Searching Techniques & Algorithms in AI

Prof. Dr. A K M Akhtar Hossain
Dept. of CSE, RU

Searching Techniques

- Almost every AI program depends on a search procedure to perform its prescribed functions.
- Problems are typically defined in terms of states, and solutions correspond to goal states.
- Solving a problem then amounts to searching through the different states until one or more of the goal states are found.

Searching Techniques Conti..

- Basically two types:

1. Blind or Uninformed Search
   I. Breadth-First Search
   II. Depth-First search
   III. Bi-directional Search
2. Informed Or Directed search
   I. Heuristics search
   II. Hill Climbing Search
   III. Best-First Search
   IV. A^* search

Blind or Uninformed Search

- In blind or uninformed search, no preference is given to the order of successor node generation and selection.
- The path selected is blindly nor mechanically followed.
- No information is used to determine the preference of one child over another.

Informed Or Directed search

- In informed or directed search, some information about the problem space is used to compute a preference among the children for exploration and expansion.
- Before proceeding with a comparison of strategies, we consider next some typical search problems.

Blind or Uninformed Search

- ## Breadth-First Search (BFS):
- Breadth-First searches are performed by exploring all nodes at a given depth before proceeding to the next level.
- This means that all immediate children of the nodes are explored before any of the children’s children are considered.
- Following Figure shows the Breadth-First Search.

Breadth-First Search(BFS) Graphical Diagram

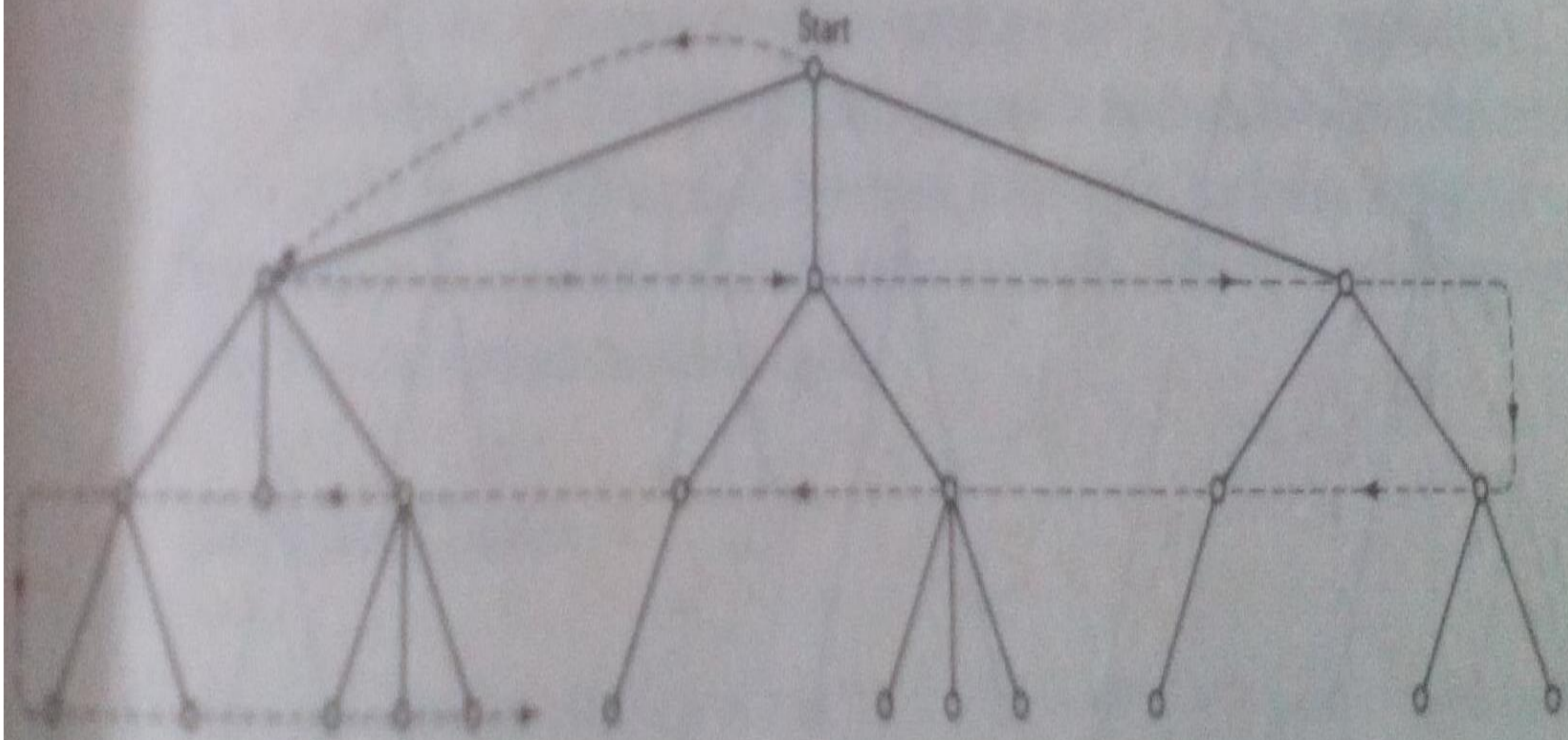


Figure 9.7   Breadth-first search of a tree.

Breadth-First Search (BFS)

- **Breadth-first search** (**BFS**) is an algorithm for traversing or searching tree or graph data structures.
- It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a **search key**) and explores the neighbor nodes first, before moving to the next level neighbors.

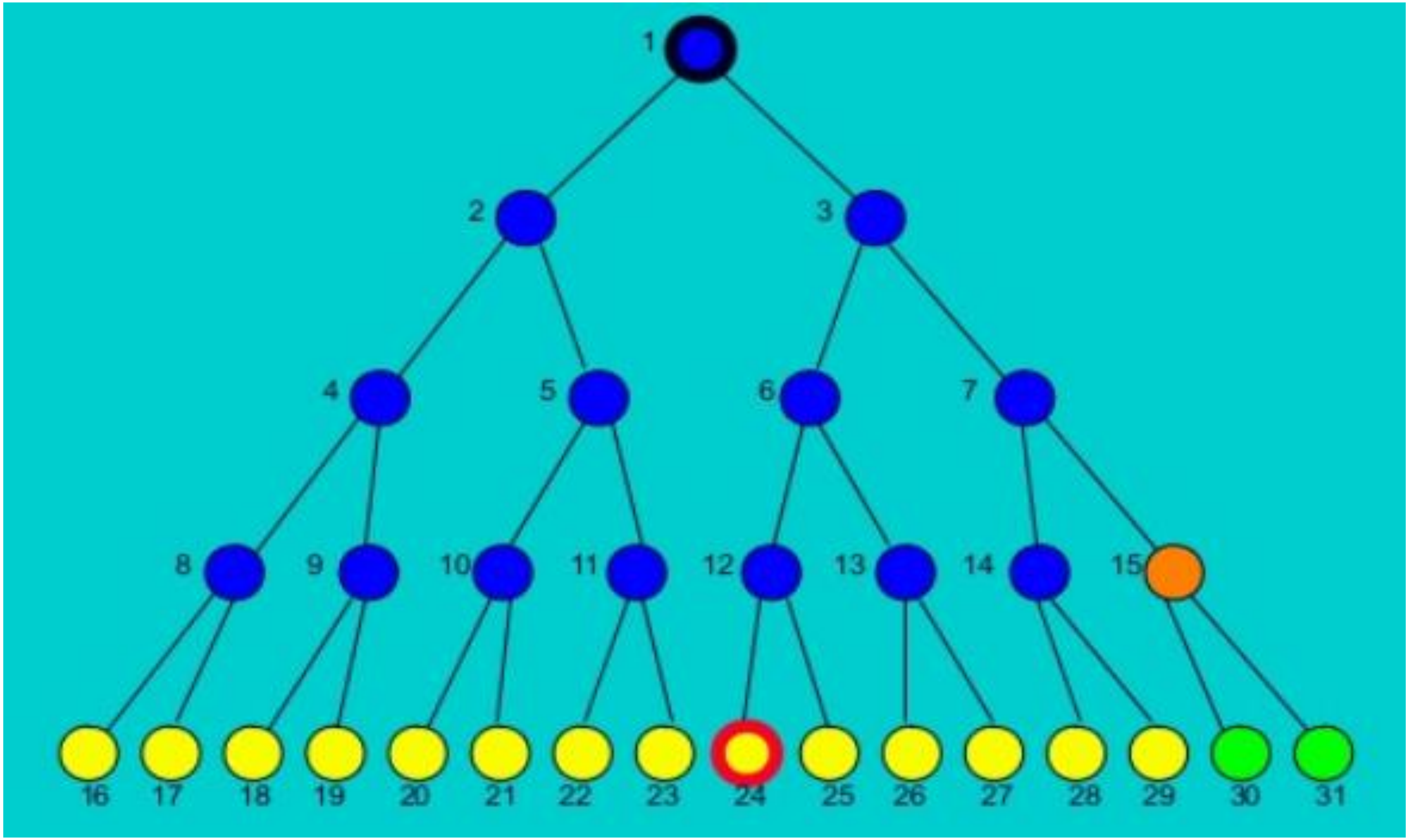
Breadth-First Search

- **<u>Algorithm:</u>**
- **<u>Step 1:</u>** Place the starting node **S** on the queue.
- **<u>Step 2:</u>** If the queue is empty, return failure and stop.
- **<u>Step 3:</u>** If the first element on the queue is a goal node, **g**, return success and stop.
- otherwise,

Breadth-First Search Conti…

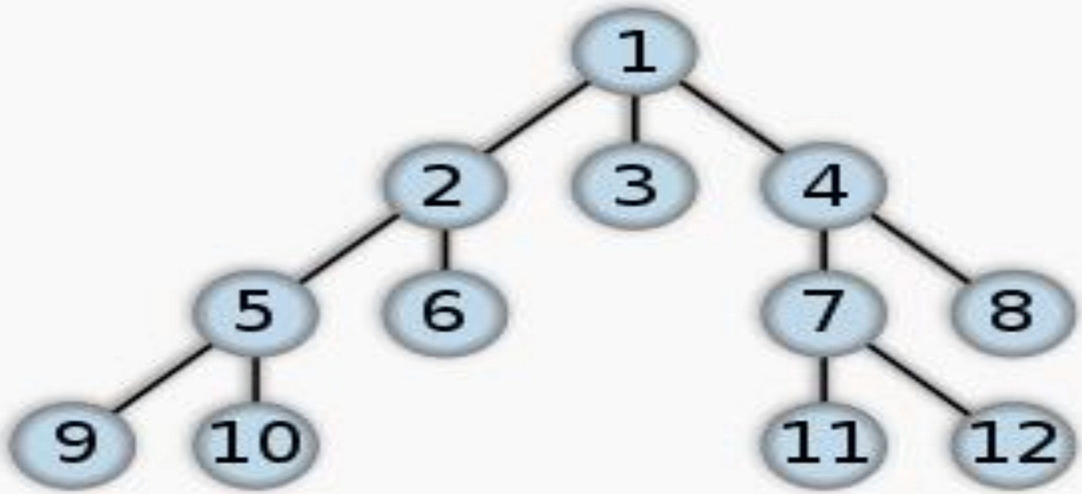
- **Step 4:** Remove and expand the first element from the queue and place all the children at the end of the queue in any order.
- **Step 5:** Return to **Step 2**.

Breadth-First Search(BFS) Graphical Diagram



Breadth-First Search(BFS)

Breadth-first search



Order in which the nodes are expanded

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\|E\|) = O(b^d)$
Worst case space complexity	$O(\|V\|) = O(b^d)$

Performance of BFS

- Time complexity
  - In worst case BFS must generate all nodes up to depth d

    $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$

    - Note on average, half of the nodes at depth d must be examined.
  - So average case time complexity is $O(b^d)$
- Space complexity
  - Space required for storing nodes at **depth d** is $O(b^d)$

Depth-First Search (DFS)

- Depth-first searches are performed by diving downward into a tree as quickly as possible.
- It does this by always generating a child node from the most recently expanded node, then generating that child's children, and so on until a goal is found or some cutoff depth point d is reached.

Depth-First Search Conti….

- If a goal is not found when a leaf node is reached or at the cutoff point, the program backtracks to the most recently expanded node and generates another of its children.
- This process continues until a goal is found or failure occurs.

Depth-first search (DFS)

- **Depth-first search** (**DFS**) is an algorithm for traversing or searching tree or graph data structures.
- One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

Depth-first search (DFS) Graphical Diagram

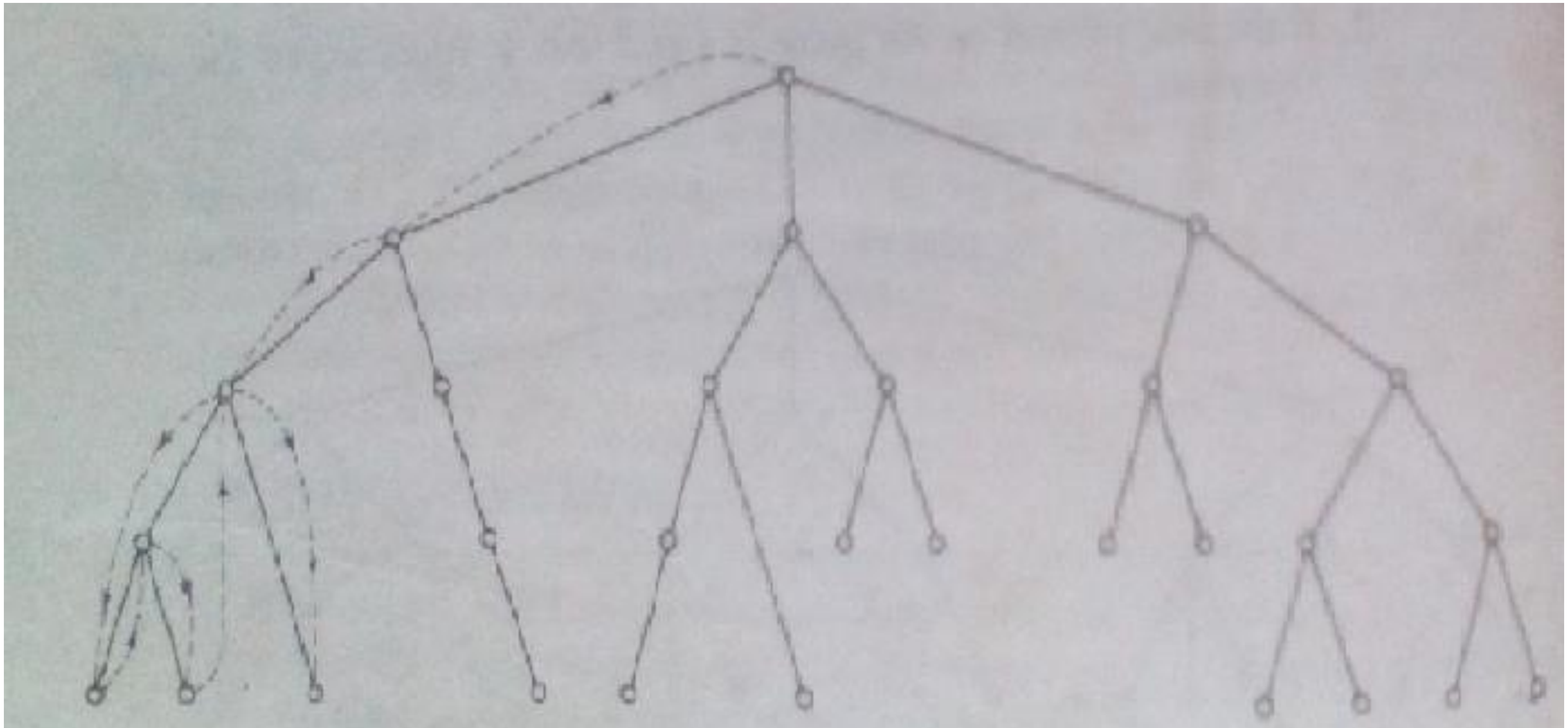


Figure 9.8 Depth-first search of a tree.

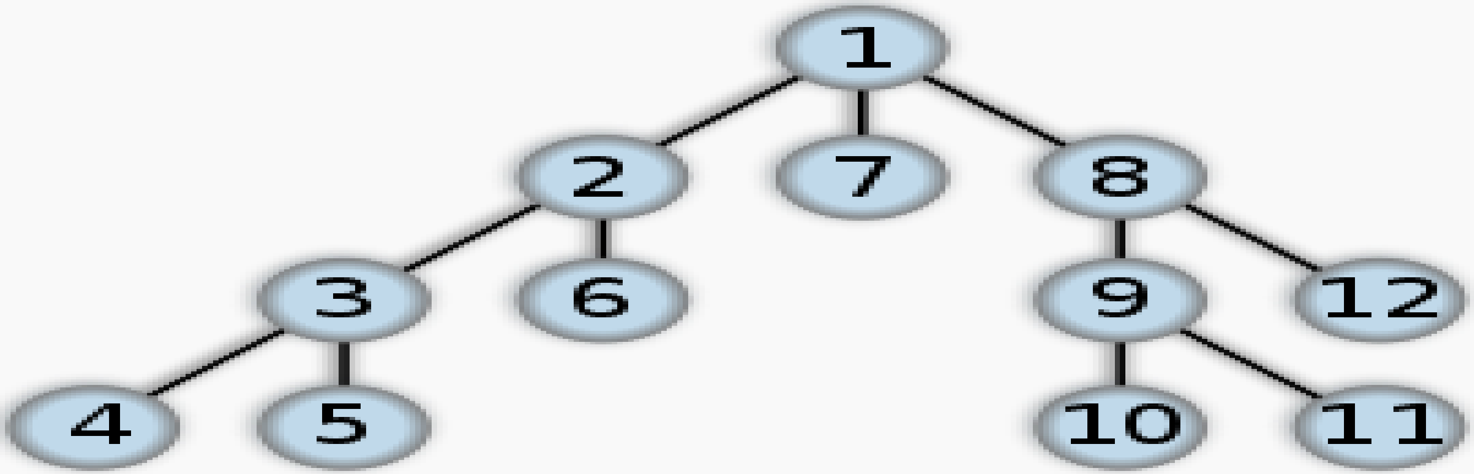
Depth-First Search Conti…

- **Algorithm:**
- **Step 1**: Place the starting node **s** on the queue.
- **Step 2:** If the queue is empty, return failure and stop.
- **Step 3:** If the first element on the queue is a goal node **g**, return success and stop.
- Otherwise,

Depth-First Search Conti…

- **<u>Step 4</u>**: Remove and expand the first element, and place the children at the front of the queue (in any order).
- **<u>Step 5</u>**: Return to **<u>Step 2</u>**.
- /**************************/
- The depth-first search is preferred over the breadth-first when the search tree is known to have a plentiful number of goals.
- Otherwise, depth-first may never find a solution.

Depth-first search



Order in which the nodes are visited

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\|E\|)$ for explicit graphs traversed without repetition, $O(b^d)$ for implicit graphs with branching factor b searched to depth d
Worst case space complexity	$O(\|V\|)$ if entire graph is traversed without repetition, O(longest path length searched) for implicit graphs without elimination of duplicate nodes

Performance of DFS

- Time complexity
  - In worst case time complexity is $O(b^d)$
- Space complexity
  - If the depth cut off is d the space requirement is $O(d)$
- DFS requires an arbitrary cut off depth.
  - If branches are not cut off and duplicates are not checked for, the algorithm may not terminate.

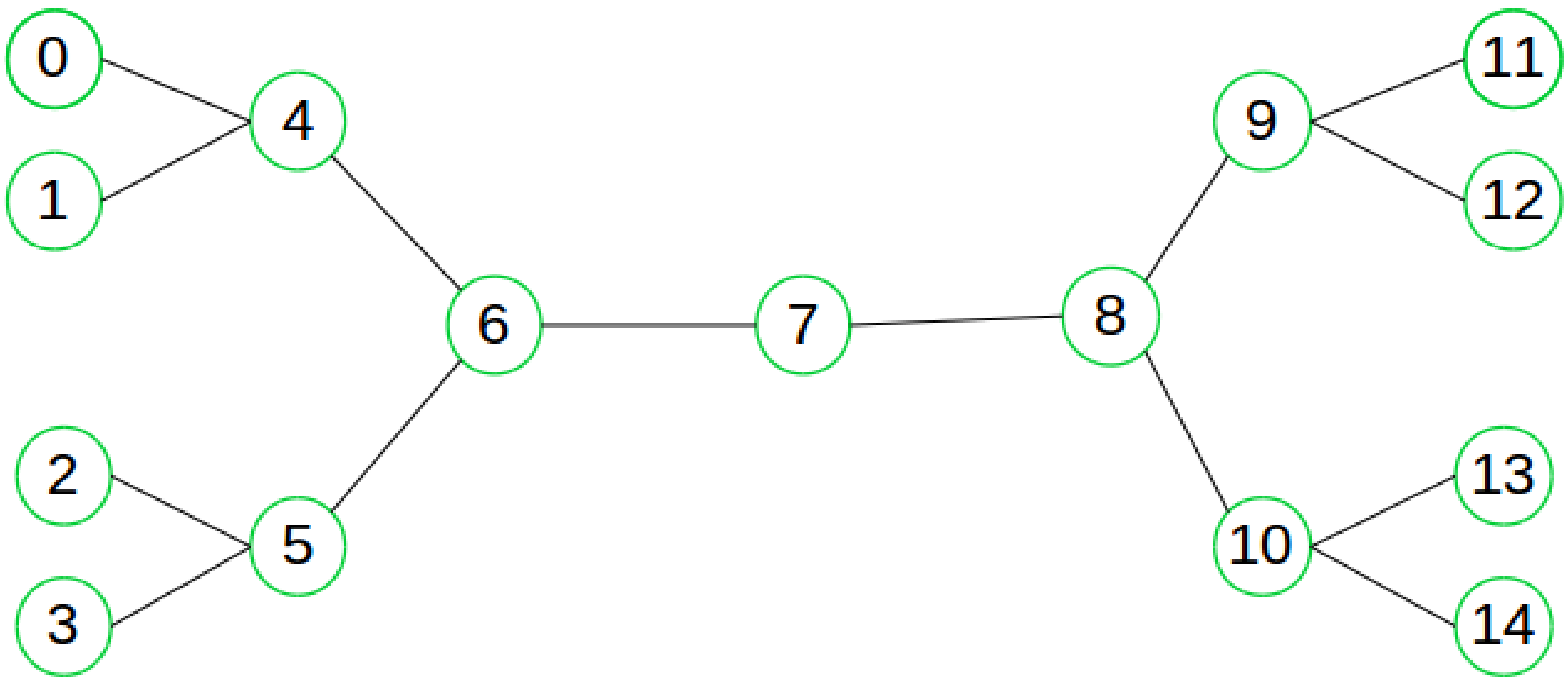
Bi-directional Search

- Bidirectional search is a graph search algorithm which find smallest path form source to goal vertex/node.
- It runs two simultaneous search:
    - Forward search form source/initial vertex toward goal vertex
    - Backward search form goal/target vertex toward source vertex

Bi-directional Search

- Bidirectional search replaces single search graph with two smaller sub graphs, one starting from initial vertex and other starting from goal vertex.
- **The search terminates when two graphs intersect.**
- Bidirectional search can be guided by a heuristic estimate of remaining distance from source to goal and vice versa for finding shortest path possible.

Bi-directional Search Graphical Diagram



Bi-directional Search

- Consider above simple example:
- Suppose we want to find if there exists a path from vertex 0 to vertex 14.
- Here we can execute two searches, one from vertex 0 and other from vertex 14.
- When both forward and backward search meet at **vertex 7**, we know that we have found a path from node 0 to 14 and search can be terminated now.

Performance measures

- Suppose if branching factor of tree is **b** and distance of goal vertex from source is **d**, then the normal BFS/DFS searching complexity would be $O(b^d)$.
- On the other hand, if we execute two search operation then the complexity would be $O(b^{d/2})$ for each search and total complexity would be $O(b^{d/2} + b^{d/2})$ which is far less than $O(b^d)$.

When to use bidirectional approach?

- We can consider bidirectional approach when,
    - Both initial and goal states are unique and completely defined.
    - The branching factor is exactly the same in both directions.

Informed Or Directed search

Informed Or Directed search:

I. Heuristics search
II. Hill Climbing Search
III. Best-First Search
IV. A^* search

Heuristics search

- **Heuristic search** refers to a **search** strategy that attempts to optimize a problem by iteratively improving the solution based on a given **heuristic function** or a cost measure.
- Heuristic search is a "rule of thumb" which is used to help guiding the search.
- It is something learned experientially and recalled when needed.
- **Heuristic Function:**
- It is a function which is applied to a state in a search space to indicate a likelihood of success if that state is selected.

Heuristics search

- Heuristic Search –
  - ❖ given a search space, a current state and a goal state
  - ❖ generate all successor states and evaluate each with our heuristic function.
  - ❖ select the move that yields the best heuristic value
- Here in the associated notes, we examine various heuristic search algorithms
  - ❖ heuristic functions can be generated for a number of problems like games.

Heuristic Search Technique

- In order to find out the heuristic values, we have to consider the following equation (**Heuristic Function**) to select the best value:
- $f(n) = g(n) + h(n)$
- Where,
  - f(n)= cost of selecting state n
  - g(n) = cost of reaching state n from the start state
  - h(n)=heuristic value for state n

Example for Heuristic Search
8-puzzle Problem:

We Know,
f(n) = g(n) + h(n)
f(n) = 0 + 4
So, f(n) = 4

Where, g(n) = 0
h(n) = 4

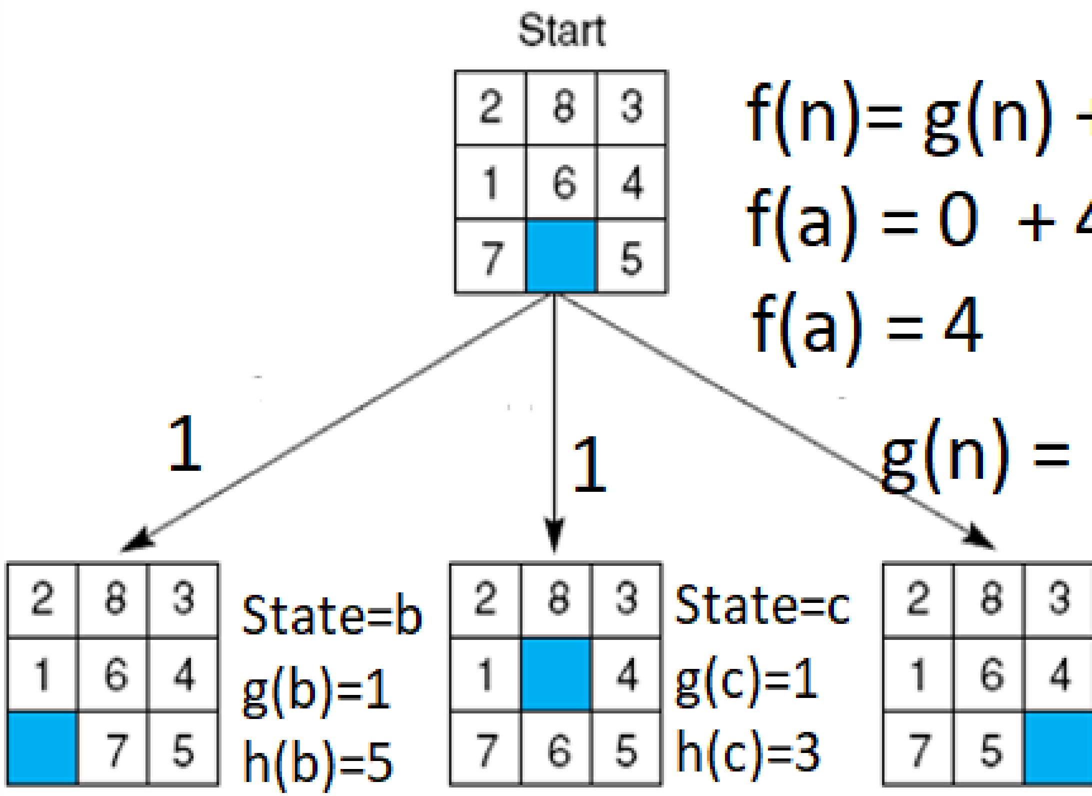
Start

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

Goal

Example Heuristic Search



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

$f(a) = 0 + 4$

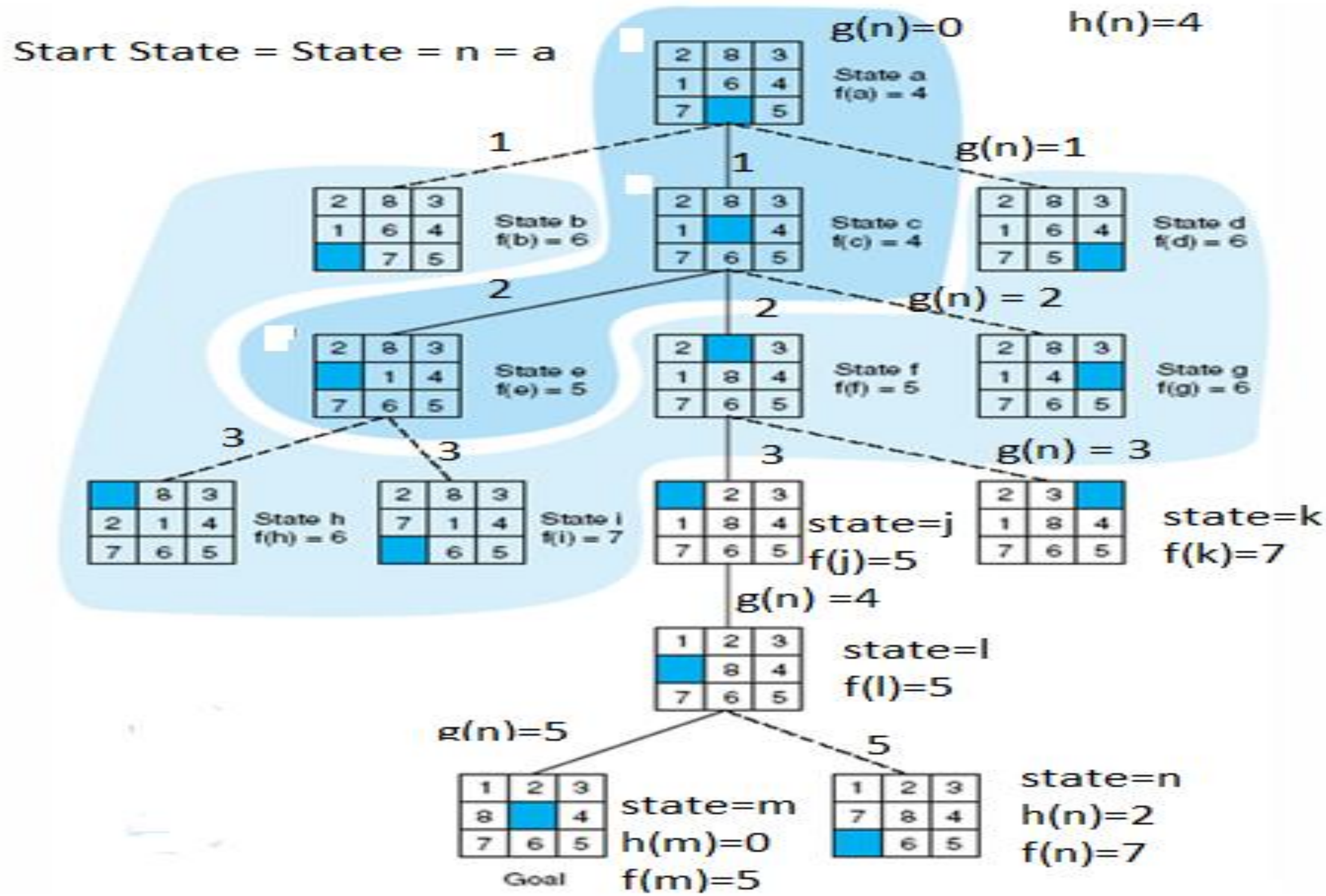
$f(a) = 4$

State=b
g(b)=1
h(b)=5
f(b)=6

State=c
g(c)=1
h(c)=3
f(c)=4

State=d
g(d)=1
h(d)=5
f(d)=6

Example Heuristic Search



Hill Climbing Methods

- **Hill climbing is searching where the most promising child is selected for expansion.**
- When the children have been generated, alternative choices are evaluated using some type of heuristic function.
- The path that appears most promising is then chosen and no further reference to the parent or other children is retained.

Hill Climbing Methods Conti…

- This process continues from node-to-node with previously expanded nodes being discarded.
- **<u>Example</u>**:   Suppose you are in an unfamiliar city without a map and you want to get downtown. You simply aim for the tall buildings.

Hill Climbing Methods Conti…

- The heuristic function is just distance between the current location and the location of the tall buildings and the desirable states are those in which this distance is minimized.
- Hill climbing can terminate whenever a goal state is reached.

Figure of Hill Climbing Searching

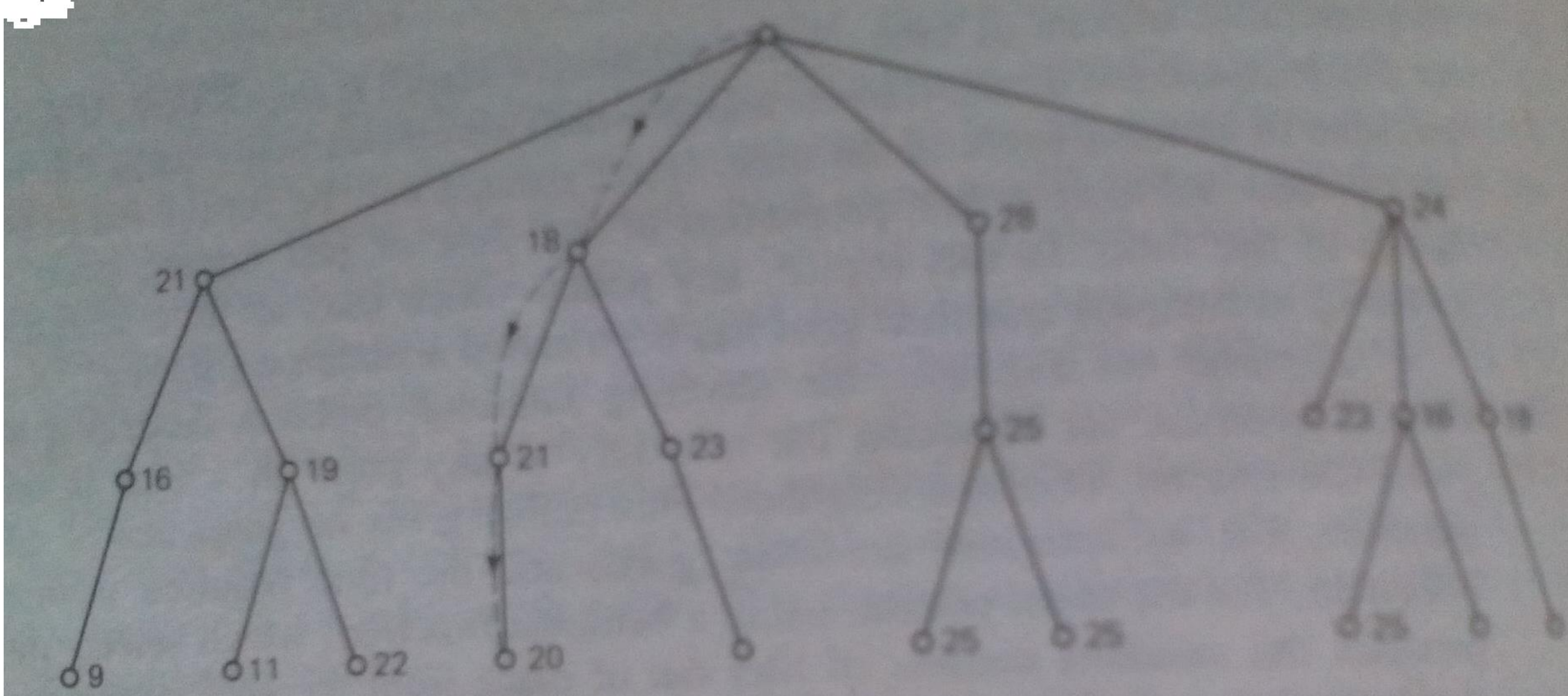


Figure 9.9 Search with hill climbing

Hill Climbing Searching(HCS)

- **Algorithm:**
- **Step 1:** Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
- **Step 2:** Loop until a solution is found or until there are no new operators left to be applied in the current state:

Hill Climbing Searching Algorithm Conti…

(a). Select an operator that has not yet been applied to the current state and apply it to produce a new state.

(b). Evaluate the new state.

(i). If it is a goal state, then return it and quit.

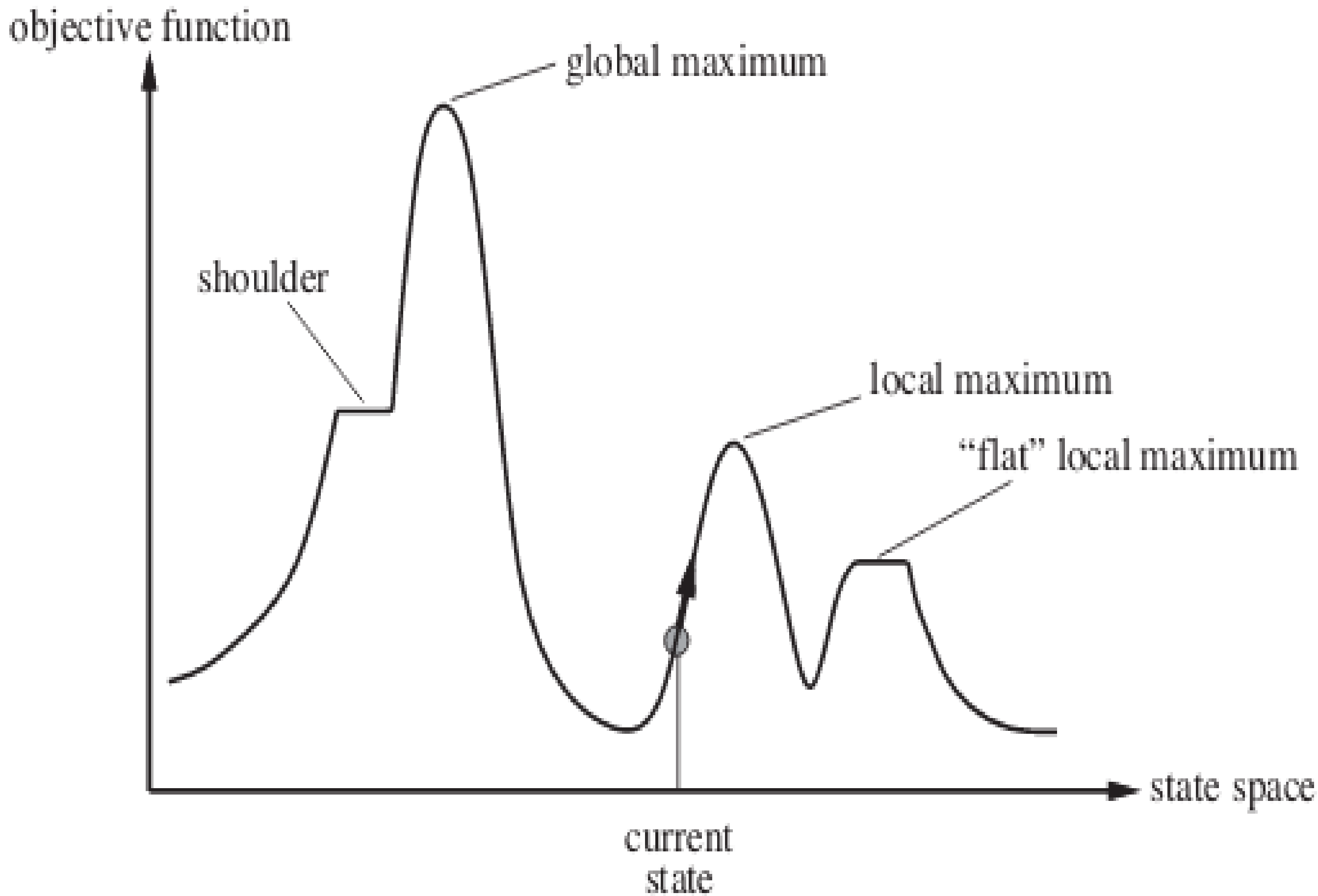
(ii). If it is a not a goal state but it is better than the current state, then make it the current state.

(iii). If it is not better than the current state, then continue is the loop.

State Space diagram for Hill Climbing

- State space diagram is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function(the function which we wish to maximize).
- **X-axis :** denotes the state space is states or configuration our algorithm may reach.
- **Y-axis :** denotes the values of objective function corresponding to a particular state.
- The best solution will be that state space where objective function has maximum value(global maximum).

State Space diagram for Hill Climbing



Different regions in the State Space Diagram:

- **Local maximum:** It is a state which is better than its neighboring state however there exists a state which is better than it(global maximum). This state is better because here the value of the objective function is higher than its neighbors.
- **Global maximum :** It is the best possible state in the state space diagram. This because at this state, objective function has highest value.
- **Plateau/flat local maximum :** It is a flat region of state space where neighboring states have the same value.

Different regions in the State Space Diagram:

- **Ridge :** It is region which is higher than its neighbors but itself has a slope. It is a special kind of local maximum.
- **Current state :** The region of state space diagram where we are currently present during the search.
- **Shoulder :** It is a plateau that has an uphill edge.

Types of Hill Climbing Search

- *Simple hill climbing*
- *Steepest ascent hill climbing*
- *Simulated annealing hill climbing*

Types of Hill Climbing Search

- In *simple hill climbing*, generate and evaluate states until you find one with a higher value, then immediately move on to it
- In *steepest ascent hill climbing*, generate all successor states, evaluate them, and then move to the highest value available (as long as it is greater than the current value)
  - in both of these, you can get stuck in a local maxima but not reach a global maxima
- Another idea is *simulated annealing*
  - the idea is that early in the search, we haven't invested much yet, so we can make some downhill moves

Dept. of CSE, RU

Best-First Search (BFS)

- **Best-first search depends on the use of a heuristic value to select most promising paths to the goal node.**
- The BFS algorithm retains all estimates computed for previously generated nodes and makes its selection based moves forward from the most promising of all the nodes generated so far.

Best-First Search (BFS)

- BFS algorithm selects the path which appears best at that moment.
- BFS algorithm is the combination of BFS(Breadth-First Search) and DFS (depth-First Search).
- Best –First Search(BFS) algorithm uses the heuristics value h(n) and search the lower cost path for the goal node.

Best-First Search (BFS) Algorithm

Best First Search Algorithm

1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the 'ordered' OPEN list
3. Repeat the next steps until GOAL node is reached
    I. If OPEN list is empty, then EXIT from the loop returning 'False'.

Best-First Search (BFS) Algorithm

II. Select the first/start node (say N) in the OPEN list and move it to the CLOSED list. Also capture the information of the parent node.

III. If N is a GOAL node, then move the node to the Closed list and exit the loop returning ‘True’. The solution can be found by backtracking the path.

IV. If N is not the GOAL node, expand node N to generate the ‘immediate’ next nodes linked to node N and add all those to the OPEN list.

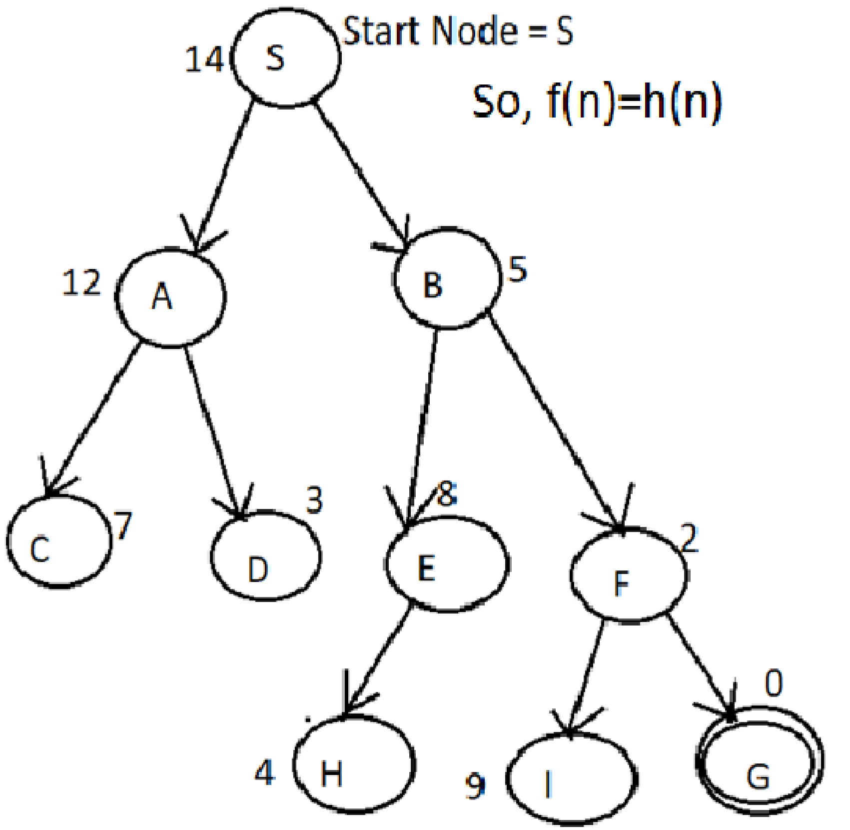
V. Reorder the nodes in the OPEN list in ascending order according to the heuristic evaluated function f(n).

Example for Best-First Search (BFS)

BFS Algorithm

State	h(n)
S	14
A	12
B	5
C	7
D	3
E	8
F	2
H	4
I	9
G	0

f(n)=[g(n)+h(n)]   Here, g(n)=0



=> initialization
OPEN List, CLOSE List
Open[S], Close[ ]
Open[B,A], Close[S]
Open[A], Close[S,B]
Open[F,E,A], Close[S,B]
Open[E,A], Close[S,B,F]
Open[G,I,E,A], Close[S,B,F]
Open[I,E,A], Close[S,B,F,G]
Goal is found, So we have gotten **Final Path.**
Final Path= S → B→ F→G

Example for Best-First Search (BFS)

Selection of lower cost path.	=> initialization
Step-1: Initialization; Start State = S; Goal State= G;	OPEN List, CLOSE List
Step-2:	Open[S], Close[ ]
S → A ; h(n)=h(A)=12 (Open)	Open[B,A], Close[S]
S → B ; h(n)= h(B)= 5, So, h(B) ≤ h(A)	Open[A], Close[S,B]
Step-3: S → B→ E ; h(E)= 8 (Open)	Open[F,E,A], Close[S,B]
S → B→ F ; h(F) = 2 , So, h(F) ≤ h(E)	Open[E,A], Close[S,B,F]
Step-4 S → B→ F→I ; h(I)= 9 (Open)	Open[G,I,E,A], Close[S,B,F]
S → B→ F→G ; h(G)= 0 , So, h(G) ≤ h(I)	Open[I,E,A], Close[S,B,F,G]
Final Path: S → B→ F→ G ;	Goal is found, So we have gotten Final Path.
	Final Path : S → B→ F→ G

Performance of BFS

- In worst case **time complexity** for **Best First Search** is O(n * Log n), where n is number of nodes. In worst case, It may has to visit all nodes before we reach goal.
- Best–First Search Space Complexity: $S(C) = O(b^m)$
- Where m is the max path length (so max number of ancestors) and b is the branching factor (number of children per ancestor).
- Performance of the algorithm depends on how well the cost or evaluation function is designed.

A* search Algorithm

- A* search algorithm finds the shortest path through the search space using the heuristic function.
- It uses heuristic value, h(n) and cost of reach the node n from the start state, g(n).
- This algorithm expands less tree and provides optimal result faster.
- A* search algorithm uses heuristic value of the node and as well as the cost to reach the node.

A* search Algorithm

- Hence we get,
- $F(n) = g(n) + h(n)$
- Where,
- $F(n)$ = Estimated cost of the path.
- $g(n)$ = Cost of reach node n from the start state.
- $h(n)$= The heuristic value of the n node.

A* search Algorithm

- Step-1: Consider OPEN List and CLOSED List. Place the starting node in OPEN list.
- Step-2: Check if the OPEN list is empty or not, if the list is empty then return, failure & stop.
- Step-3: Select the node from the OPEN list which has the lower cost value of the Evaluated Heuristic function [$F(n)=g(n) + h(n)$], if node n is goal node, then return success & stop, otherwise,
- Step-4: Expand node n & generate all of its successors & put n in the closed list:-

A* search Algorithm

i. For each successor node n check whether node n is already in the OPEN or CLOSED list.
ii. If not then compute evaluation function for node n and place into OPEN list.

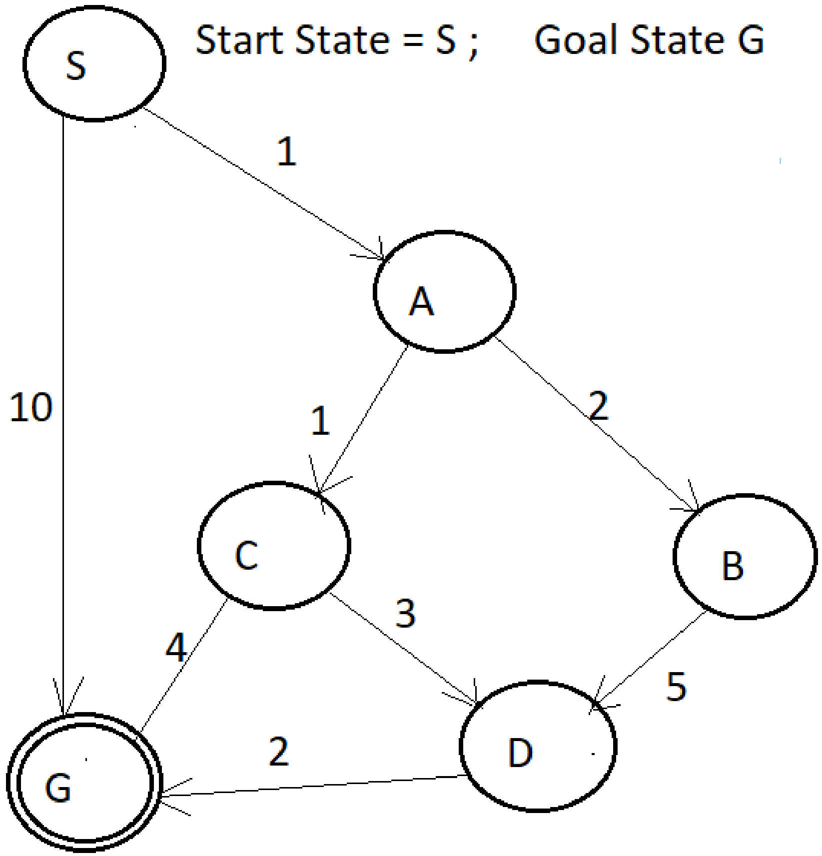
Step-5: Else if node n is already in OPEN & CLOSED, then it should be attached to the back pointer which rejects the lowest g (n) value.

Step-6: Return to Step-2.

Example for A* search Algorithm

State	h(n) Value
S	5
A	3
B	4
C	2
D	6
G	0

Start State = S ; Goal State G



Step-1:

S → A ; F(n) = g(n) + h(n)

= 1 + 3 = 4

S → G ; F(n) = 10 + 0 = 10 (Hold)

Step-2:

S → A → B ; F(n) = 3 + 4 = 7 (Hold)

S → A → C ; F(n) = 2 + 2 = 4

Step-3:

S → A → B → D ; F(n) = 5 + 6 = 11 (Hold)

S → A → C → **G** ; F(n) = 6 + 0 = 6

Final Path: S → A → C → **G** ; F(n) = 6

Example for A* search Algorithm

Step-1: Initialization; Start State = S; Goal State = G;	=> initialization OPEN List, CLOSE List Open[S], Close[ ] Open[A,G], Close[S] Open[G], Close[S,A] Open[C,B,G], Close[S,A,C] Open[G,D,B,G], Close[S,A,C] Open[D,B,G], Close[S,A,C,G] Lower cost goal is found. So, we have gotten **Final Path**. **Final Path= S → A→ C→ G**
Step-2: S → A ; F(n) = g(n) + h(n) = 1 + 3 = 4 S → G ; F(n) = 10 + 0 = 10 (Hold)	
Step-3: S → A→ B ; F(n) = 3 + 4 = 7 (Hold) S → A→ C ; F(n) = 2 + 2 = 4	
Step-4: S → A→ B→D ; F(n) = 5 + 6 = 11 (Hold) S → A→ C→**G** ; F(n) = 6 + 0 = 6	
Final Path: S → A→ C→ **G** ; F(n) = 6	

Reference Books

- 1. Dan W. Patterson, AI & Expert Systems.
- 2. Rich, Knight, Nair, AI, Third Edition.
- 3. https://www.geeksforgeeks.org/a-search-algorithm/

- **THE END**