Chapter 3

Solving problems by searching

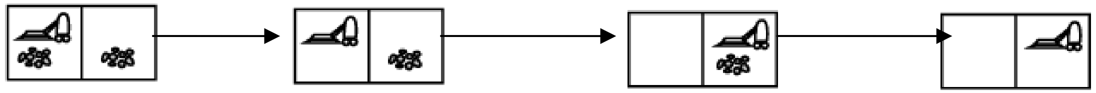
Objectives

- ❖Identify the type of agent that solve problem by searching
- ❖Problem formulation and goal formulation
- ❖Types of problem based on environment type
- ❖Discuss various techniques of search strategies

- Type of agent that solve problem by searching
  - Such agent is not reflex or model based reflex agent because this agent needs to achieve some target (goal)
  - It can be goal based or utility based or learning agent
- The basic algorithm for problem-solving agents consists of 3 phases:
  - Formulate the problem
  - Search for a solution and
  - Execute the solution.

- In problem solving by searching, solution can be described into two ways. **Solution** can be provided as state sequence or action sequence
- for example consider the **vacuum cleaner world** with initial state as shown bellow

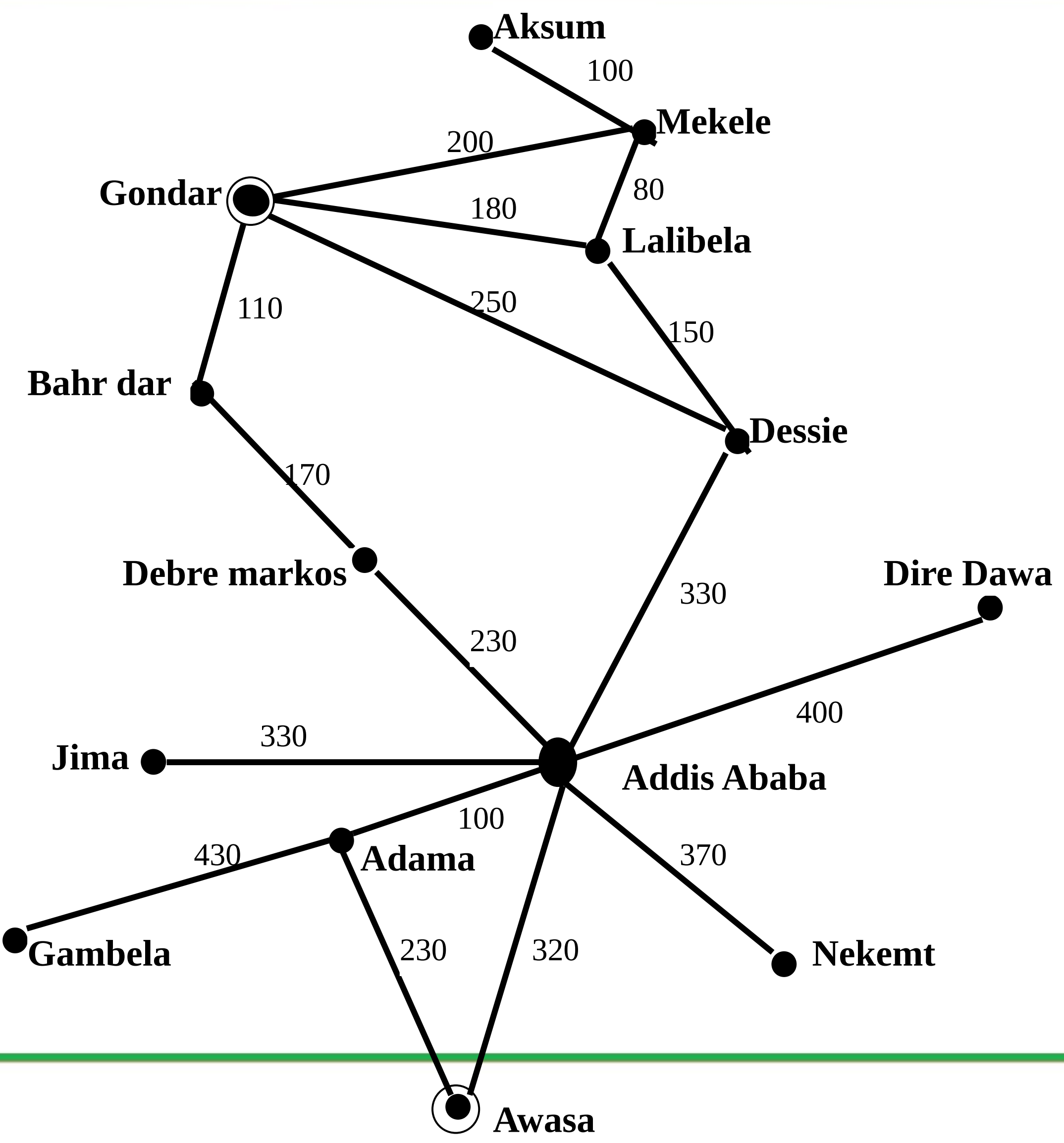
➢Solution as state sequence becomes:



➢Solution as action sequence becomes

suck → move Right → suck

Example: Road map of Ethiopia



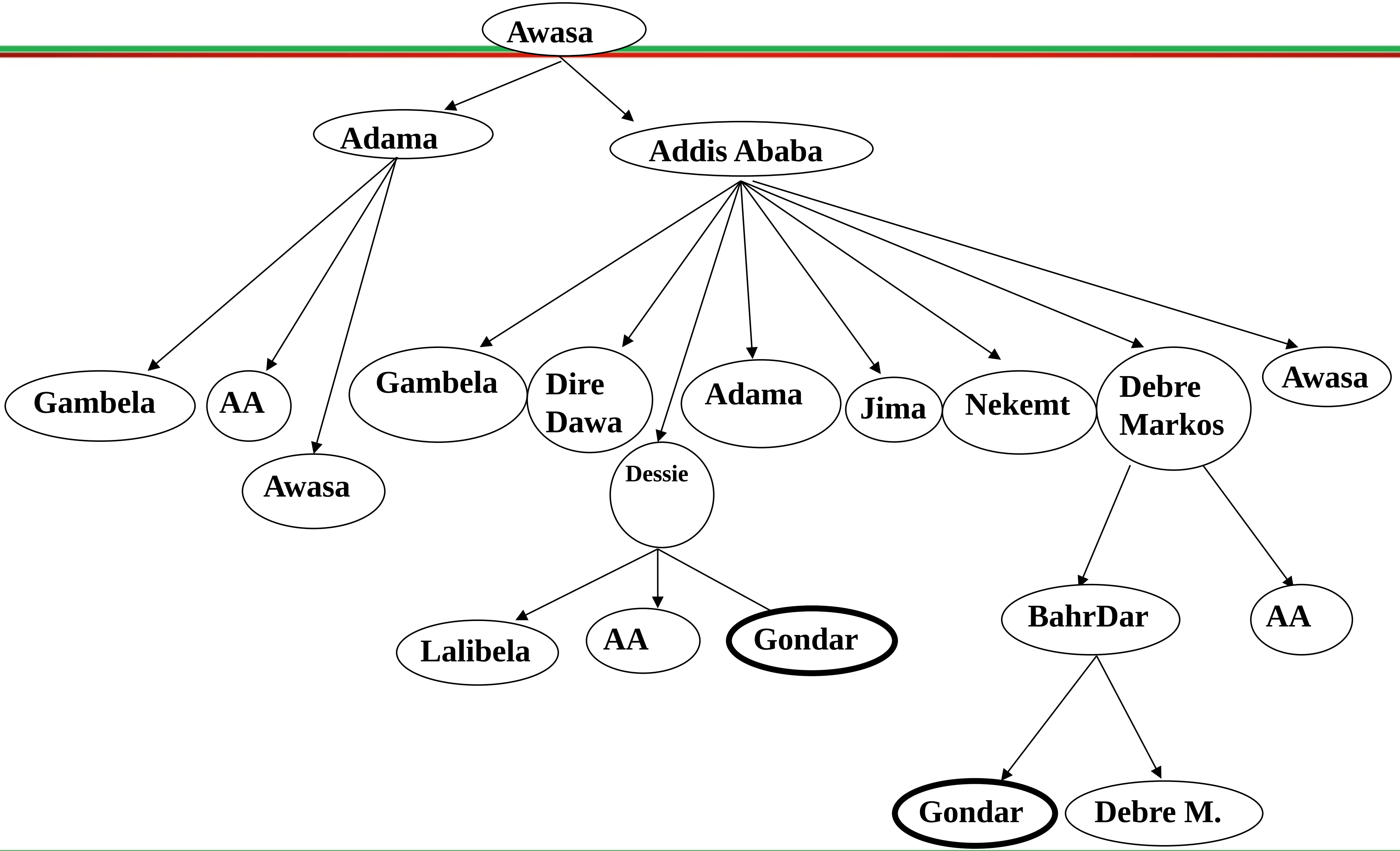
Example: Road map of Ethiopia

- **Current position of the agent: Awasa.**
- **Needs to arrive to: Gondar**
- **Formulate goal:**
  - **be in Gondar**
- **Formulate problem:**
  - **states: various cities**
  - **actions: drive between cities**
- **Find solution:**
  - **sequence of cities, e.g., Awasa, Adama, Addis Ababa, Dessie, Godar**

Generating action sequences- search trees

- **A search process** can be viewed as building a search tree over the **state space**.
- **Search tree**: is a tree structure defined by initial state and a successor function.
- **Search Node**: is the root of the search tree representing initial state and without a parent.
- **A child node:** is a node adjacent to the parent node obtained by applying an operator or rule.

Tree search example



Implementation: general tree search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are **evaluated** along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- 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 ∞)
- Generally, **searching strategies** can be classified in to two as uninformed and informed search strategies

Uninformed search (blind search) strategies

- **Uninformed** *search strategies* **use only the information available** in the **problem definition.**
- They **have no information** about the **number of steps** or **the path cost** from the **current state to the goal.**
- They can **distinguish** the **goal state** from **other states**
- They are **still important** because **there are problems** with **no additional information**.
- ***Six kinds*** of such **search strategies** will be discussed and each depends on the order of expansion of successor nodes.
    1. Breadth-first search
    2. Uniform-cost search
    3. Depth-first search
    4. Depth-limited search
    5. Iterative deepening search
    6. Bidirectional search

Generating action sequences- search trees

- **Leaf Node:** is a **node without successors** (or children).
- **Depth (d):** of a node is the **number of actions** required **to reach it** from the **initial state**.
- **Frontier** or **Fringe Nodes**: are the **collection of nodes** that are **waiting to be expanded**.
- **Path cost:** of a node is the **total cost leading** to **this node**.
- **Branch Factor**(*b*): Max. number of successors for any node.

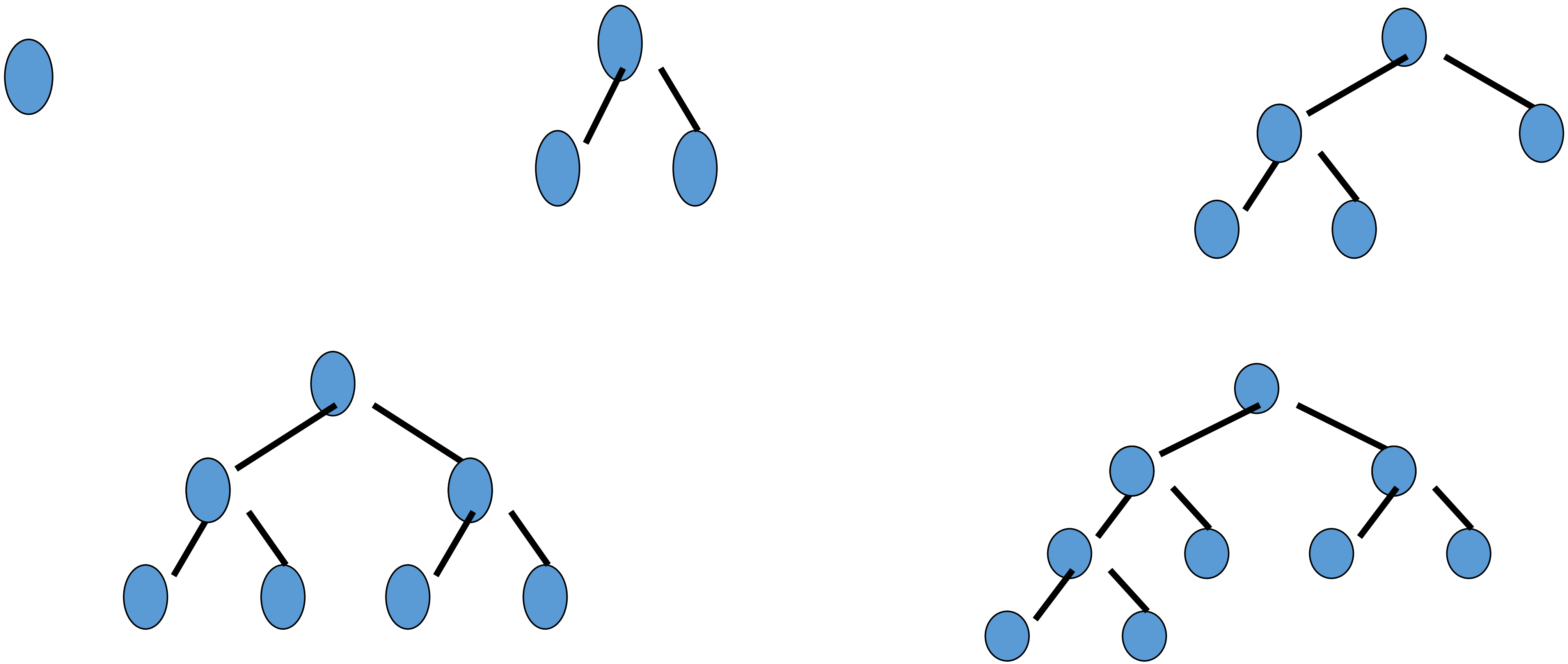
Breadth-first search

- Is one simple search strategy
  - Uses no prior information, nor knowledge
  - It tracks all nodes because it does not know whether this node leads to a goal or not!
  - Keep on trying until you get solution
  - We are searching but with cost

Breadth-first search

- **In this strategy**
  - All nodes are expanded from the root node
  - That is, the root node is expanded first
  - Then, all the nodes generated by the root node are expanded next
  - Then, their successors, and so on
  - That is, BFS expands all nodes at level *d* before expanding nodes at level *d+1*
  - It checks all paths of a given length before moving to any longer path
  - Expands the shallowest node first

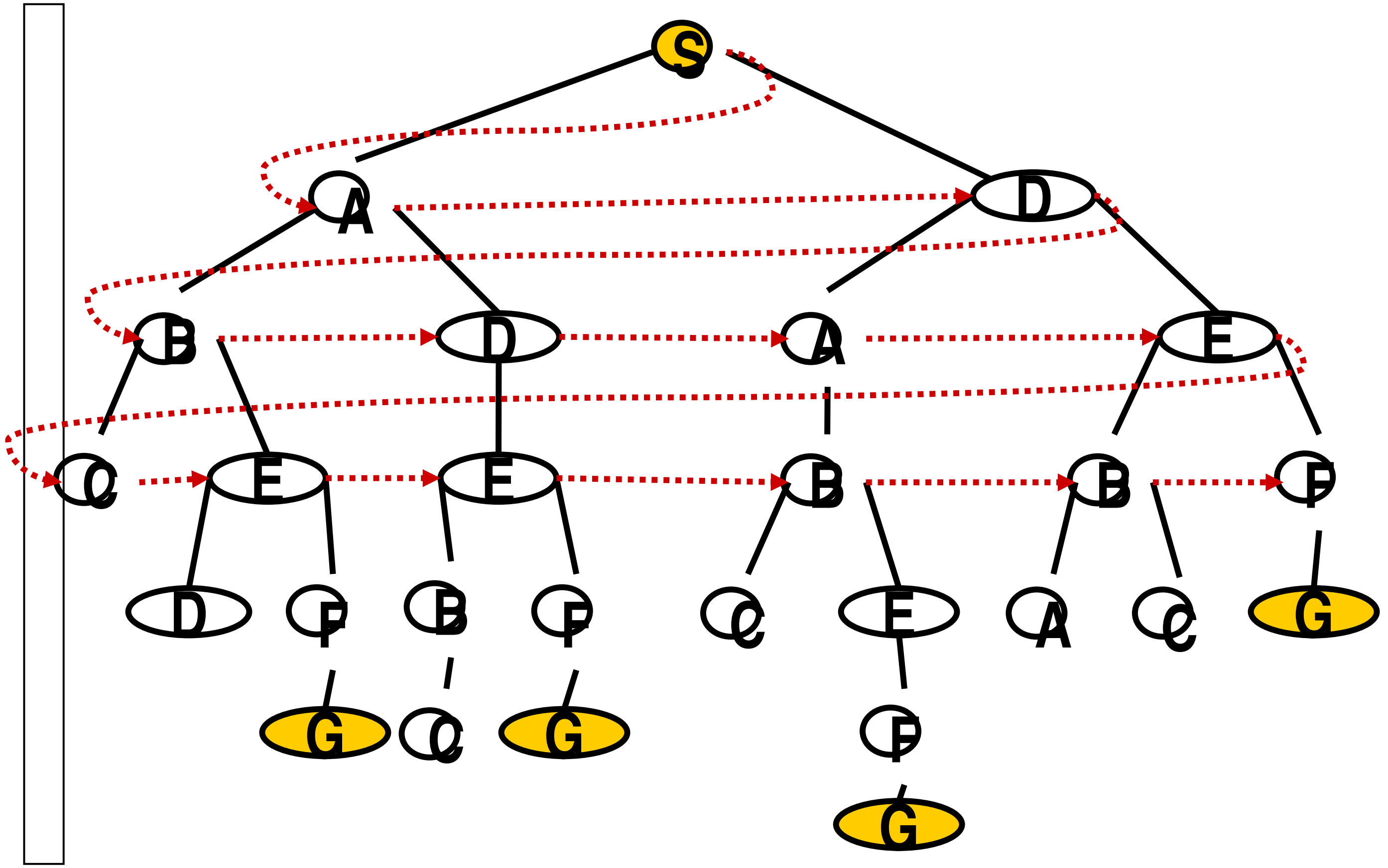
Breadth-First Search



The figure shows the progress of the search on a simply binary tree

BFS trees after 0, 1, 2, 3, and 4 nodes expansion

Breadth-First Search- Example



- Move downwards, level by level, until goal is reached.

Breadth-First Search algorithm

- Blind search in which the list of nodes is a queue
- To solve a problem using breadth-first search:
  1. Set L to be a list of the initial node in the problem.
  2. If L is empty, **return failure** otherwise pick the first node *n* from L
  3. If n is a goal state, quit and return the path from initial node to *n*
  4. **Otherwise** remove *n* **from L** and **add to the end of L** all of ***n*'s children**. Label each child with its path from initial node
  5. Return to 2.

Algorithm for Breadth-first search

BFS can be implemented using a queuing function that puts the newly generated states at the end of the the que, after all previously generated states

1. QUEUE <- path only containing the root;

2. WHILE QUEUE is not empty
   AND goal is not reached

   DO remove the first path from the QUEUE;
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths to **back** of QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;

Properties of breadth-first search

- Complete? Yes (if *b* is finite, which is true in most cases)
- Time? $1+b+b^2+b^3+\ldots+b^d = O(b^{d+1})$
  - at depth value = i , there are b^i nodes expanded for $i \leq d$
- Space? $O(b^d)$ (keeps every node in memory)
  - a maximum of this much node will be while reaching to the goal node
  - This is a major problem for real problem
- Optimal? Yes (if cost = constant (k) per step)
- Space is the bigger problem (more than time)

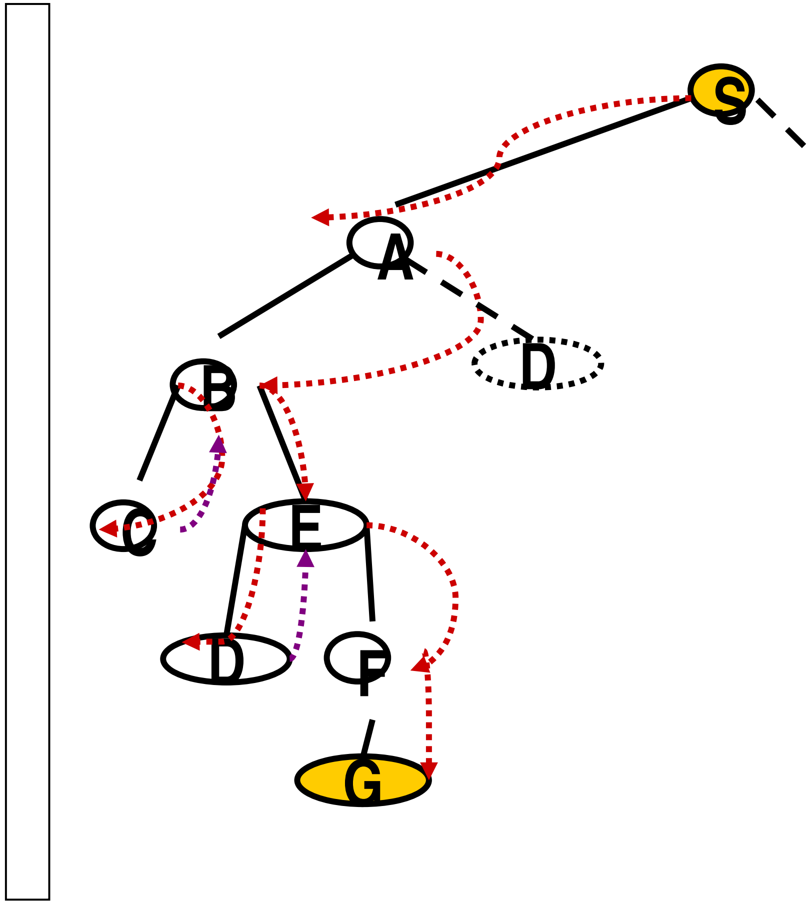
Using the same hypothetical state space find the time and memory required for a BFS with branching factor b=10 and various values of the solution depth d

Depth	Nodes	Time		Memory	
0	1	1	millisecond	100	bytes
2	111	0.1	second	11	kilobytes
4	11,111	11	seconds	1	megabyte
6	10^6	18	minutes	111	megabytes
8	10^8	31	hours	11	gigabytes
10	10^{10}	128	days	1	terabyte
12	10^{12}	35	years	111	terabytes
14	10^{14}	3500	years	11,111	terabytes

Depth-first Search (DFS)

- Pick one of the children at every node visited, and work forward from that child.
- **Always expands** the deepest node reached so far (and therefore searches one path to a leaf before allowing up any other path).
- Thus, it finds the **left most solution**

Depth-first search- Chronological backtracking



- **Select a child**
- **convention**: left-to-right
- **Repeatedly** go to **next child**, as long as possib
- Return to **left-over alternatives** (higher-u when needed.

Depth-first search algorithm

1. *QUEUE* <-- *path only containing the root;*

2. *WHILE* *QUEUE is not empty*
   AND goal is not reached

   DO *remove the first path from the QUEUE;*
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths to ***front*** *of QUEUE;*

3. *IF goal reached*
   THEN success;
   ELSE failure;

Time Requirements of Depth-First Search

- Then the worst case time complexity is **$O(b^m)$**
- However, for very deep (or infinite due to cycles) trees this search may spend a lot of time (forever) searching down the wrong branch
- It is also more likely to return a solution path that is longer than the optimal
- Because it may not find a solution if one exists, this search strategy is ***not complete*.**
- **Remarks:** Avoid DFS for large or infinite maximum depths.

DFS Practical evaluation:

DFS is a **method** of choice **when there is a known** (and reasonable) **depth bound**, and finding **any solution is sufficient.**

1.Depth-first search:

IF the **search space** contains **very deep branches** without solution, THEN **Depth-first may waste much time in them**.

2. Breadth-first search:

Is **VERY demanding on memory** !

Solutions ??

Iterative deepening

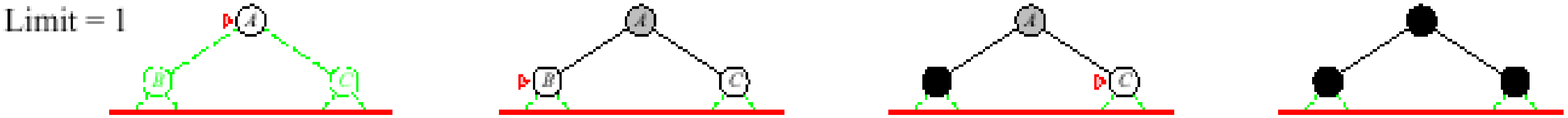
Iterative Deepening Search Algorithm

The order of expansion of states is similar to BFS,except that some states are expanded multiple times

Iterative Deepening Search l = 0

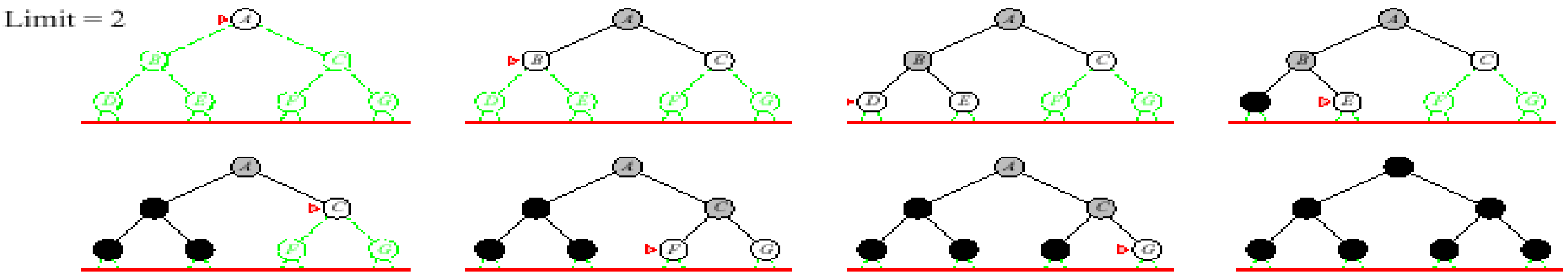
Limit = 0

Iterative Deepening Search l = 1



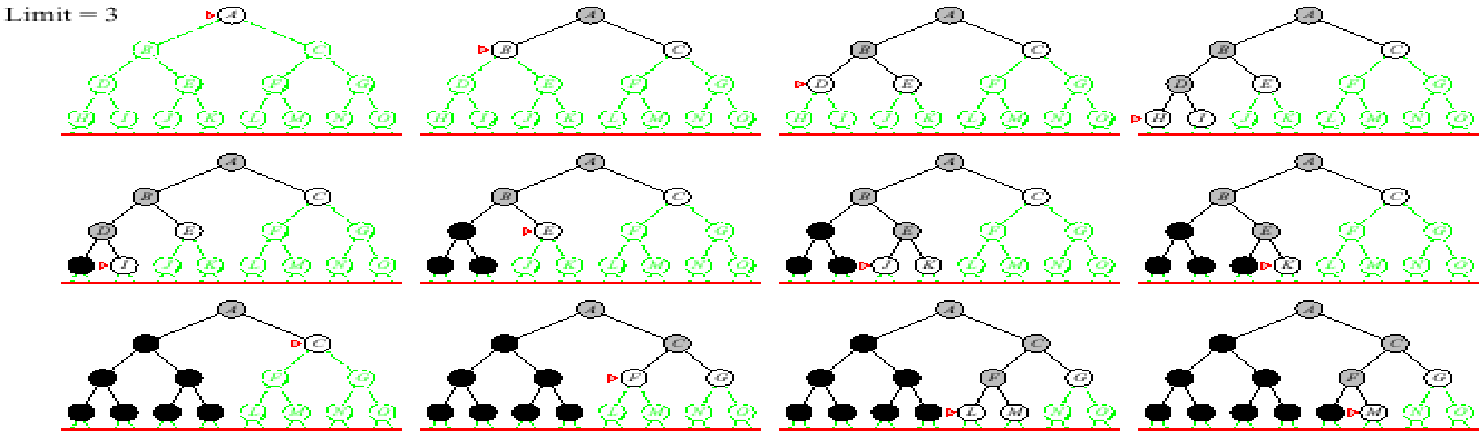
Iterative Deepening Search l = 2

Limit = 2

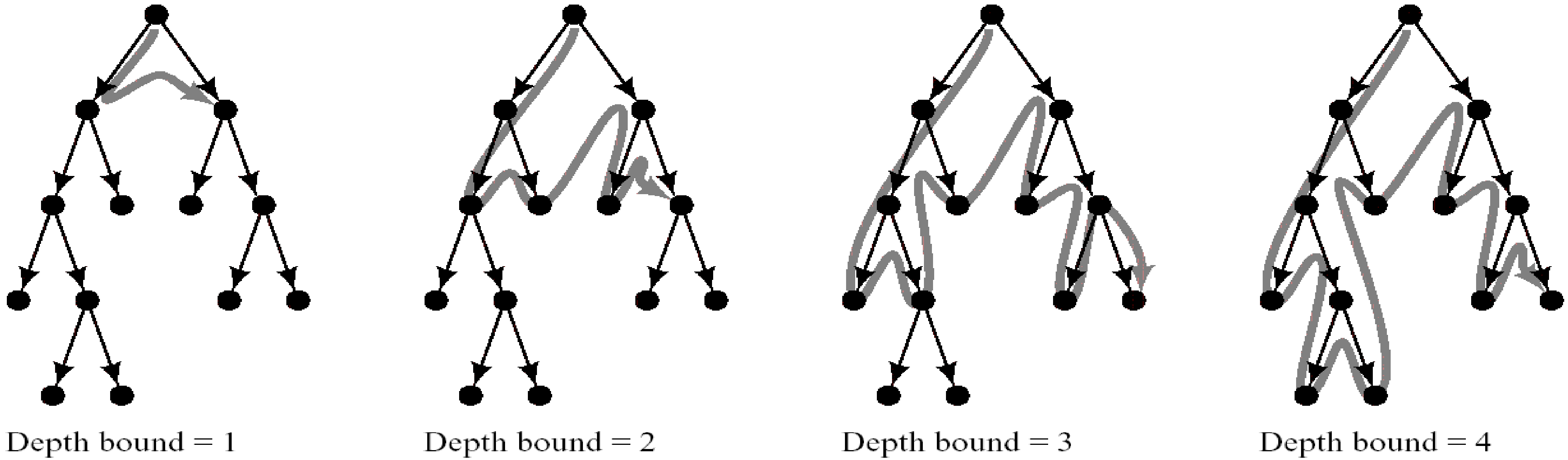


Iterative Deepening Search l = 3

Limit = 3



Iterative Deepening Search l = 1 to l=4



Stages in Iterative-Deepening Search

- It **requires little memory** (a constant times depth of the current node)
- **Is complete**
- Finds **a minim-depth solution** as does **BFS**

Iterative Deepening Search Algorithm

1. DEPTH <-- 1

2. WHILE goal is not reached

   DO { perform Depth-limited search;
   increase DEPTH by 1;

- **It is a strategy** that avoids (sidesteps) the issue of choosing the best depth limit by **trying all possible depth limits**
- Finds the **best depth limit** by gradually increase the limit -> 0, 1, 2, …until **goal** is found at depth limit d

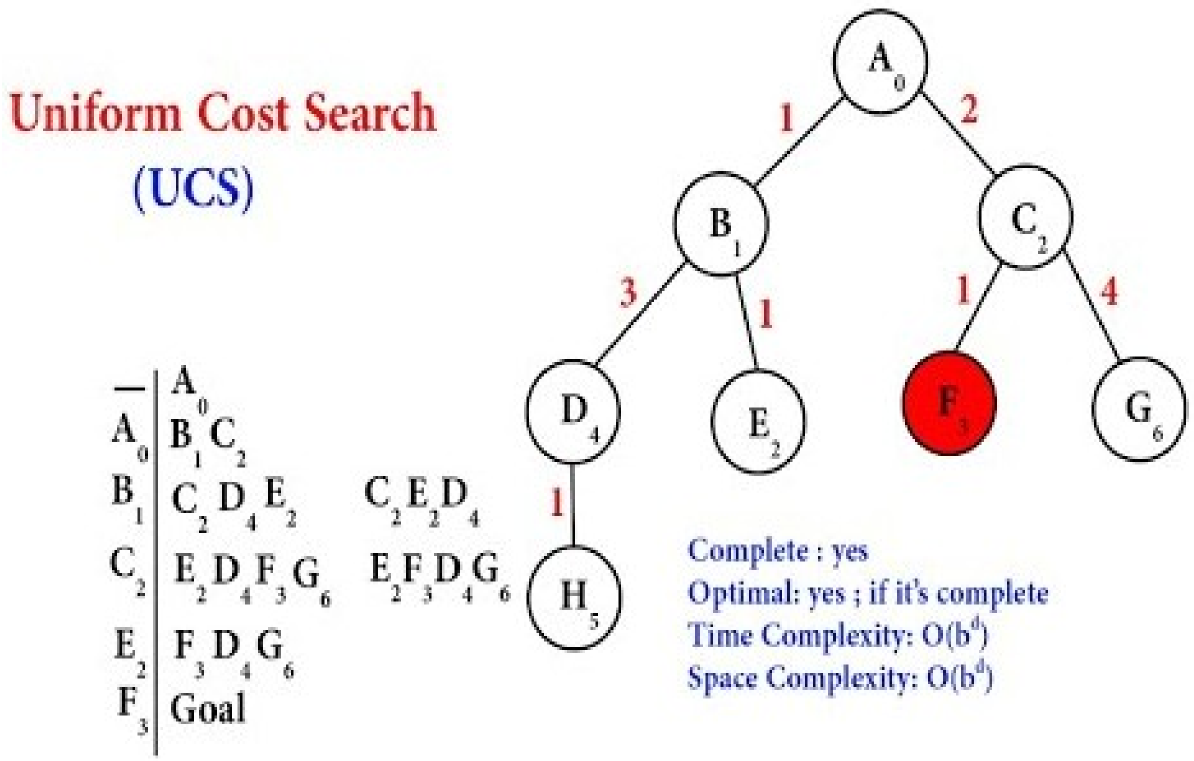
Completeness and optimality of Iterative Deepening Search

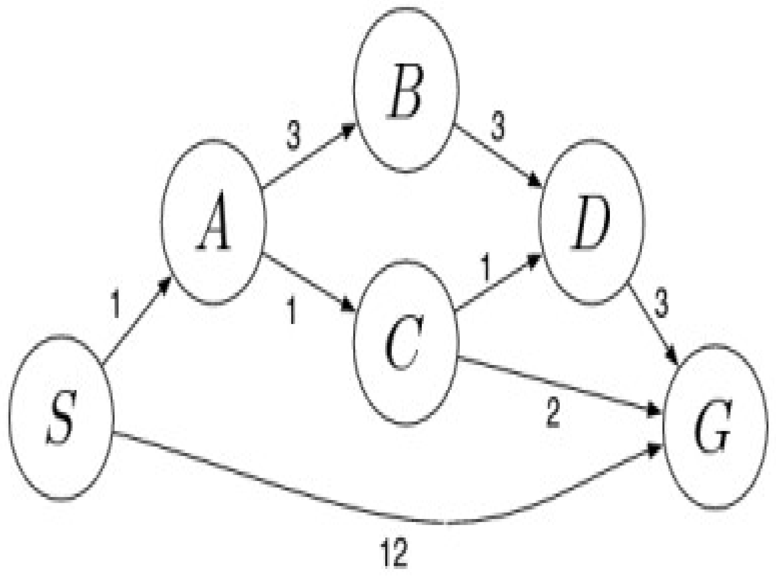
- Completeness
  - It is complete
  - It finds a solution if exists
- Optimality
  - It is optimal
  - Finds the shortest path (like breadth first)
    - Guarantee shortest path
    - Guarantee for goal node of minimal depth

Uniform-cost search

- **Uniform-cost search** is a searching algorithm used for traversing a weighted tree or graph.
- The **primary goal** of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.
- **Uniform-cost search** expands nodes according to their path costs form the **root node**.
- **A uniform-cost search algorithm** is implemented by the **priority queue**. It gives maximum priority to the lowest cumulative cost.
- **Uniform cost search** is equivalent to BFS algorithm if the path cost of all edges is the same.

- **Implementation**:
    - ***fringe* = queue ordered by path cost**





Bidirectional search

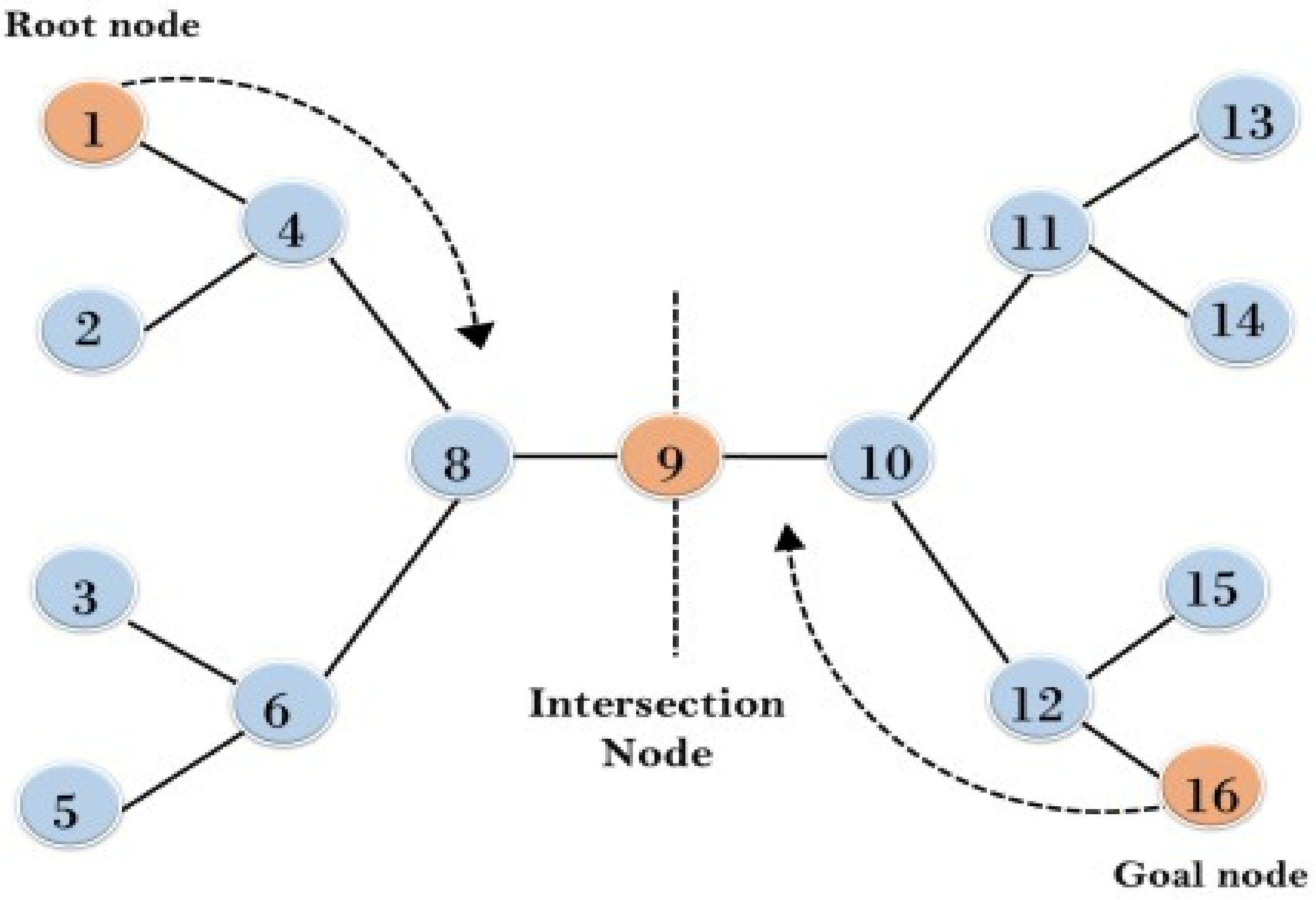
- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as **BFS**, **DFS**, **DLS**, etc.

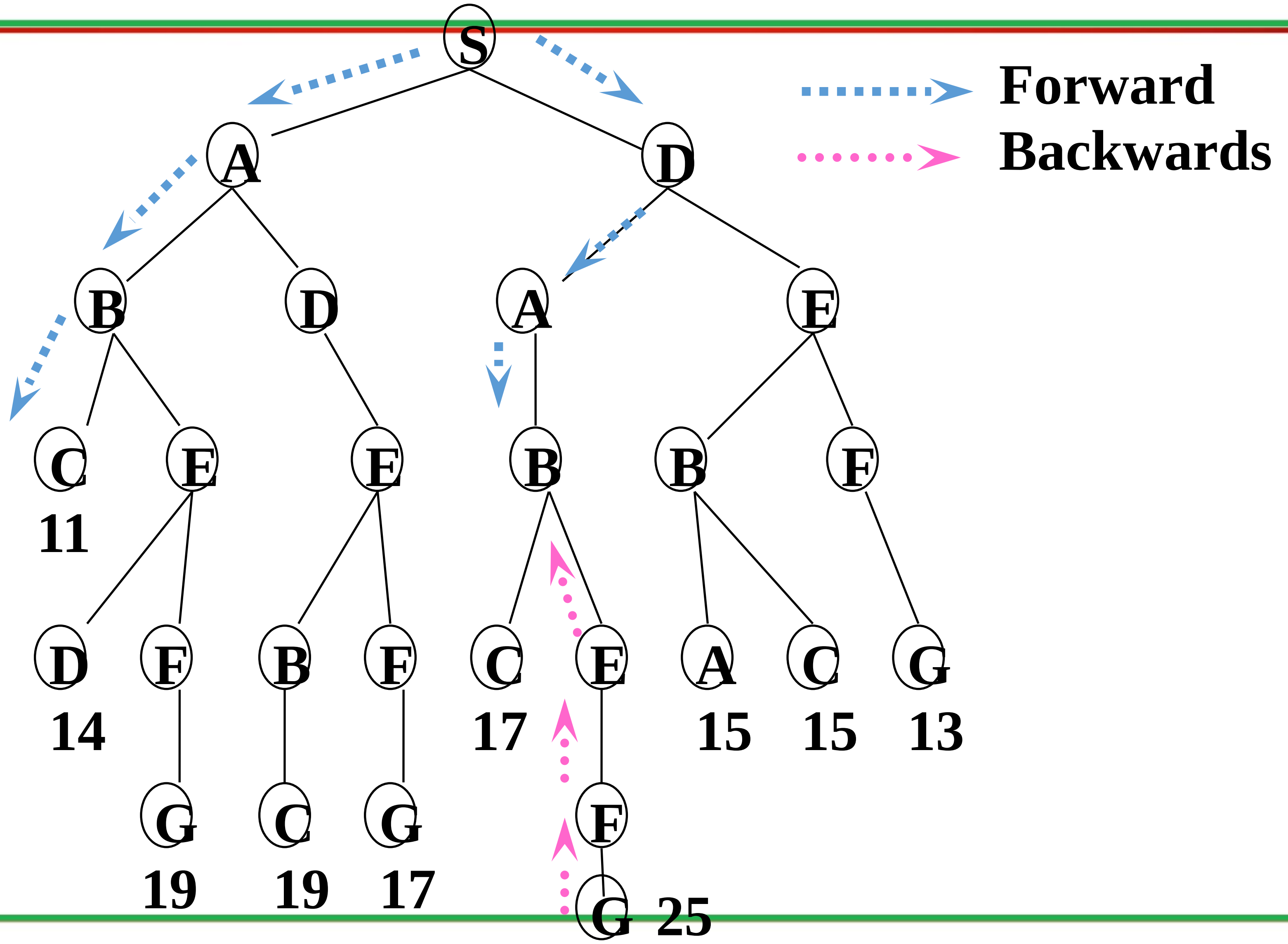
Advantages:

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

Disadvantages:

Implementation of the bidirectional search tree is difficult.

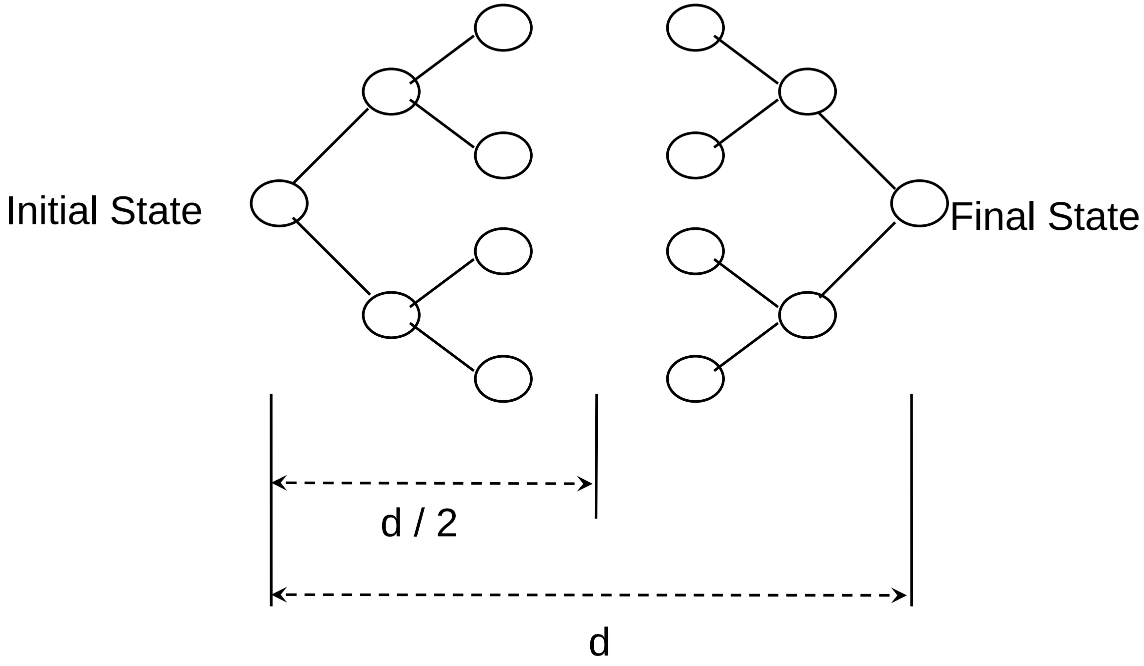

Root node
1
2
3
4
5
6
8
9
10
11
12
13
14
15
16
Intersection
Node
Goal node



Forward
Backwards
S
A
D
B
D
A
E
C
E
E
B
B
F
11
D
F
B
F
C
E
A
C
G
14
17
15
15
13
G
C
G
F
19
19
17
G
25

Search

- **Bi-directional Search**



- **Completeness**: yes
- **Optimality**: yes
- **Time complexity**: $O(b^{d/2})$
- **Space complexity**: $O(b^{d/2})$

$O(b^d)$ vs. **$O(b^{d/2})$?** with **b=10** and **d=6** results in **1,111,111** vs. **2,222.**

Informed search

- Informed search algorithms
  - Best-first search
  - Memory Bound Best First search
  - Iterative improvement algorithm (Local search algorithms)
- Use an evaluation function f(n)
- Admissible heuristics

Informed search algorithms

- **Informed search** is a **strategy** that **uses information about the cost** that may **incur** to **achieve the goal state** from the **current state**.
- **The information may not** be **accurate**. But **it will help** the agent to **make better decision**
- **This information** is called **heuristic information**

There **several algorithms** that belongs to **this group**. Some of these are:

- **Best-first search**
  1. Greedy best-first search
  2. A^* search
- **Memory Bound Best First search**
  1. Iterative deepening A* (IDA*) search
- **Iterative improvement algorithm (Local search algorithms)**
  1. Hill-climbing search
  2. Simulated annealing search

Best-first search

- **Idea:** use an evaluation function *f(n)* for each node
  - Estimate of "desirability“ using heuristic and path cost
  - Expand most desirable unexpanded node
- The information gives a clue about which node to be expanded first
- The best node according to the evaluation function may not be best

Implementation:

- Order the nodes in fringe in decreasing order of desirability (increasing order of cost evaluation function)

Best-First Search

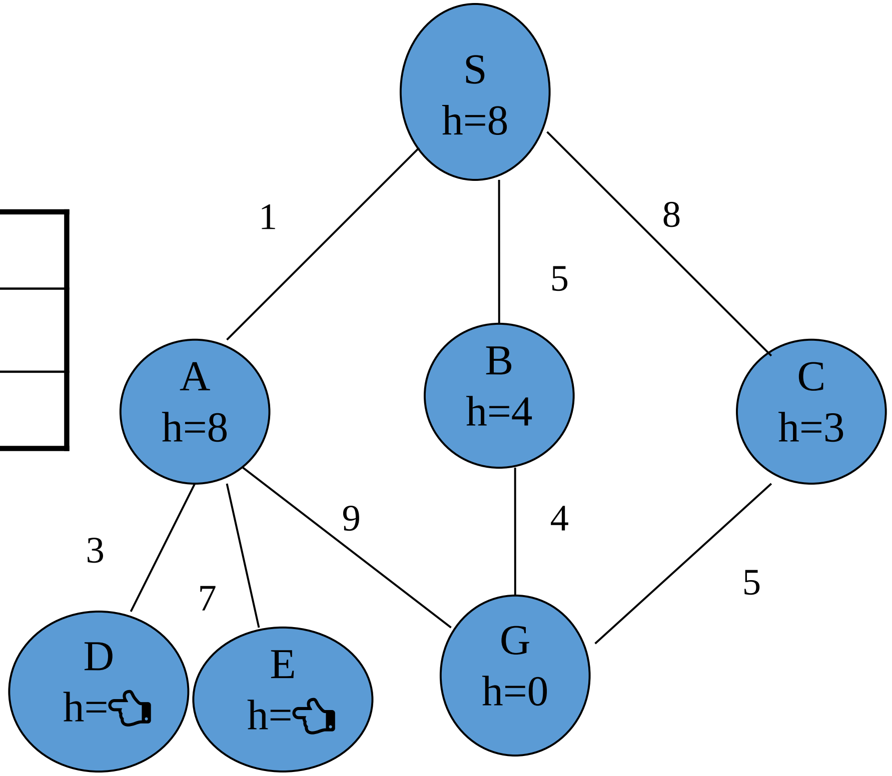
1. Put the **initial node** on a **list START**
2. If **(START is empty)** or **(START =GOAL)** **terminate search**
3. **Remove the first node** form **START**. **Call this node *n*.**
4. **If (*n* = GOAL)** **terminate search** with success.
5. **Else** **if node n** has **successor**, generate **all of them**. Find out **how far** the goal node. **Sort all the children generated** so far by the **remaining distance** from the **goal**.Name this **list** as **START1**
6. Replace **START** with **START1**
7. Goto Step**2**.

Greedy Search

$f(n) = h(n)$

# of nodes tested 1, expanded 1

Expanded Node	OPEN list
	(S:8)
S not goal	(C:3,B:4,A:8)

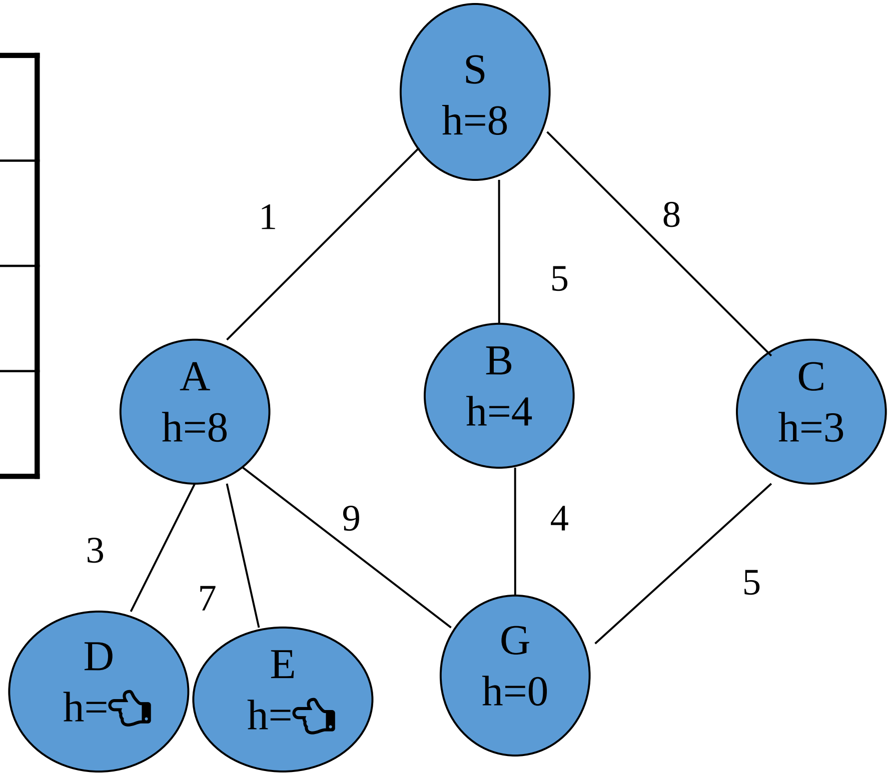


Greedy Search

$f(n) = h(n)$

\# of nodes tested 2, expanded 2

Expanded Node	OPEN list
	(S:8)
S	(C:3,B:4,A:8)
C not goal	(G:0,B:4,A:8)

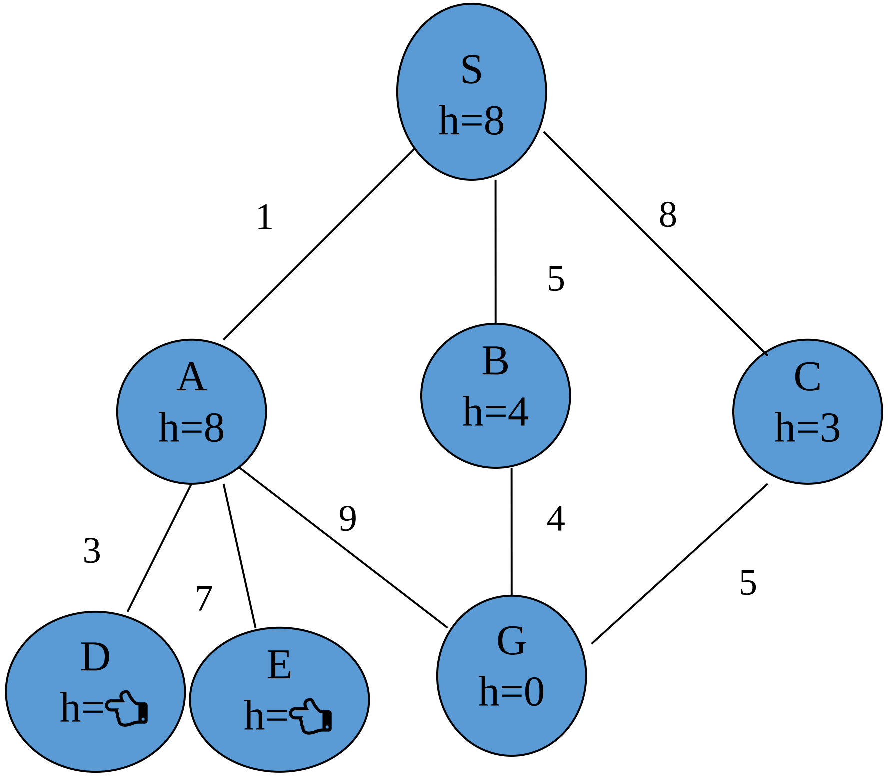


Greedy Search

$f(n)= h(n)$

# of nodes tested 3, expanded 2

Expanded Node	OPEN list
	(S:8)
S	(C:3,B:4,A:8)
C	(G:0,B:4,A:8)
G goal	(B:4.A:8) no expand



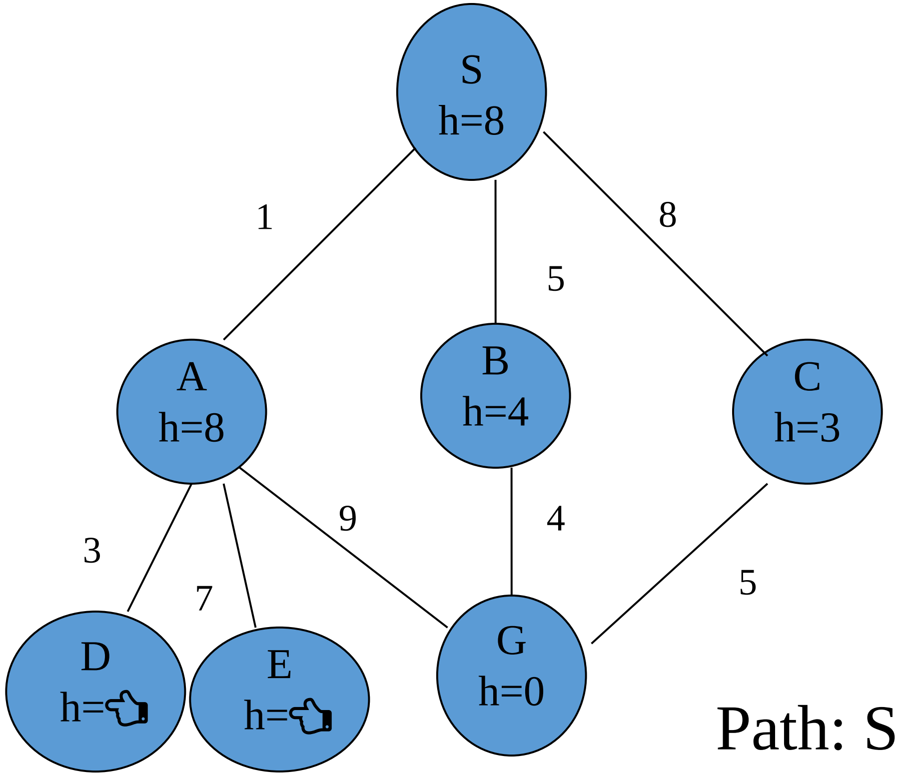
Greedy Search

f(n)= h(n)

# of nodes tested 3, expanded 2

Expanded Node	OPEN list
	(S:8)
S	(C:3,B:4,A:8)
C	(G:0,B:4,A:8)
G goal	(B:4.A:8)

* Fast but **not optimal**



Path: S,C,G

Cost: 13

Greedy best-first search

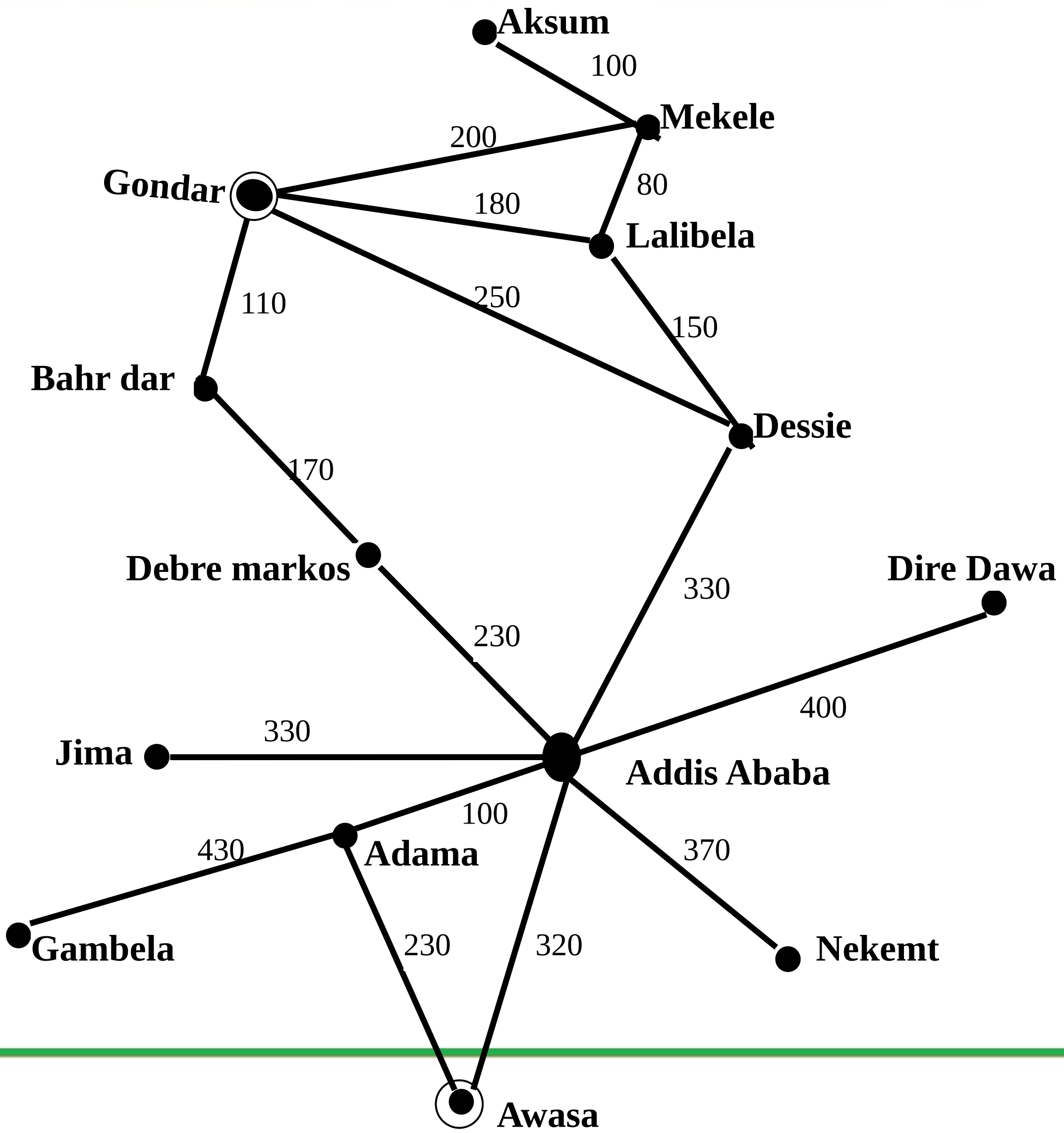
- Evaluation function $f(n) = h(n)$ (heuristic) = estimate of cost from n to *goal*
- That means the agent prefers to choose the action which is assumed to be best after every action.
- e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal (It tries to minimizes the estimated cost to reach the goal)

Example One

Greedy best-first search example

Show the flow to move from Awasa to Gondar using the given road map graph

Ethiopia Map with step costs in km

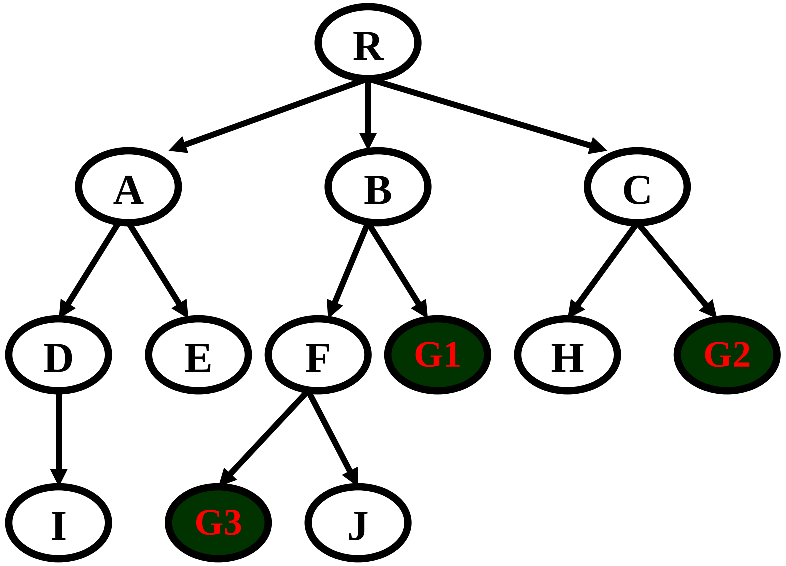


Straight Line distance to Gondar

Gondar	0
Aksum	100
Mekele	150
Lalibela	110
Desseie	210
Bahrdar	90
Debre Markos	170
Addis Ababa	321
Jima	300
Diredawa	350
Adama	340
Gambela	410
Awasa	500
Nekemt	420

Example Two:- Greedy best-first search example

- **Given the following tree structure, show the content of the open list and closed list generated by Greedy best first search algorithm.**



Heuristic

R ✉ G -------------- 100
A ✉ G -------------- 60
B ✉ G -------------- 80
C ✉ G -------------- 70
D ✉ G -------------- 65
E ✉ G -------------- 40
F ✉ G -------------- 45
H ✉ G ---------------10
I ✉ G ---------------- 20
J ✉ G ---------------- 8
G_1,G_2,G_3 ✉G ------------ 0

Properties of greedy best-first search

- Complete? Yes if repetition is controlled otherwise it can can get stuck in loops
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$, keeps all nodes in memory
- Optimal? No

A^* search

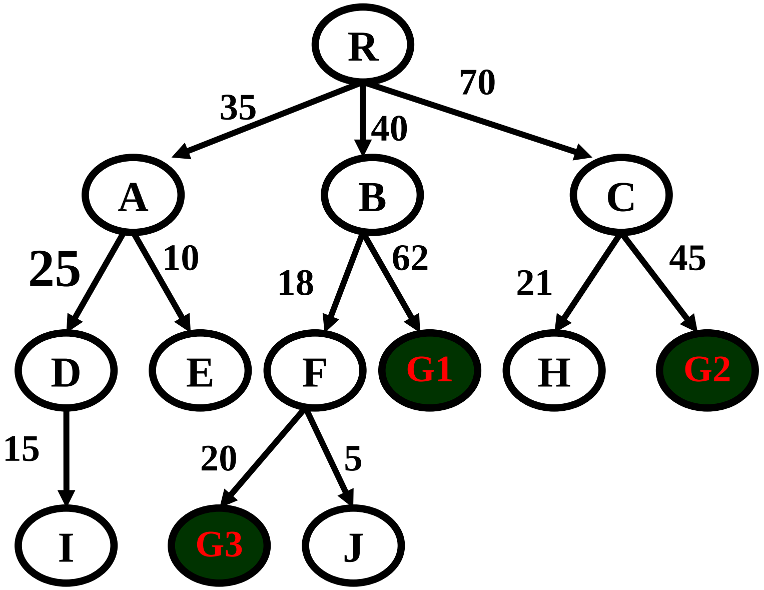
- Idea: Avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$ *where*
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal
- It tries to minimizes the total path cost to reach into the goal at every node N.

- Example two

by using map of Ethiopia in previous slid , Indicate the flow of search to move from Awasa to Gondar using A*

Example Two

- Given the following tree structure, show the content of the open list and closed list generated by A* best first search algorithm.



Heuristic

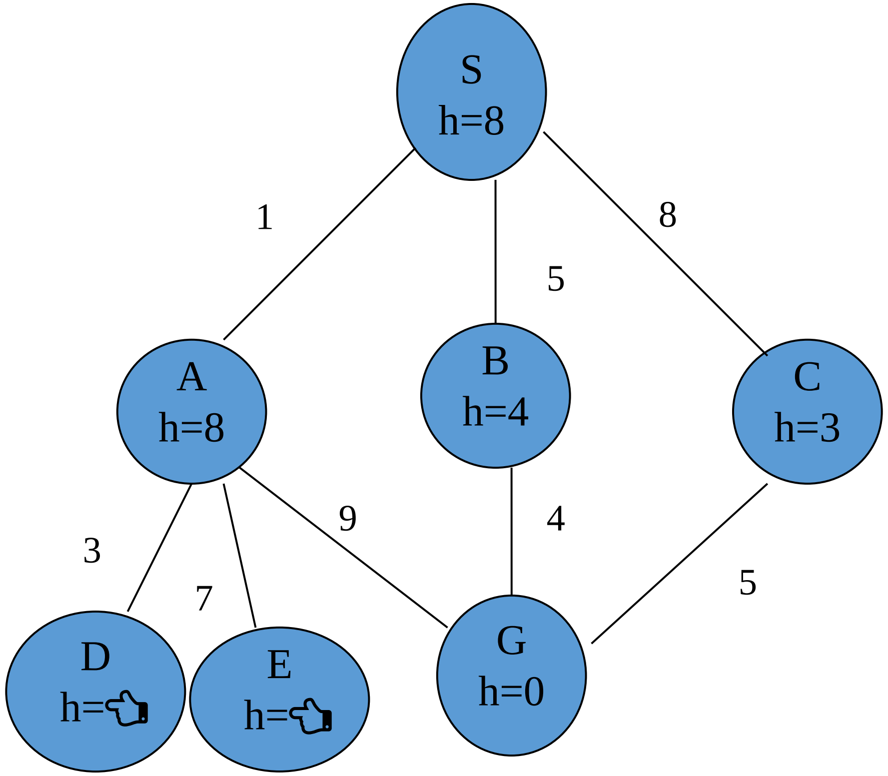
R ✉ G -------------- 100
A ✉ G -------------- 60
B ✉ G -------------- 80
C ✉ G -------------- 70
D ✉ G -------------- 65
E ✉ G -------------- 40
F ✉ G -------------- 45
H ✉ G ---------------10
I ✉ G ---------------- 20
J ✉ G ---------------- 8
G_1,G_2,G_3 ✉G ------------ 0

Admissible heuristics

- A heuristic $h(n)$ is admissible if for every node n,
  $h(n) \leq h^*(n)$, 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
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem:
  If $h(n)$ is admissible, A^* using TREE-SEARCH is optimal

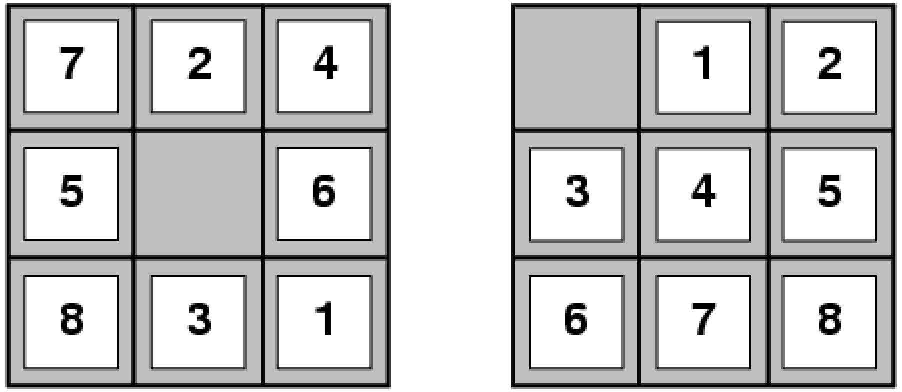
Example

n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0



Since $h(n) \le h^*(n)$ $\forall n$, ***h* is admissible.**

Find Admissible heuristics for the 8-puzzle?



Start State

Goal State

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance (i.e., no. of squares from desired location of each tile). This is also called city cap distance

- $h_1(S)$ = ?
- $h_2(S)$ = ?

Iterative Improvement Algorithm (Local search algorithms)

- Iterative best improvement is a local search algorithm that selects a successor of the current assignment that most improves some evaluation function.
- If there are several possible successors that most improve the evaluation function, one is chosen at random.
- When the aim is to minimize, this algorithm is called greedy descent.
- When the aim is to maximize a function, this is called hill climbing or greedy ascent. We only consider minimization; to maximize a quantity, you can minimize its negation

Iterative Improvement Algorithm (Local search algorithms)

- There are two types of Iterative Improvement algorithms
  - Hill climbing if the evaluation function is quality
    - also called Gradient Descent if the evaluation function is a cost rather than a quality
  - Simulated Annealing

1. Hill-climbing search

- Tries to make changes that improve the current state cost
- It continually move in the direction of increasing value
- The node data structure maintain only records of state and evaluation cost

Hill-climbing (Gradient Descent) search

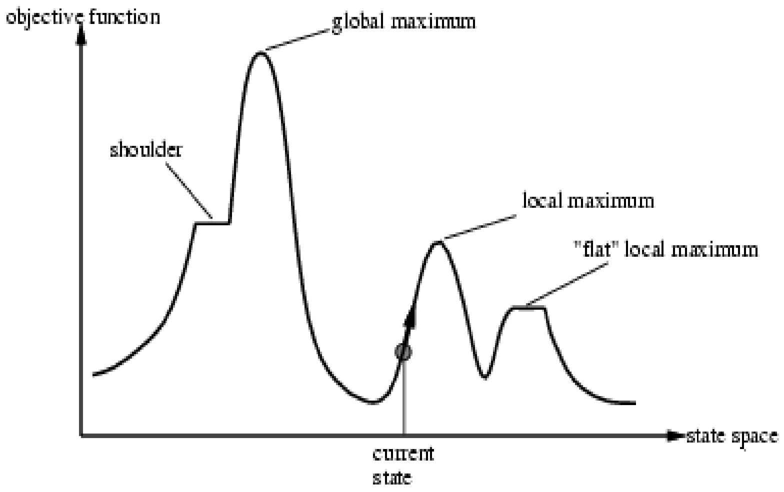
- **Tries to make changes that improve the current state cost**

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```

Problem:

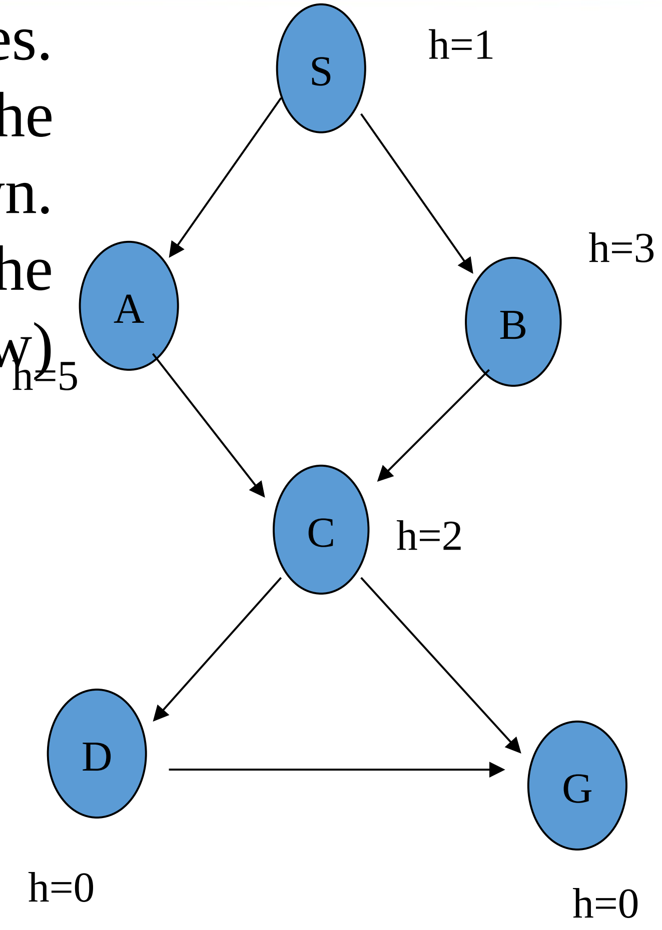
1. Depending on initial state, can **get stuck** in **local maxima**
2. **Plateaux** (after some progress the algorithm will make a random walk)
3. **Ridges** (a place where two sloppy sides meet). In this case the search may oscillate from side to side



Exercise 1

Consider the search space with start S and goal G states. The value of heuristics *h* are shown at each node. The cost associated with the each arc is not known. However, the trace of the OPEN list produced by the execution of A* algorithm is available(given below) along with the *f* values.

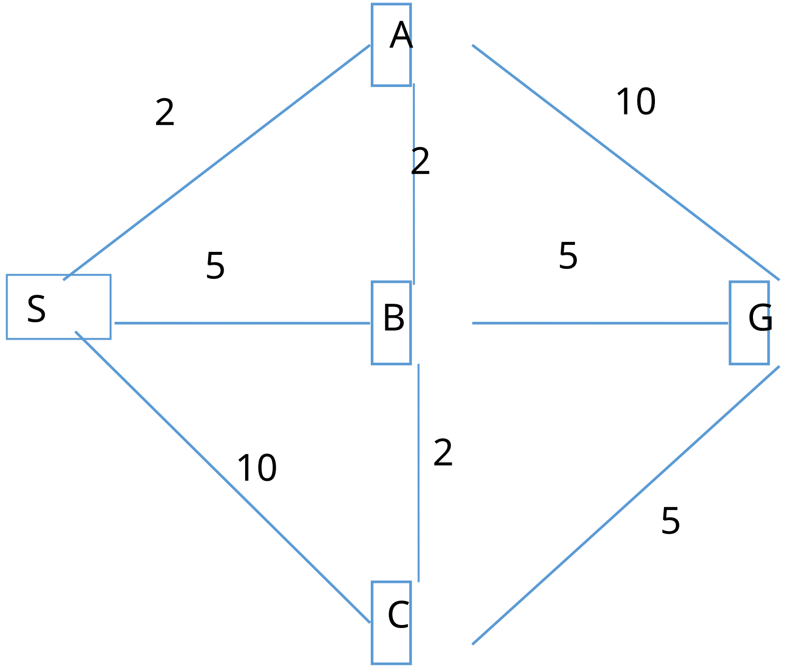
Determine the arc cost of arcs.



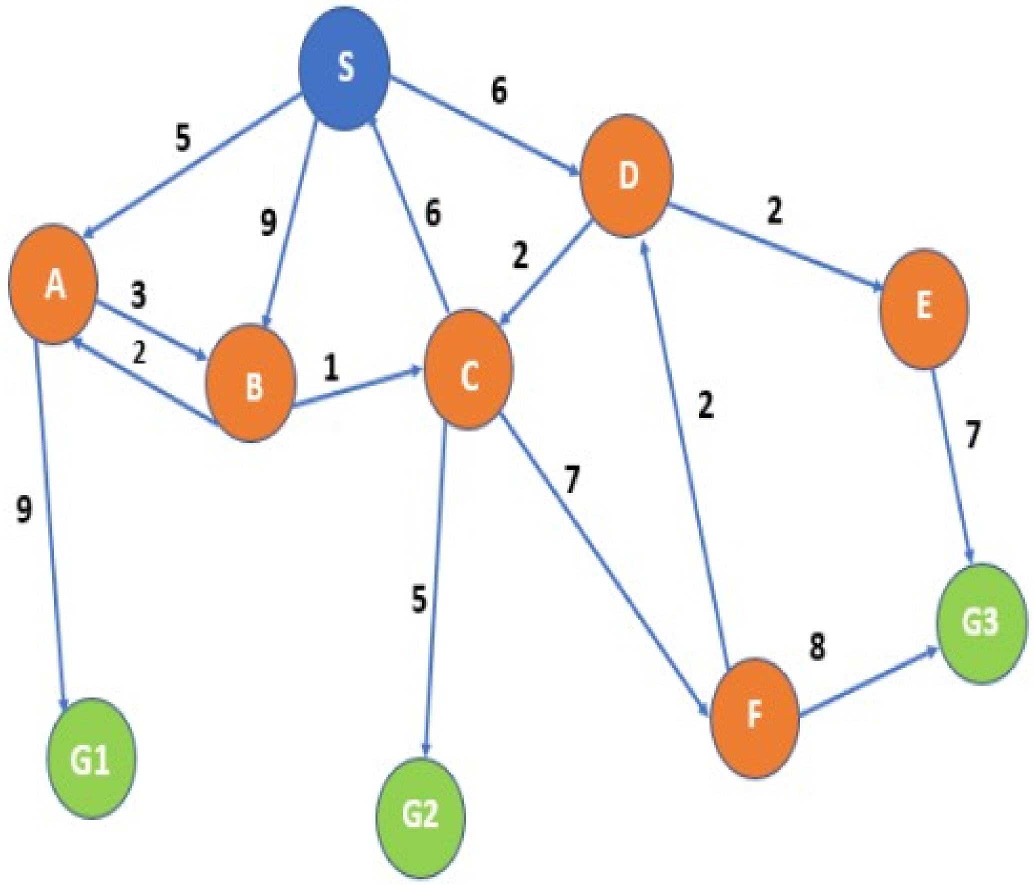
1. { (S, f=1)}
2. {(B, f=5), (A, f=6)}
3. {(A, f=6), (C, f=7)}
4. {(C, f=6)}
5. {(D, f=5), (G, f=7)}
6. {(G,f=6)}

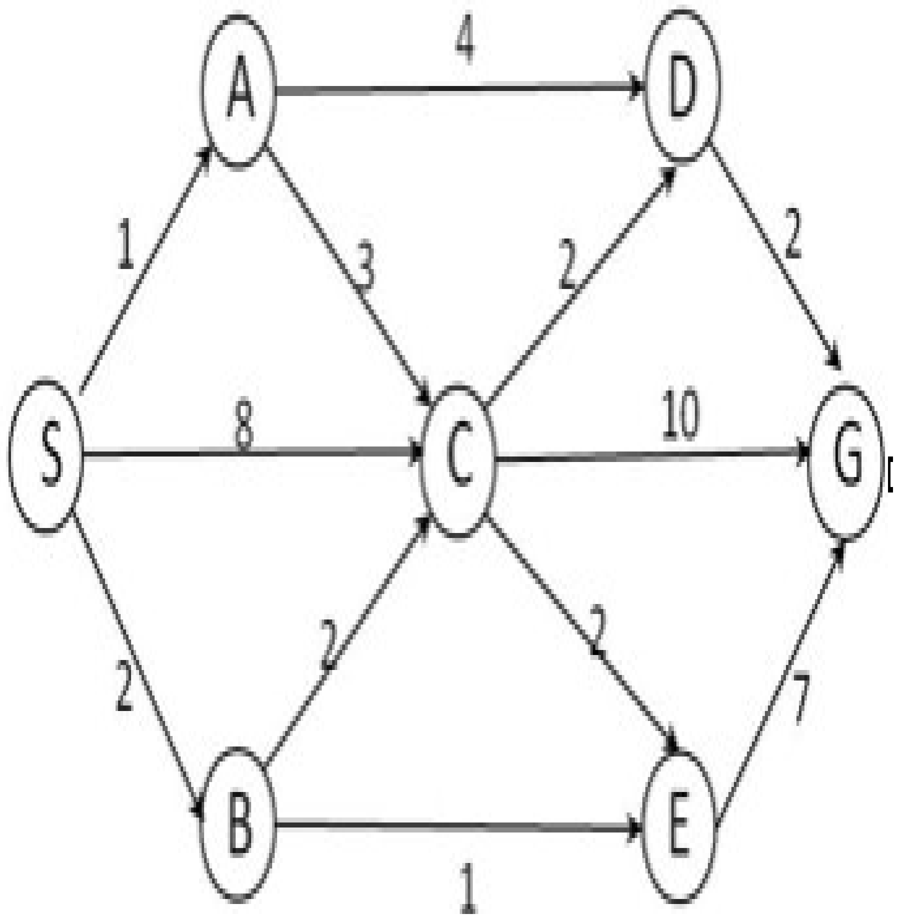
Exercise 2. list out all nodes to reach the goal with Greedy best and A* search and which one is optimal

S	A	1
S	B	3
S	C	5
S	G	9
A	B	1
A	C	3
A	G	3
B	C	2
B	G	4
C	G	4

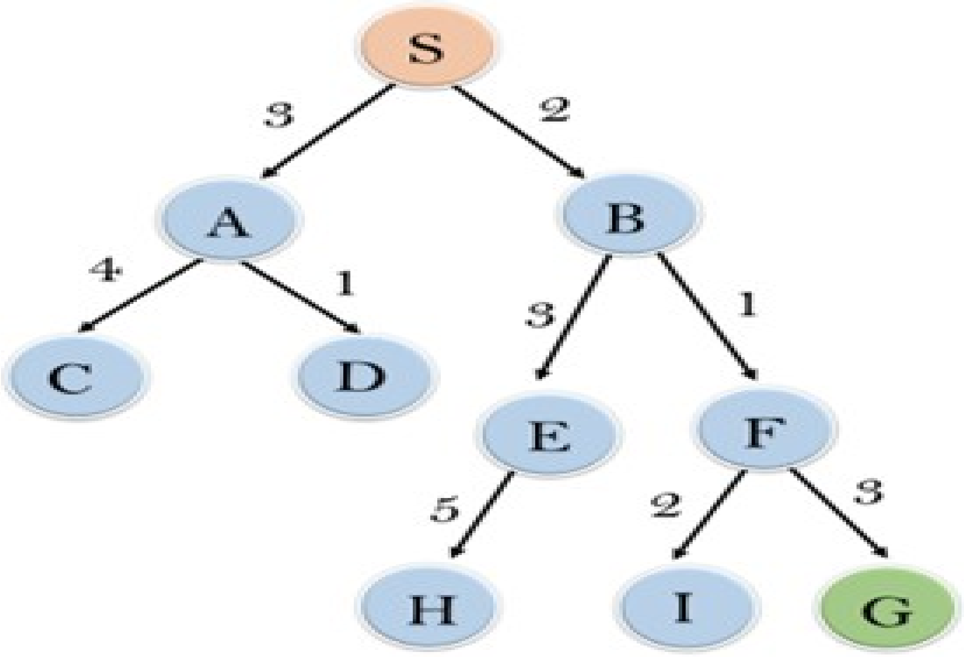


Exercises





Exercises



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

Thanks!!!