tree

## Iterative deepening search algorithm

**Advantages:**

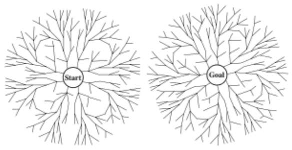
- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

**Disadvantages:**

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

## Bidirectional search

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



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

## Bidirectional search

**Advantages:**

- Bidirectional search is fast.
- Bidirectional search requires less memory

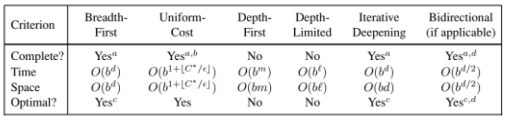
**Disadvantages:**

- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

## Bidirectional search

- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.
- The check can be done when each node is generated or selected for expansion and, with a hash table, will take constant time.
- For example, if a problem has solution depth d = 6, and each direction runs breadth-first search one node at a time, then in the worst case the two searches meet when they have generated all of the nodes at depth 3.
- The time complexity of bidirectional search using breadth-first searches in both directions is $O(b^{d/2})$.
- The space complexity is also $O(b^{d/2})$.

## Comparing uninformed search strategies



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

b - branching factor
d - depth of the shallowest solution
m - maximum depth of the search tree
l - depth limit

Superscript caveats are as follows:

- $^a$ complete if b is finite;
- $^b$ complete if step costs ≥ Ɛ for positive Ɛ
- $^c$ optimal if step costs are all identical
- $^d$ if both directions use breadth-first search

## INFORMED (HEURISTICS) SEARCH STRATEGIES

## Introduction

- Informed search strategy
  - **uses problem-specific knowledge** beyond the definition of the problem itself
  - can find solutions more efficiently than can an uninformed strategy
- The general approach we consider is called **best-first search**
- Best-first search is an instance of the general **TREE-SEARCH or GRAPH-SEARCH** algorithm in which a node is selected for expansion based on an evaluation function, f(n)
- The evaluation function is construed as a cost estimate, so the **node with the lowest evaluation is expanded first**
- The implementation of best-first graph search is identical to that for uniform-cost search, except for the use of f instead of g to order the priority queue
- The choice of ‘f’ determines the search strategy
- Most best-first algorithms include a heuristic function h(n) as a component of ‘f’:
  - **h(n)** = estimated cost of the cheapest path from the state at node n to a goal state.
- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm.
- Assumption about h(n): arbitrary, nonnegative, problem-specific functions, with one constraint
  - if n is a goal node, then h(n) = 0.

## Greedy best-first search

- **Greedy best-first search** is an informed search algorithm where the evaluation function is strictly equal to the heuristic function, disregarding the edge weights in a weighted graph because only the heuristic value is considered.
- In order to search for a goal node it expands the node that is closest to the goal as determined by the heuristic function.
- It's called "Greedy" because at each step it tries to get as close to the goal as it can.

1. Completeness: This algorithm is minimal, but not complete, since it can lead to a dead end.
2. Optimality: may not be optimal since a shorter path may exist.
3. Search Time: This approach assumes that it is likely to lead to a solution quickly.
4. Search cost: minimum since the solution is found without expanding a node that is not on the solution path.

## Evaluation & Heuristic Functions

**Evaluation Function:**

The evaluation function, **f(x)**, for the greedy best-first search algorithm is the following:

**f(x) = h(x)**

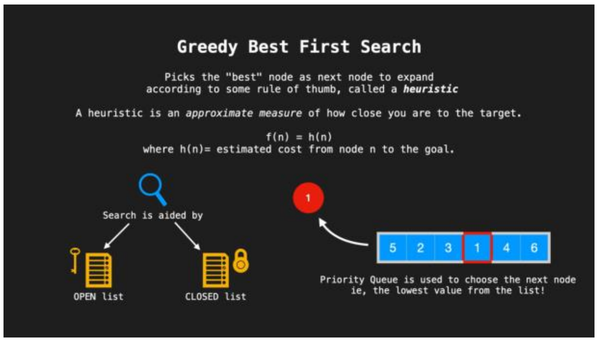
Here, the evaluation function is equal to the heuristic function. Since this search disregards edge weights, finding the lowest-cost path is not guaranteed.

**Heuristic Function:**

A heuristic function, **h(x)**, evaluates the successive node based on how close it is to the target node. In other words, it chooses the immediate low-cost option. As this is the case, however, it does not necessarily find the shortest path to the goal.

Suppose a bot is trying to move from point A to point B:

- In greedy best-first search, the bot will choose to move to the position that brings it closest to the goal, disregarding if another position ultimately yields a shorter distance.
- In the case that there is an obstruction, it will evaluate the previous nodes with the shortest distance to the goal, and continuously choose the node that is closest to the goal.



Greedy Best First Search

Picks the "best" node as next node to expand according to some rule of thumb, called a *heuristic*

A heuristic is an *approximate measure* of how close you are to the target.

f(n) = h(n)
where h(n)= estimated cost from node n to the goal.

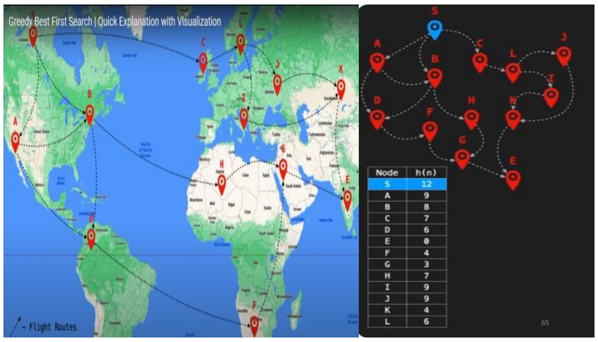
Search is aided by

OPEN list

CLOSED list

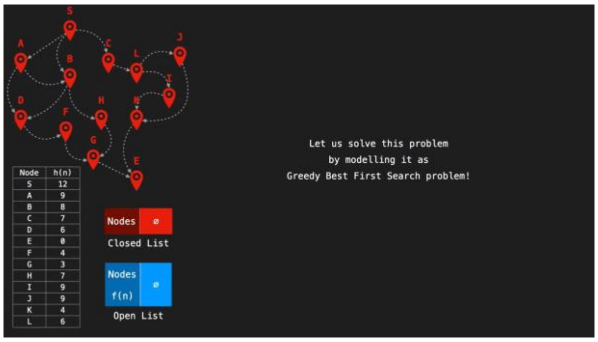
Priority Queue is used to choose the next node ie, the lowest value from the list!

Greedy Best First Search | Quick Explanation with Visualization



Greedy Best First Search | Quick Explanation with Visualization

| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

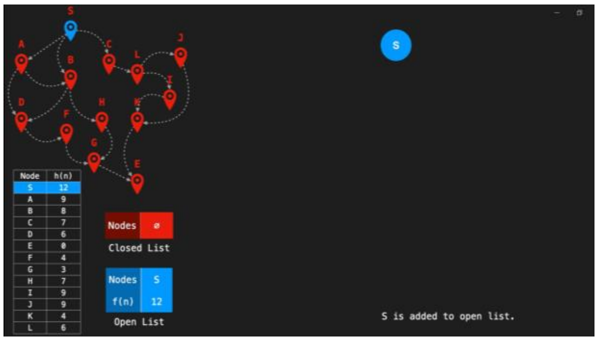
| Nodes | ∅ |
|---|---|

Closed List

| Nodes | ∅ |
|---|---|
| f(n) | |

Open List

Let us solve this problem by modelling it as Greedy Best First Search problem!



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

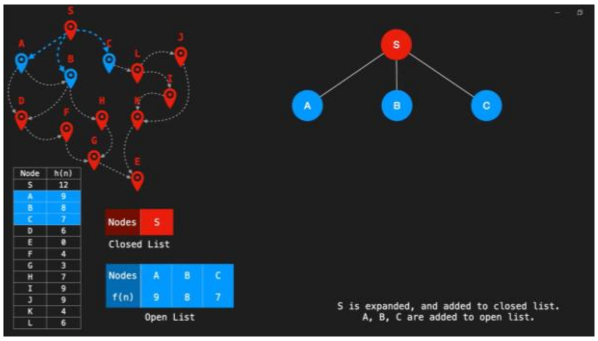
Closed List

| | |
|---|---|
| Nodes | ø |

Open List

| | |
|---|---|
| Nodes | S |
| f(n) | 12 |

S is added to open list.



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

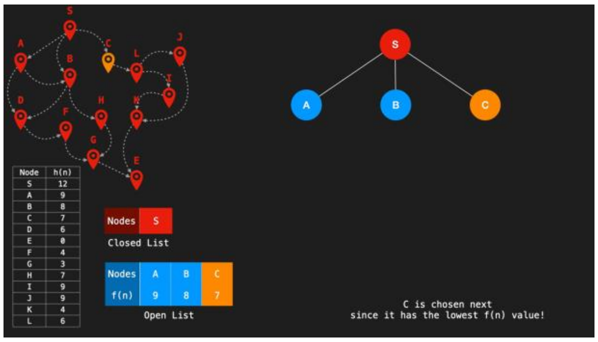
Closed List

| | |
|---|---|
| Nodes | S |

Open List

| | | | |
|---|---|---|---|
| Nodes | A | B | C |
| f(n) | 9 | 8 | 7 |

S is expanded, and added to closed list.
A, B, C are added to open list.



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

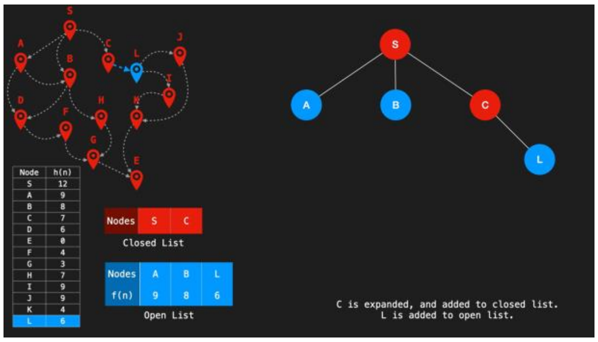
Closed List

| | |
|---|---|
| Nodes | S |

Open List

| | | | |
|---|---|---|---|
| Nodes | A | B | C |
| f(n) | 9 | 8 | 7 |

C is chosen next
since it has the lowest f(n) value!



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

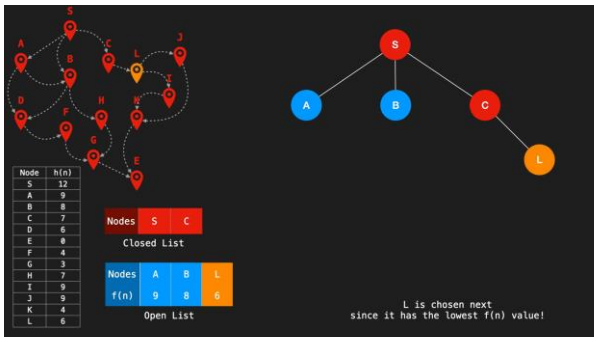
Closed List

| | | |
|---|---|---|
| Nodes | S | C |

Open List

| | | | |
|---|---|---|---|
| Nodes | A | B | L |
| f(n) | 9 | 8 | 6 |

C is expanded, and added to closed list.
L is added to open list.



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

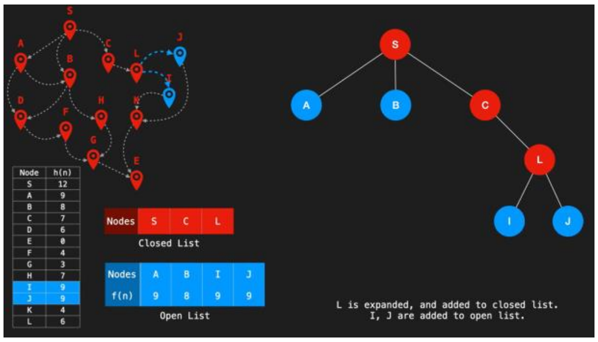
Closed List

| | | |
|---|---|---|
| Nodes | S | C |

Open List

| | | | |
|---|---|---|---|
| Nodes | A | B | L |
| f(n) | 9 | 8 | 6 |

L is chosen next
since it has the lowest f(n) value!



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

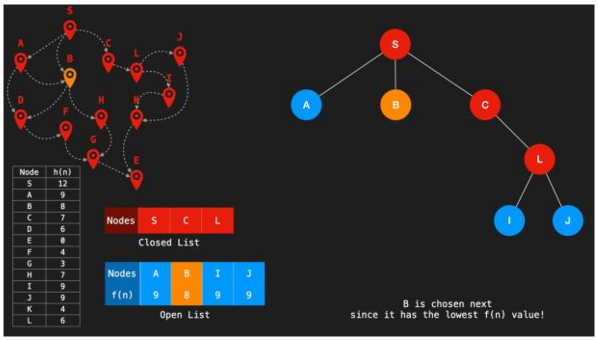
Closed List

| | | | |
|---|---|---|---|
| Nodes | S | C | L |

Open List

| | | | | |
|---|---|---|---|---|
| Nodes | A | B | I | J |
| f(n) | 9 | 8 | 9 | 9 |

L is expanded, and added to closed list.
I, J are added to open list.



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

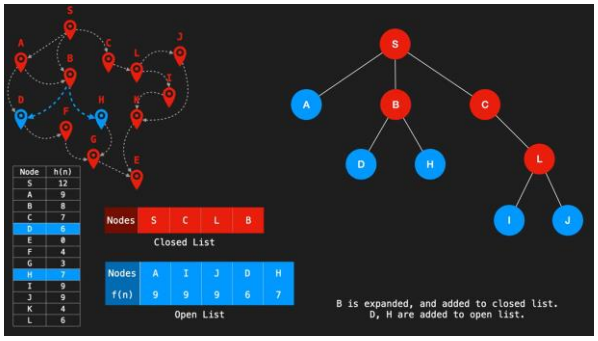
| | | | |
|---|---|---|---|
| Nodes | S | C | L |

Closed List

| | | | | |
|---|---|---|---|---|
| Nodes | A | B | I | J |
| f(n) | 9 | 8 | 9 | 9 |

Open List

B is chosen next
since it has the lowest f(n) value!



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

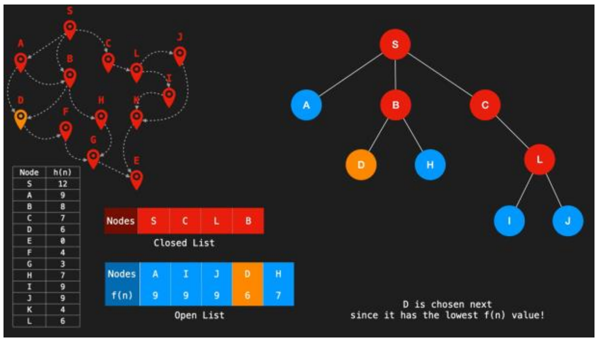
| | | | | |
|---|---|---|---|---|
| Nodes | S | C | L | B |

Closed List

| | | | | | |
|---|---|---|---|---|---|
| Nodes | A | I | J | D | H |
| f(n) | 9 | 9 | 9 | 6 | 7 |

Open List

B is expanded, and added to closed list.
D, H are added to open list.



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

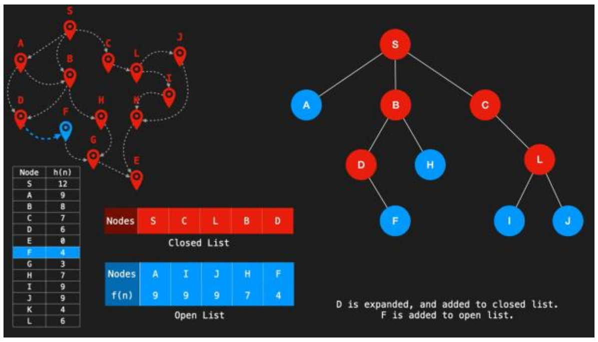
| | | | | |
|---|---|---|---|---|
| Nodes | S | C | L | B |

Closed List

| | | | | | |
|---|---|---|---|---|---|
| Nodes | A | I | J | D | H |
| f(n) | 9 | 9 | 9 | 6 | 7 |

Open List

D is chosen next
since it has the lowest f(n) value!



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

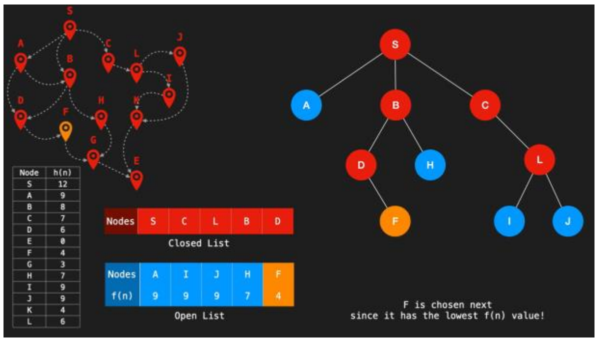
| | | | | | |
|---|---|---|---|---|---|
| Nodes | S | C | L | B | D |

Closed List

| | | | | | |
|---|---|---|---|---|---|
| Nodes | A | I | J | H | F |
| f(n) | 9 | 9 | 9 | 7 | 4 |

Open List

D is expanded, and added to closed list.
F is added to open list.



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

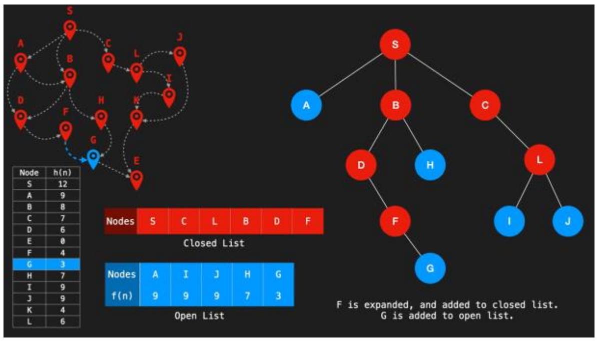
| | | | | | |
|---|---|---|---|---|---|
| Nodes | S | C | L | B | D |

Closed List

| | | | | | |
|---|---|---|---|---|---|
| Nodes | A | I | J | H | F |
| f(n) | 9 | 9 | 9 | 7 | 4 |

Open List

F is chosen next
since it has the lowest f(n) value!



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

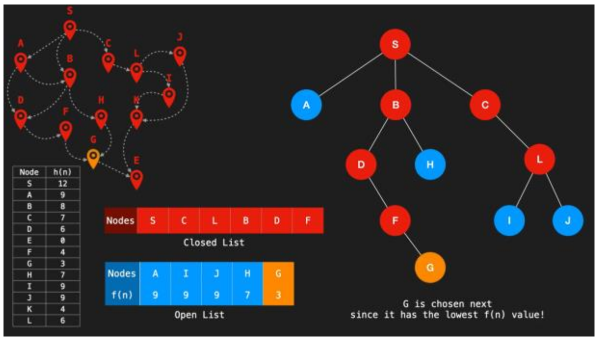
| | | | | | | |
|---|---|---|---|---|---|---|
| Nodes | S | C | L | B | D | F |

Closed List

| | | | | | |
|---|---|---|---|---|---|
| Nodes | A | I | J | H | G |
| f(n) | 9 | 9 | 9 | 7 | 3 |

Open List

F is expanded, and added to closed list.
G is added to open list.



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

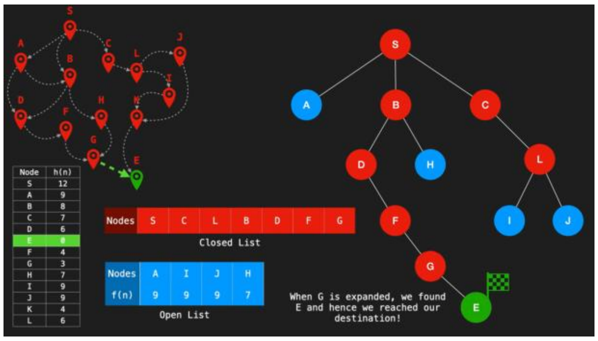
| Nodes | S | C | L | B | D | F |
|---|---|---|---|---|---|---|

Closed List

| Nodes | A | I | J | H | G |
|---|---|---|---|---|---|
| f(n) | 9 | 9 | 9 | 7 | 3 |

Open List

G is chosen next since it has the lowest f(n) value!



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

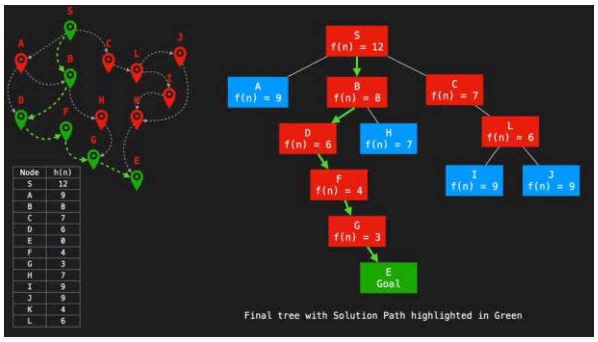
| Nodes | S | C | L | B | D | F | G |
|---|---|---|---|---|---|---|---|

Closed List

| Nodes | A | I | J | H |
|---|---|---|---|---|
| f(n) | 9 | 9 | 9 | 7 |

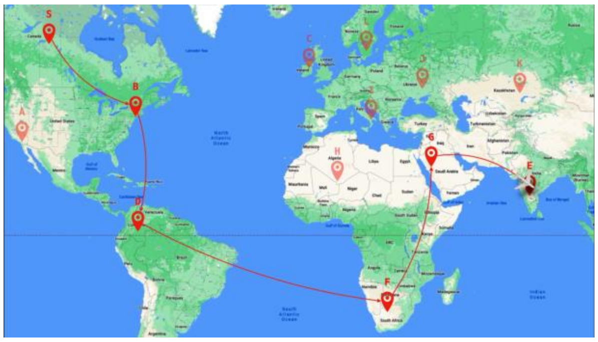
Open List

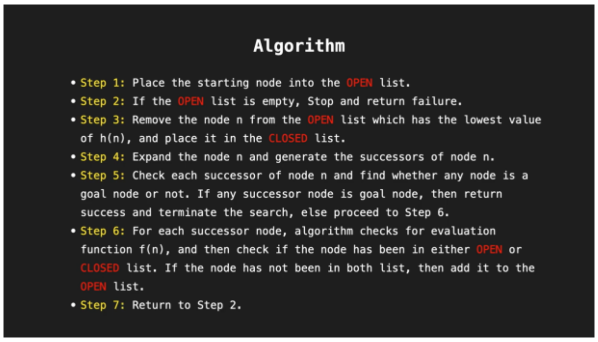
When G is expanded, we found E and hence we reached our destination!



| Node | h(n) |
|---|---|
| S | 12 |
| A | 9 |
| B | 8 |
| C | 7 |
| D | 6 |
| E | 0 |
| F | 4 |
| G | 3 |
| H | 7 |
| I | 9 |
| J | 9 |
| K | 4 |
| L | 6 |

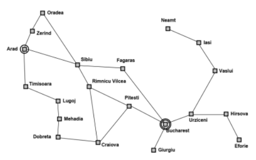
Final tree with Solution Path highlighted in Green





## Algorithm

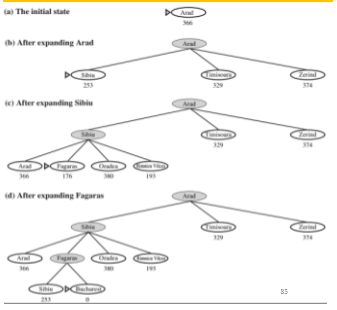
- Step 1: Place the starting node into the OPEN list.
- Step 2: If the OPEN list is empty, Stop and return failure.
- Step 3: Remove the node n from the OPEN list which has the lowest value of h(n), and place it in the CLOSED list.
- Step 4: Expand the node n and generate the successors of node n.
- Step 5: Check each successor of node n and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- Step 6: For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- Step 7: Return to Step 2.



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

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

Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic $h_{SLD}$. Nodes are labeled with their h-values.



## Advantages of Greedy Best-First Search:

•**Simple and Easy to Implement:** Greedy Best-First Search is a relatively straightforward algorithm, making it easy to implement.

•**Fast and Efficient:** Greedy Best-First Search is a very fast algorithm, making it ideal for applications where speed is essential.

•**Low Memory Requirements:** Greedy Best-First Search requires only a small amount of memory, making it suitable for applications with limited memory.

•**Flexible:** Greedy Best-First Search can be adapted to different types of problems and can be easily extended to more complex problems.

•**Efficiency:** If the heuristic function used in Greedy Best-First Search is good to estimate, how close a node is to the solution, this algorithm can be a very efficient and find a solution quickly, even in large search spaces.

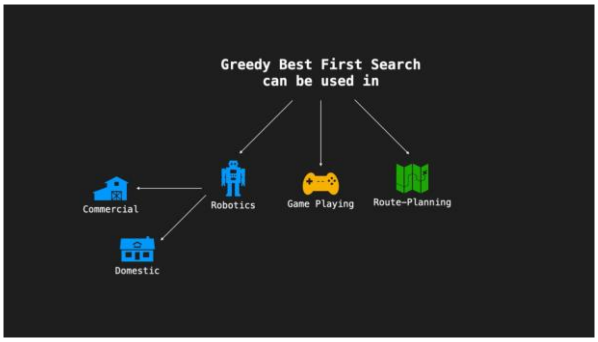
## Disadvantages of Greedy Best-First Search:

•**Inaccurate Results:** Greedy Best-First Search is not always guaranteed to find the optimal solution, as it is only concerned with finding the most promising path.

•**Local Optima:** Greedy Best-First Search can get stuck in local optima, meaning that the path chosen may not be the best possible path.

•**Heuristic Function:** Greedy Best-First Search requires a heuristic function in order to work, which adds complexity to the algorithm**.**

•**Lack of Completeness:** Greedy Best-First Search is not a complete algorithm, meaning it may not always find a solution if one is exists. This can happen if the algorithm gets stuck in a cycle or if the search space is a too much complex.



## Applications of Greedy Best-First Search:

•**Path finding:** Greedy Best-First Search is used to find the shortest path between two points in a graph. It is used in many applications such as video games, robotics, and navigation systems.

•**Machine Learning:** Greedy Best-First Search can be used in machine learning algorithms to find the most promising path through a search space.

•**Optimization:** Greedy Best-First Search can be used to optimize the parameters of a system in order to achieve the desired result.

•**Game AI:** Greedy Best-First Search can be used in game AI to evaluate potential moves and chose the best one.

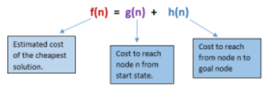
•**Navigation:** Greedy Best-First Search can be use to navigate to find the shortest path between two locations.

•**Natural Language Processing:** Greedy Best-First Search can be use in natural language processing tasks such as language translation or speech recognition to generate the most likely sequence of words.

•**Image Processing:** Greedy Best-First Search can be use in image processing to segment image into regions of interest.

## A* search

- A* search is the most commonly known form of best-first search.
- It uses h(n) - heuristic function, and g(n) - cost to reach the node n from the start state.
- It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently.
- A* search algorithm finds the shortest path through the search space using the heuristic function.
- This search algorithm expands less search tree and provides optimal result faster.
- A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).
- Hence we can combine both costs as following, and this sum is called as a **fitness number**.


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

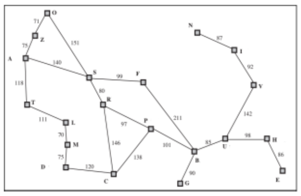
**At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.**

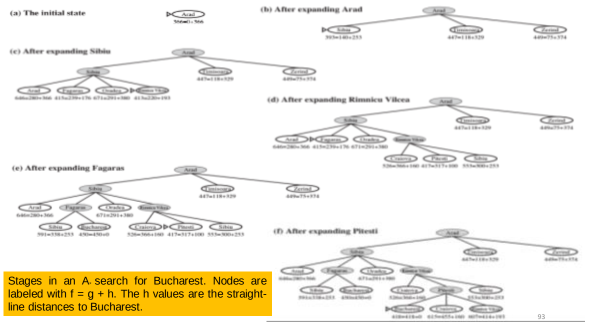
## Algorithm of A* search:

- **Step1:** Place the starting node in the OPEN list.
- **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
- **Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
- **Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- **Step 6:** Return to **Step 2**.

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

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





Stages in an A search for Bucharest. Nodes are labeled with f = g + h. The h values are the straight-line distances to Bucharest.

## Pros and Cons

- **Advantages:**
  - A* search algorithm is the best algorithm than other search algorithms.
  - A* search algorithm is optimal and complete.
  - This algorithm can solve very complex problems.
- **Disadvantages:**
  - It does not always produce the shortest path as it mostly based on heuristics and approximation.
  - The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.