Search Strategies

Module 2

Search Tree-Example

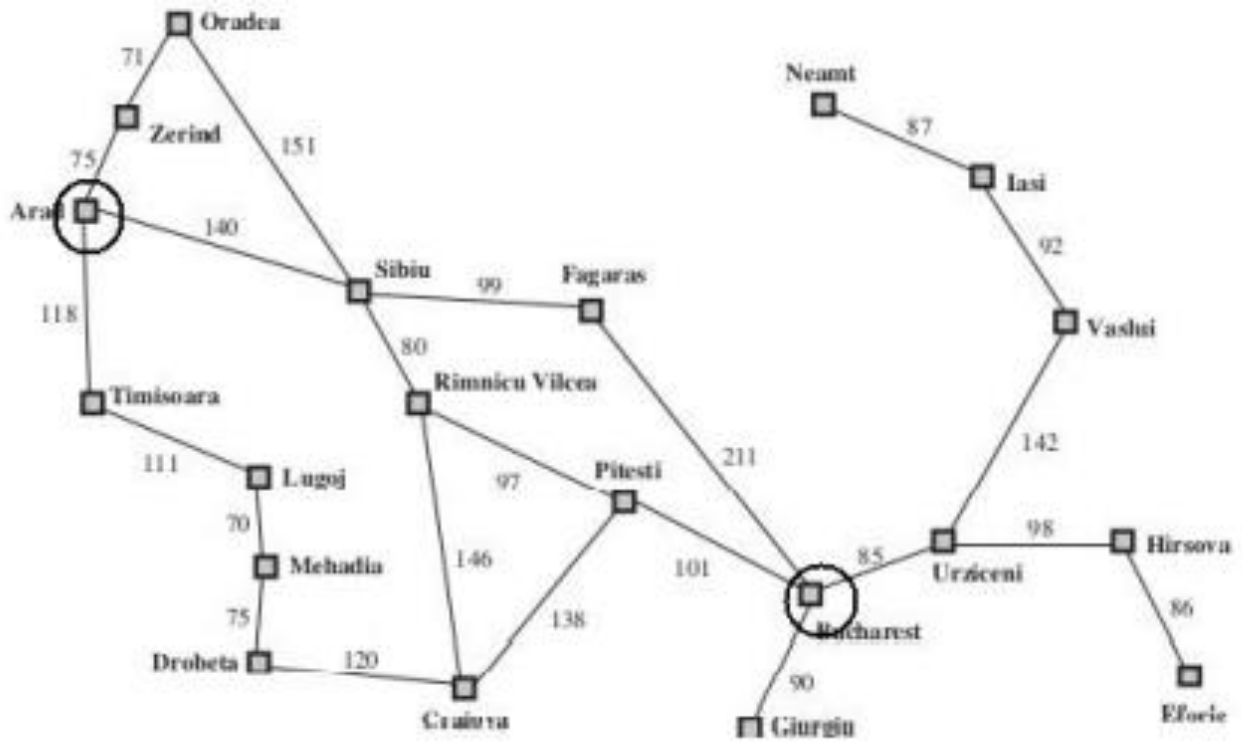


Figure 3.4: A road map of Romania

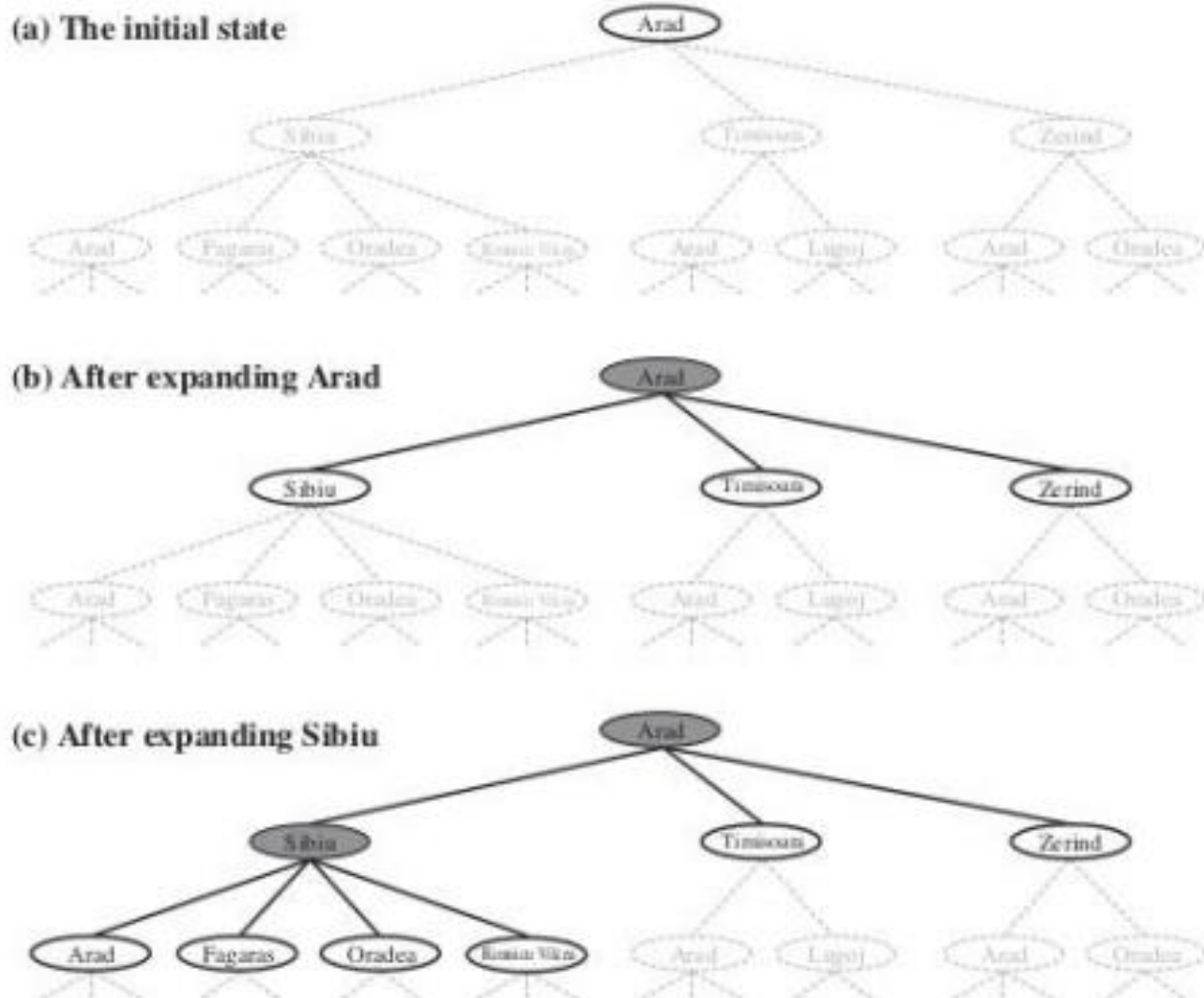
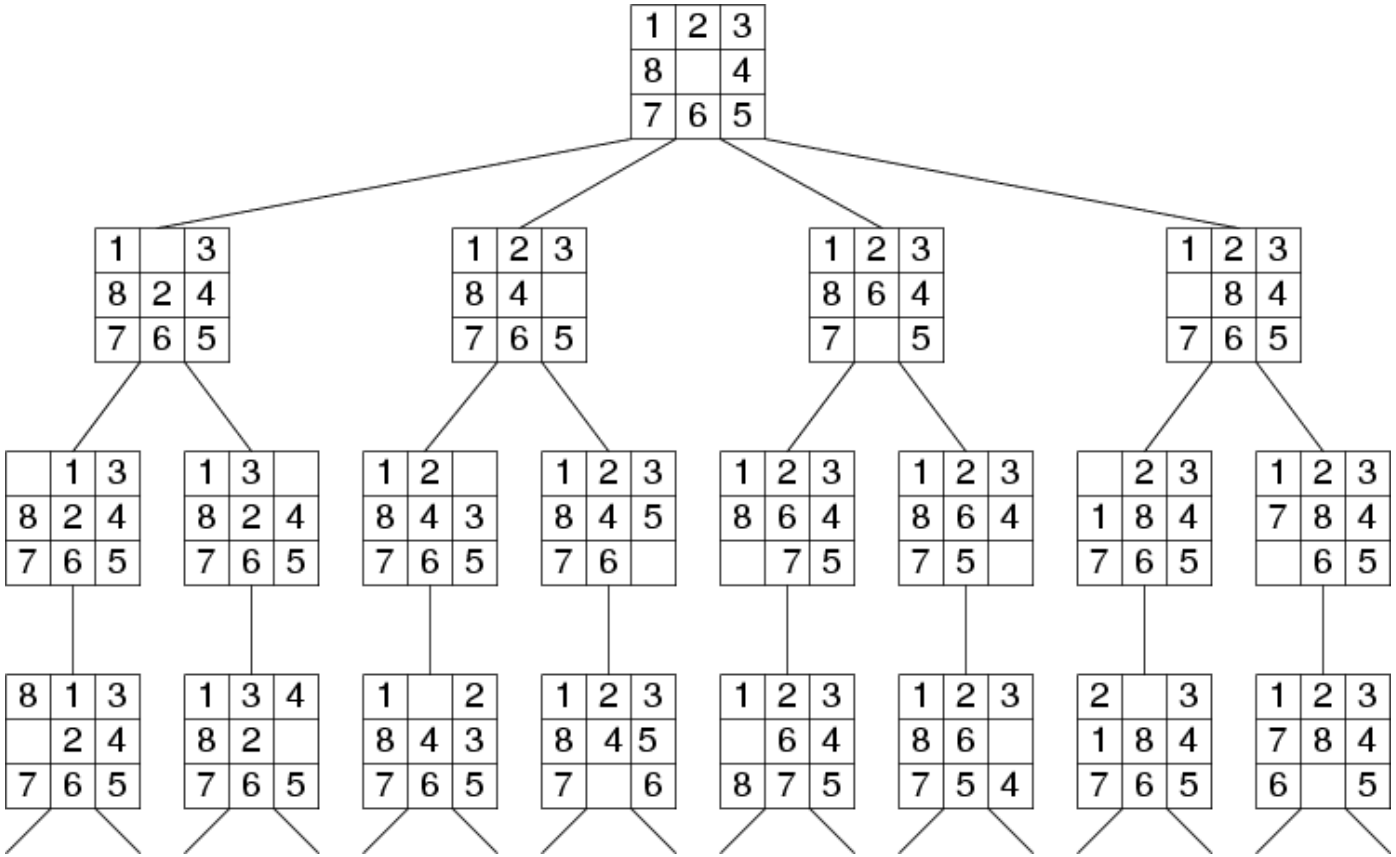


Figure 3.5: Partial search trees for finding a route from Arad to Bucharest

Search Tree for 8 puzzle



Search Strategy

- The first step is to test whether this is a goal state.
- Then we need to consider taking various actions.
- We do this by expanding the current state; that is, applying each legal action to the current state, thereby generating a new set of states.
- Now we must choose which of these possibilities to consider further.
- A search strategy specifies how to choose which state to expand next.

Blind search

- A blind search (also called an uninformed search) is a search that has no information about its domain.
- The only thing that a blind search can do is distinguish a non-goal state from a goal state.
- The blind search algorithms have no domain knowledge of the problem state.
- The only information available to blind search algorithms is the state, the successor function, the goal test and the path cost.
- Strategies that know whether one non-goal state is “more promising” than another are called informed search or heuristic search strategies.

Uninformed search strategies

- Breadth First Search
- Depth First Search
- Depth-Limited Search (DLS)
- Iterative Deepening Depth-First Search (IDDFS)
- Uniform Cost Search (UCS)

Breadth First Search

- It starts at the initial state and explores all the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.
- BFS uses a queue data structure to keep track of the nodes that are yet to be explored.
- It guarantees the shortest path from the initial state to the goal state if the path cost is a non-decreasing function of the depth of the node.

Breadth First Search

- The algorithm works by starting at the initial state and adding all its neighbours to a queue.
- It then dequeues the first node in the queue, adds neighbours to the end of the queue, and repeats the process until the goal state is found or the queue is empty.

BFS- Advantages

- **Completeness:**
  BFS is guaranteed to find the goal state if it exists in the search space, provided the branching factor is finite.
- **Optimal solution:**
  BFS is guaranteed to find the shortest path to the goal state, as it explores all nodes at the same depth before moving on to nodes at a deeper level.
- **Simplicity:**
  BFS is easy to understand and implement, making it a good baseline algorithm for more complex search algorithms.
- **No redundant paths:**
  BFS does not explore redundant paths because it explores all nodes at the same depth before moving on to deeper levels.

BFS-Disadvantages

- **Memory-intensive:**
  BFS can be memory-intensive for large search spaces because it stores all the nodes at each level in the queue.
- **Time-intensive:**
  BFS can be time-intensive for search spaces with a high branching factor because it needs to explore many nodes before finding the goal state.
- **Inefficient for deep search spaces:**
  BFS can be inefficient for search spaces with a deep depth because it needs to explore all nodes at each depth before moving on to the next level.

BFS

- **Time complexity-** The time complexity of BFS is proportional to the number of nodes in the search space, as BFS explores all nodes at each level before moving on to deeper levels. $O(b^d)$
- **Space Complexity -** The space complexity of BFS is proportional to the maximum number of nodes stored in the queue during the search. $O(b^d)$
- **Example Use**: Finding the shortest route in an unweighted graph, like a map without distances.

Depth First Search

The algorithm starts at a given node in the graph and explores as far as possible along each branch before backtracking.

DFS

- Mark the starting node as visited.
- Explore all adjacent nodes that have not been visited.
- For each unvisited adjacent node, repeat steps 1 and 2 recursively.
- Backtrack if all adjacent nodes have been visited or there are no unvisited nodes.
- DFS can be implemented using a stack data structure or recursion. The recursive implementation is simpler to understand but can cause a stack overflow if the graph or tree is too large.

DFS-Advantages

- **Memory efficiency:**
  DFS uses less memory than breadth-first search because it only needs to keep track of a single path at a time.
- **Finds a solution quickly:**
  If the solution to a problem is located deep in a tree, DFS can quickly reach it by exploring one path until it reaches the solution.
- **Easy to implement:**
  DFS is a simple algorithm to understand and implement, especially when using recursion.
- **Can be used for certain types of problems:**
  DFS is particularly useful for problems that involve searching for a path, such as maze-solving or finding the shortest path between two nodes in a graph.

DFS-Disadvantages

- **Can get stuck in infinite loops:**
  DFS can get stuck in an infinite loop if there are cycles in the graph or tree. This can be avoided by keeping track of visited nodes.
- **May not find the optimal solution:**
  DFS only sometimes finds the shortest path to a solution. It may find a suboptimal path before finding the shortest one.
- **Can take a long time:**
  In some cases, DFS may take a long time to find a solution, especially if the solution is located far from the starting node.

DFS

- **Time complexity-** $O(b^d)$
- **Space complexity-**O(d)
- **Example Use**: Useful in scenarios where memory is limited, like in certain types of games.

Sl No	Breadth-first	Depth-first
1	Requires more memory because all of the tree that has so far been generated must be stored.	Requires less memory because only the nodes in the current path are stored.
2	All parts of the search tree must be examined at level n before any node on level $n + 1$ can be examined.	May find a solution without examining much of the search space. It stops when one solution is found.
3	Will not get trapped exploring a blind alley.	May follow an unfruitful path for a long time, perhaps for ever.
4	If there is a solution, Guaranteed to find a solution if there exists one. If there are multiple solutions then a minimal solution (that is, a solution that takes a minimum number of steps) will be found.	May find a long path to a solution in one part of the tree when a shorter path exists in some other unexplored path of the tree.
5	Breadth-first search uses queue data structure.	Depth-first search uses stack data structure.
6	More suitable for searching vertices which are closer to the given source.	More suitable when there are solutions away from source.

Depth-Limited Search (DLS)

- It is a variant of DFS where the depth of the search is limited to a certain level.
- This is done to avoid infinite loops and memory overflow in cases where the search space is infinite or too large.
- DLS is particularly useful when the search space is large and there is limited memory available. By limiting the depth of the search tree, we can ensure that the search algorithm only explores a finite portion of the search space, which reduces the risk of running out of memory.

DLS

- **Initialize the search:**
  Start by setting the initial depth to 0 and initialize an empty stack to hold the nodes to be explored. Push the root node to the stack.
- **Explore the next node:**
  Pop the next node from the stack and increment the current depth.
- **Check if the node is a goal:**
  If the node is in a goal state, terminate the search and return the solution.
- **Check if the node is at the maximum depth:**
  If the node's depth is equal to the maximum depth allowed for the search, do not explore its children and remove it from the stack.
- **Expand the node:**
  If the node's depth is less than the maximum depth, generate its children and push them to the stack.
- **Repeat the process:**
  Go back to step 2 and repeat the process until either a goal state is found or all nodes within the maximum depth have been explored.

DLS-Advantages

- **Memory efficient:**
  DLS limits the depth of the search tree, which means that it requires less memory compared to other search algorithms like breadth-first search (BFS) or iterative deepening depth-first search (IDDFS).
- **Time efficient:**
  DLS can be faster than other search algorithms, especially if the solution is closer to the root node. Since DLS only explores a limited portion of the search space, it can quickly identify if a solution exists within that portion.
- **Avoids infinite loops:**
  Since DLS limits the depth of the search tree, it can avoid infinite loops that can occur in regular depth-first search (DFS) if there are cycles in the graph.

DLS-Disadvantages

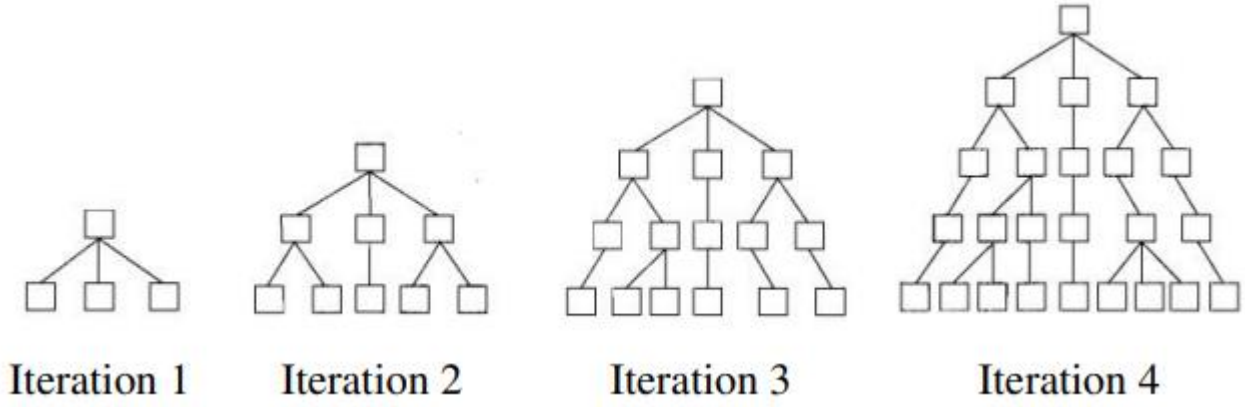
- **Suboptimal solutions:**
  DLS may not find the optimal solution, especially if the depth limit is set too low. If the optimal solution is deeper in the search tree, DLS may terminate prematurely without finding it.
- **Highly dependent on depth limit:**
  The performance of DLS is highly dependent on the depth limit. If the depth limit is set too low, it may not find the solution at all. On the other hand, if the depth limit is set too high, it may take too long to complete.

Iterative Deepening Depth-First Search (IDDFS)

- It is a combination of BFS and DFS.
- It repeatedly applies DFS with increasing depth limits until the goal node is found.
- IDDFS combines the benefits of BFS (guaranteed shortest path) and DFS (less memory consumption) by gradually increasing the depth limit.

Iterative Deepening Depth-First Search

- 1. Set SEARCH-DEPTH = 1.
- 2. Conduct a depth-first search to a depth of SEARCH-DEPTH. If a solution is found, then return it.
- 3. Otherwise, increment SEARCH-DEPTH by 1 and go to step 2



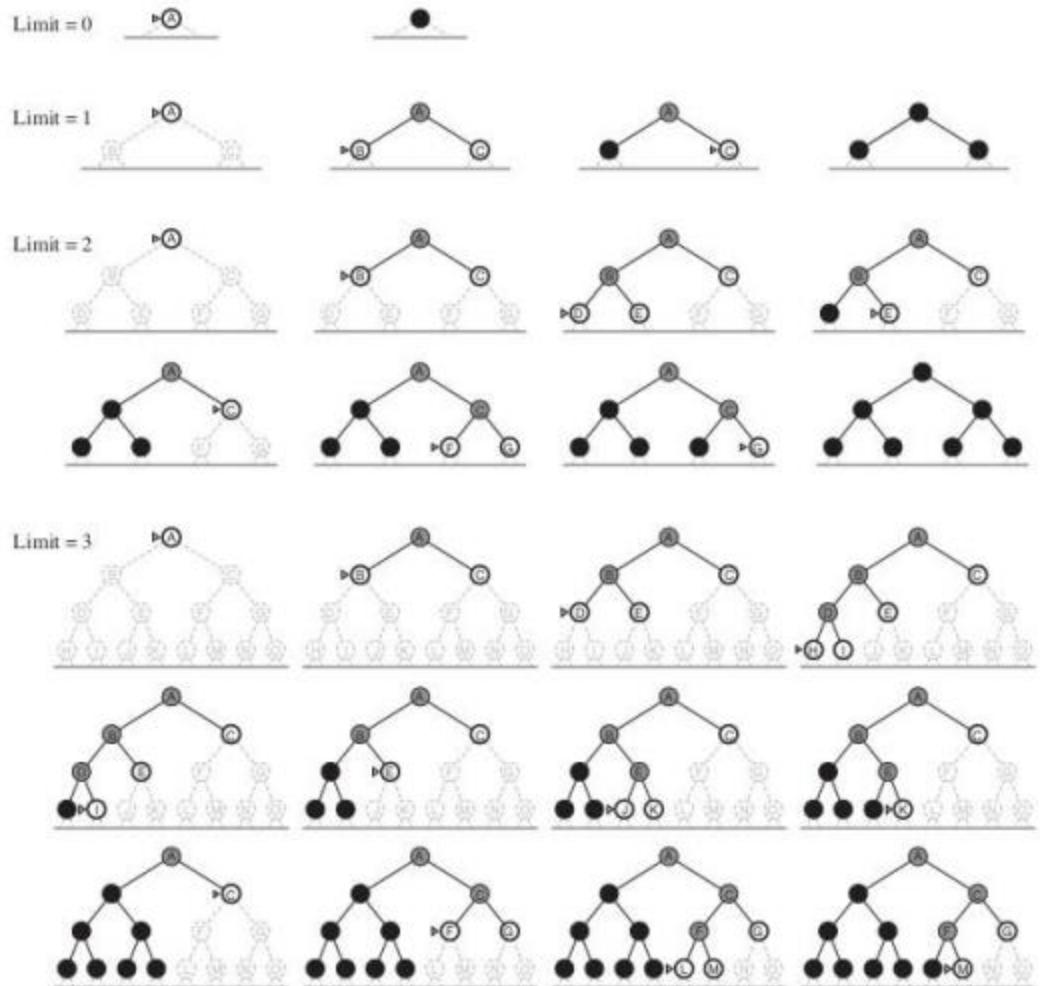


Figure 3.18: The order in which the nodes are explored in iterative deepening

IDDFS-Advantages

- **Optimal solution:**
  IDDFS guarantees to find the optimal solution, provided that the cost of each edge is identical.
- **Memory-efficient:**
  IDDFS uses less memory than breadth-first search (BFS) because it only stores the current path in memory, not the entire search tree.
- **Complete:**
  IDDFS is complete, meaning it will always find a solution if one exists.
- **Simple implementation:**
  IDDFS is easy to implement since it is based on the standard depth-first search algorithm.

IDDFS-Disadvantages

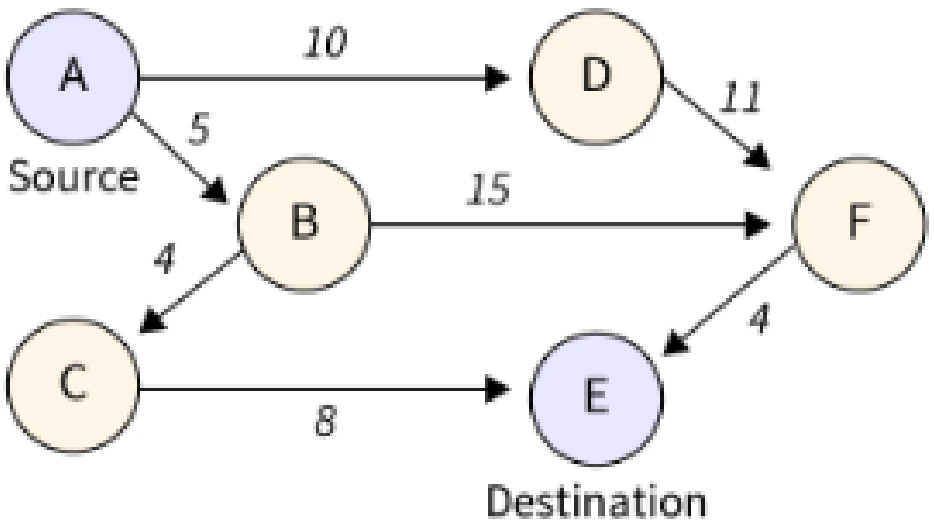
- **Inefficient for branching factors that are very high:**
  If the branching factor is very high, IDDFS can be inefficient since it may repeatedly explore the same nodes at different depths.
- **Redundant work:**
  IDDFS may explore the same nodes multiple times, leading to redundant work.
- **Time complexity can be high:**
  The time complexity of IDDFS can be high if the depth limit needs to be increased multiple times before finding the goal state.

Uniform-cost search (UCS)

- UCS is like BFS, but it prioritizes the lowest cost path. It uses a priority queue to explore paths based on cost.
- UCS guarantees finding the optimal path to the goal node when the path cost is non-negative.
- It's useful in situations where paths have different costs associated with them.
- UCS expands the least costly node first, ensuring that when it reaches a goal node, it has found the least cost path to that node.

Uniform-cost search (UCS)

- **Completeness**: UCS will find a solution if one exists.
- **Optimal for Costly Paths**: It finds the least costly path when costs vary.
- **High Memory Usage**: Like BFS, it can consume a lot of memory.


A
10
D
11
5
Source
B
15
F
4
4
C
E
8
Destination

Scenario	Best Algorithm
Finding the shortest path in an unweighted graph	**BFS**
Searching deep trees with limited memory	**DFS**
Large state spaces with memory constraints	**IDS**
Finding the lowest-cost path	**UCS**

Algorithm	Time Complexity	Space Complexity	Completeness	Optimality
Breadth-First Search (BFS)	$O(b^d)$	$O(b^d)$	✅ Yes	✅ Yes (if all step costs are equal)
Depth-First Search (DFS)	$O(b^d)$	$O(d)$	✅ Yes (if infinite depth is avoided)	❌ No
Depth-Limited Search (DLS)	$O(b^l)$	$O(l)$	❌ No (if cutoff < depth)	❌ No
Iterative Deepening Search (IDS)	$O(b^d)$	$O(d)$	✅ Yes	✅ Yes
Uniform Cost Search (UCS)	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	✅ Yes	✅ Yes

- **b** = branching factor (number of children per node).
- **d** = depth of the shallowest goal node.
- **C^*** = cost of the optimal solution path.
- **ϵ** = smallest edge cost in the graph.
- **Time Complexity** – Number of nodes expanded before finding a solution.
- **Space Complexity** – Number of nodes stored in memory.

Strategies for state space search

- By whether using domain-specific information:
  - Blind search methods(Uninformed)
  - Heuristic search (Informed)
- By the search directions:
  - data-driven search (forward chaining)
  - goal-driven search (backward chaining)
  - bidirectional search

Heuristic search strategies

- Informed search strategies, also known as heuristic search strategies,
- It use problem-specific knowledge to find solutions more quickly than uninformed (blind) search strategies.
- **Uses Heuristics** – It relies on a heuristic function h(n) to estimate the cost of reaching the goal from a given node.
- **More Efficient** – It reduces the number of nodes explored, making the search process faster than uninformed methods.
- **Guided Approach** – Instead of blindly exploring all paths, it focuses on the most promising paths based on heuristic information

Heuristic function

- A heuristic function, also called simply a heuristic, is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow

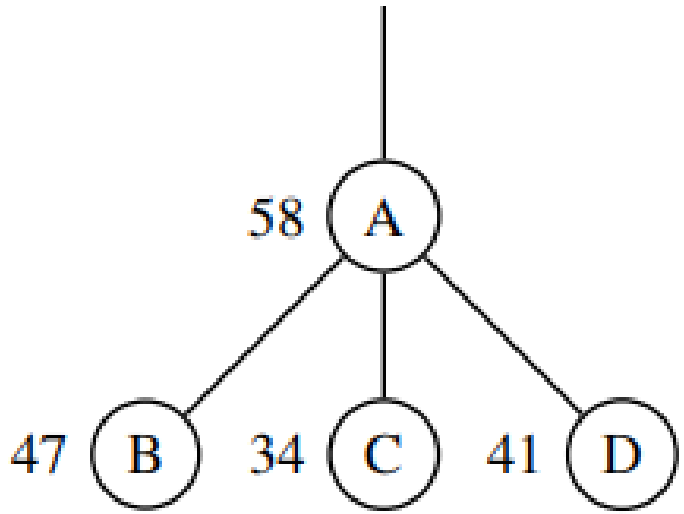


Figure 4.1: Values of a heuristic function

- The heuristic function for a search problem is a function h(n) defined as follows:
- h(n) = estimated cost of the cheapest path from the state at node n to a goal state.

- Unless otherwise specified, it will be assumed that a heuristic function h(n) has the following properties:
- 1. $h(n) \geq 0$ for all nodes n.
- 2. If n is the goal node, then h(n) = 0.

Admissible heuristic function

- An **admissible heuristic** is a heuristic function that **never overestimates** the actual cost of reaching the goal.
- A heuristic function h(n) is admissible if **$h(n) \leq h^*(n)$ for all n**
  - Where h(n) is the estimated cost from node n to the goal
  - and $h^*(n)$ is the true cost from node n to the goal.

Remarks

- 1. There are no limitations on the function h(n). Any function of our choice is acceptable. As an extreme case, the function h(n) defined by
- h(n) = c for all n where c is some constant can also be taken as a heuristic function.
- The heuristic function defined by h(n) = 0 for all n is called the trivial heuristic function

- 2. The heuristic function for a search problem depends on the problem in the sense that the definition of the function would make use of information that are specific to the circumstances of the problem. Because of this, there is no one method for constructing heuristic functions that are applicable to all problems.

- 3. However, there is a standard way to construct a heuristic function: It is to find a solution to a simpler problem, which is one with fewer constraints. A problem with fewer constraints is often easier to solve (and sometimes trivial to solve). An optimal solution to the simpler problem cannot have a higher cost than an optimal solution to the full problem because any solution to the full problem is a solution to the simpler problem

Key considerations in using heuristics

- **Admissibility** – The heuristic should not overestimate the actual minimum cost to reach the goal. Overestimates could mislead the search. Admissible heuristics guarantee optimality.
- **Consistency** – The heuristic value should decrease monotonically as the search expands nodes getting closer to the goal. Inconsistent heuristics can cause issues like looping.
- **Efficiency** – Computing heuristic values should not be too expensive. Simple heuristics tend to scale better than complex ones.
- **Domain-specific** – Heuristics designed for a problem are more informative than generic ones. But problem-specific heuristics lack reusability across applications.
- **Accuracy** – The heuristic should closely approximate actual goal distance as much as possible. More accurate heuristics yield more effective guidance

Examples of admissible heuristics

- **1. Straight-Line Distance (Euclidean Distance)**
- Used in **pathfinding problems** like GPS navigation.
- **Example:** If you're trying to drive from **City A** to **City B**, the straight-line distance between them is an admissible heuristic because the actual driving distance can only be equal to or longer than this.

Example- h(n) in path search in map

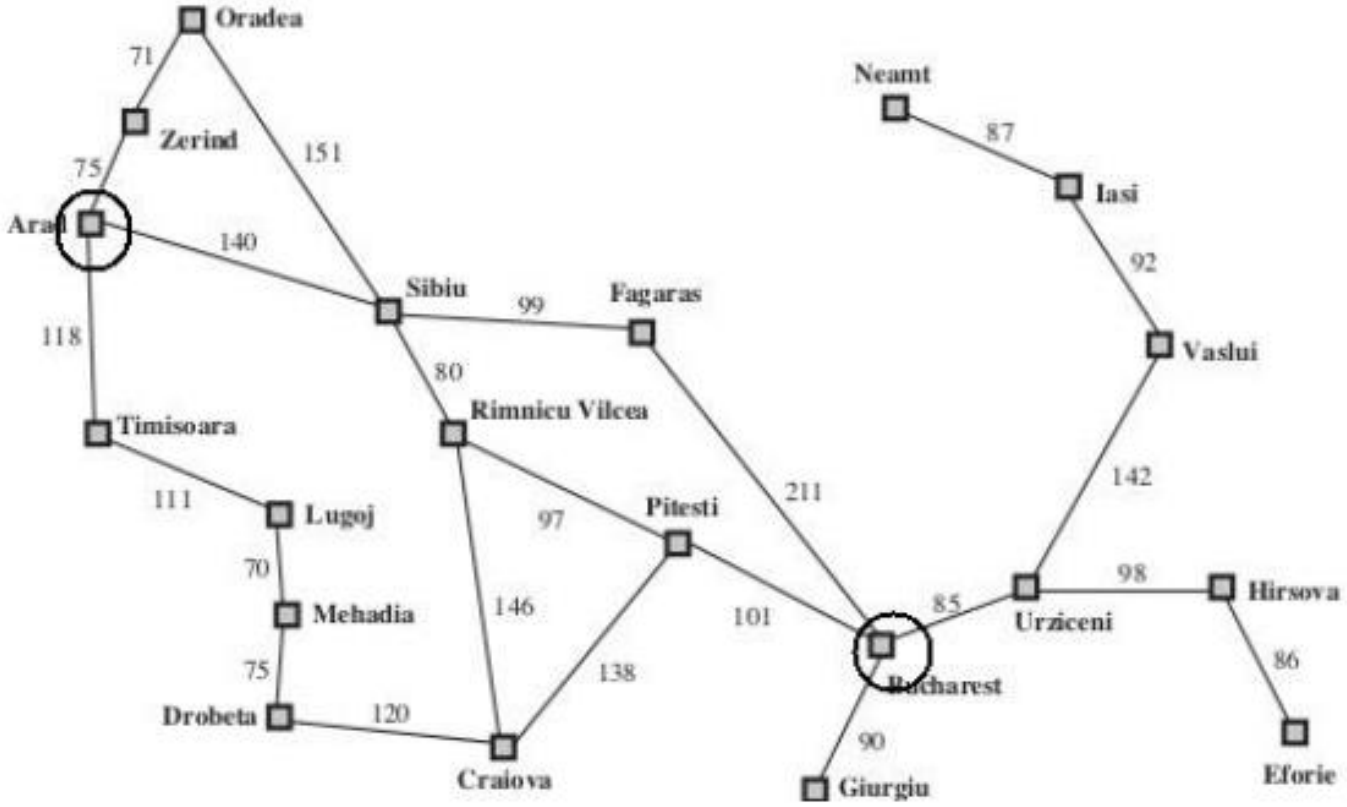


Figure 3.4: A road map of Romania

Example- h(n) in path search in map

- Consider the problem of finding the shortest path from Arid to Bucharest in Romania
- h(n) = The straight line distance (air distance) from n to Bucharest
- shows the values of h(n). Note that $h(n) \geq 0$ for all n and h(Bucharest) = 0. Now $h*(n)$ is the the length of the shortest road-path from n to Bucharest. Since, the air distance never exceeds the road distance we have $h(n) \leq h*(n)$ for all n. Thus, h(n) as defined above is an admissible heuristic function for the problem

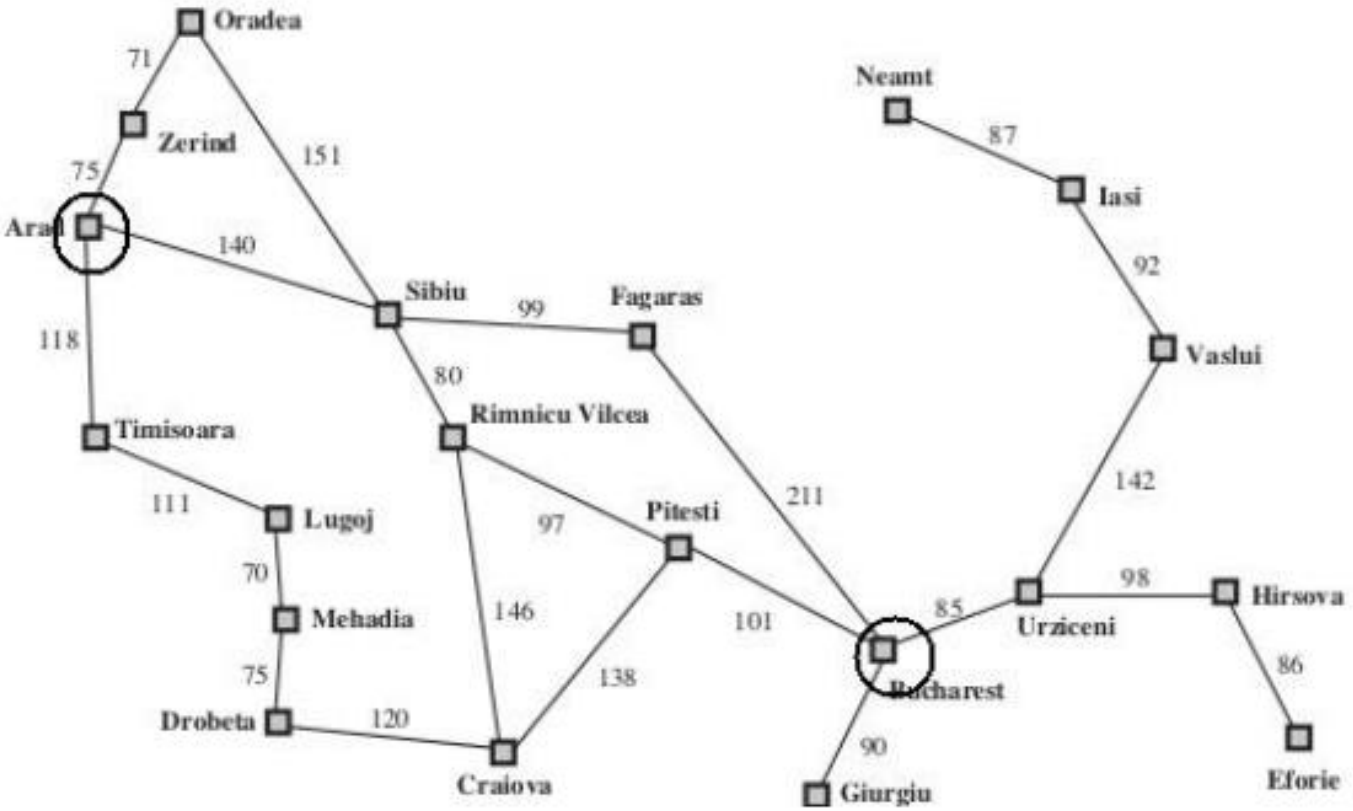


Figure 3.4: A road map of Romania

City (n)	$h(n)$
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244

City (n)	$h(n)$
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Table 4.1: Heuristic function for Romania travel problem

Example- h(n) in the 8-puzzle problem

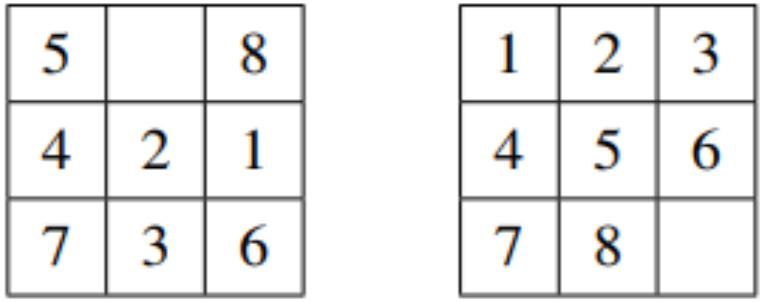
- **Restrictions**
- 1. A tile can be moved only to a nearby vacant slot.
- 2. Tiles can only be moved in the horizontal or vertical direction.
- 3. One tile cannot slide over on another tile

- We define two different heuristic functions h1(n) and h2(n) for the 8-puzzle problem.
- h1(n) is obtained by solving the problem without all the three restrictions given above.
- h2(n) is obtained by solving the problem obtained by relaxing first restriction to the restriction that “a tile may be moved to any nearby slot and deleting the third restriction”

Examples of admissible heuristics

2. Number of Misplaced Tiles (8-Puzzle Problem)

- In an **8-puzzle**, where tiles must be moved into a correct order, counting the number of misplaced tiles serves as an admissible heuristic
- Hamming distance heuristic

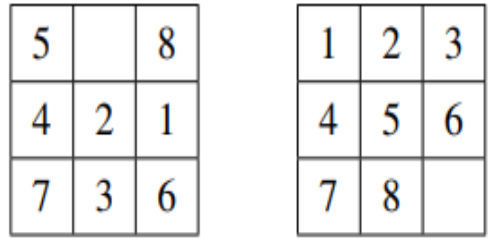


(a) Given state (n) (b) Goal state

h1(n) = Number of tiles out of position.

- h1(n)=6

Examples of admissible heuristics



(a) Given state (n) (b) Goal state

3. Manhattan Distance (for Grid-based Problems)

- Used in **tile-based games, like the 8-puzzle**.
- Calculates the sum of horizontal and vertical movements needed to reach the goal.

- h2(n) = Sum of the Manhattan distances of every numbered tile to its goal position
- h2(n) = Manhattan distance of 1 from goal position+ Manhattan distance of 2 from goal position+ · · · + Manhattan distance of 8 from goal position
- h2(n)= 3 + 1 + 3 + 0 + 2 + 1 + 0 + 3 = 13

Types of Informed Search Strategies

- Greedy Best-First Search (GBFS)
- *A* Search Algorithm*
- Hill Climbing
- Simulated Annealing

Best First Search

- An informed search uses an evaluation function f(n) to decide which among the various available nodes is the most promising (or “best”) before traversing to that node. An algorithm that implements this approach is known as a best first search algorithm.
- It should be emphasised that, in general, the evaluation function is not the same as the heuristic function.
- There are different best first search algorithms depending on what evaluation function is used to determine the most promising node.

- One commonly used evaluation function is the heuristic function. The corresponding algorithm is sometimes called the greedy best first search algorithm.
- Another variant known as the A* best search algorithm uses the sum of the heuristic function and the function that specifies the cost of traversing the path from the start node to a particular node as the evaluation function

Greedy Best First Search

- Uses a heuristic function h(n) to decide which node to expand.
- Ignores the cost to reach the current node g(n), focusing only on getting closer to the goal.
- Uses a priority queue to always expand the node with the smallest heuristic value.
- Can be faster than other algorithms but does not guarantee an optimal solution.

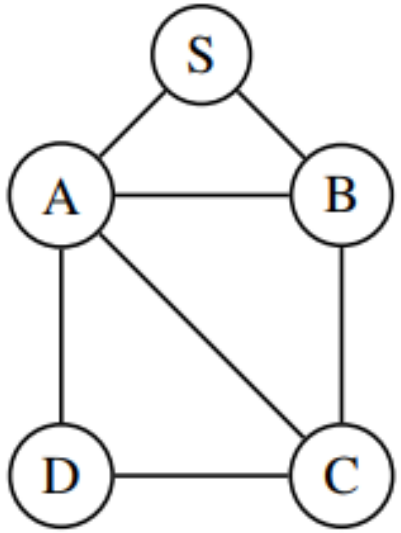
Greedy Best First Search

1. Initialize an empty priority queue.
2. Insert the start node into the priority queue.
3. Repeat the following until the goal is found or the queue is empty:
    1. Remove the node with the smallest heuristic value h(n).
    2. If this node is the goal, return the solution.
    3. Otherwise, expand the node and insert its successors into the priority queue.
4. If the queue is empty, return failure (no path found).

Example 1

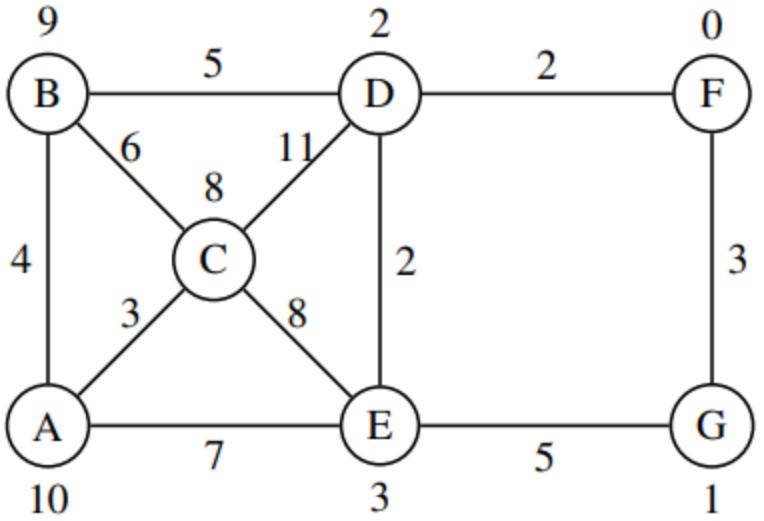
- Use the greedy best first search algorithm to find a path from S to C in the graph
- Use the following heuristic function

Node n	S	A	B	C	D
$h(n)$	6	2	3	0	2



Example 2

- Using the greedy best first search algorithm, find an optimal path from A to F in the search graph shown in Figure. In the figure, the numbers written alongside the nodes are the values of the heuristic function and the numbers written alongside the edges are the costs associated with the edges.



Example 3

- Using the greedy best first search algorithm find the optimal path from Arid to Bucharest in Figure using the heuristic function defined by Table

Greedy Best First Search-Advantages

- **Fast Execution** – Expands fewer nodes than uninformed searches.
- **Good for Simple Problems** – Works well when heuristics are well-designed.
- **Less Memory Usage** – Stores fewer nodes compared to A*.

Greedy Best First Search-Disadvantages

- **Not Optimal** – May find a suboptimal solution.
- **May Get Stuck in Local Minima** – Ignores g(n), so it might choose a misleading path.
- **Incomplete in Some Cases** – Can enter an infinite loop in graphs with cycles.

A∗ Algorithm

- A* is an **informed search algorithm** used to find the shortest path in a graph.
- It is widely used in **pathfinding and graph traversal problems**, such as GPS navigation, AI in games, and robotics.
- It efficiently finds the **shortest path** by using both actual cost and heuristic estimates.
- A∗ algorithm uses the evaluation function: **f(n) = g(n) + h(n).**
- f(n)= **Total estimated cost** of the node.
- g(n)= **Actual cost** from the start node to the current node.
- h(n)= **Heuristic estimate** of the cost from the current node to the goal.

A* algorithm

1.Initialize:

- Create an **open list** (priority queue) and add the **start node**.
- Create a **closed list** (to track visited nodes).

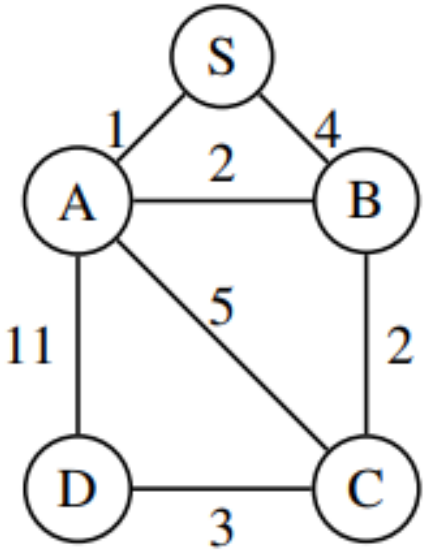
2.Loop until the goal is found or the open list is empty:

- Remove the node with the **lowest f(n)** from the open list.
- If it is the **goal**, return the path.
- Move the node to the **closed list**.
- Expand the node and calculate f(n) for its neighbors.
  - If a neighbor is not in the open/closed list, add it to the open list.
  - If a better path is found for a neighbor, update its cost.

3.If the open list is empty, return failure (no path found).

Example 1

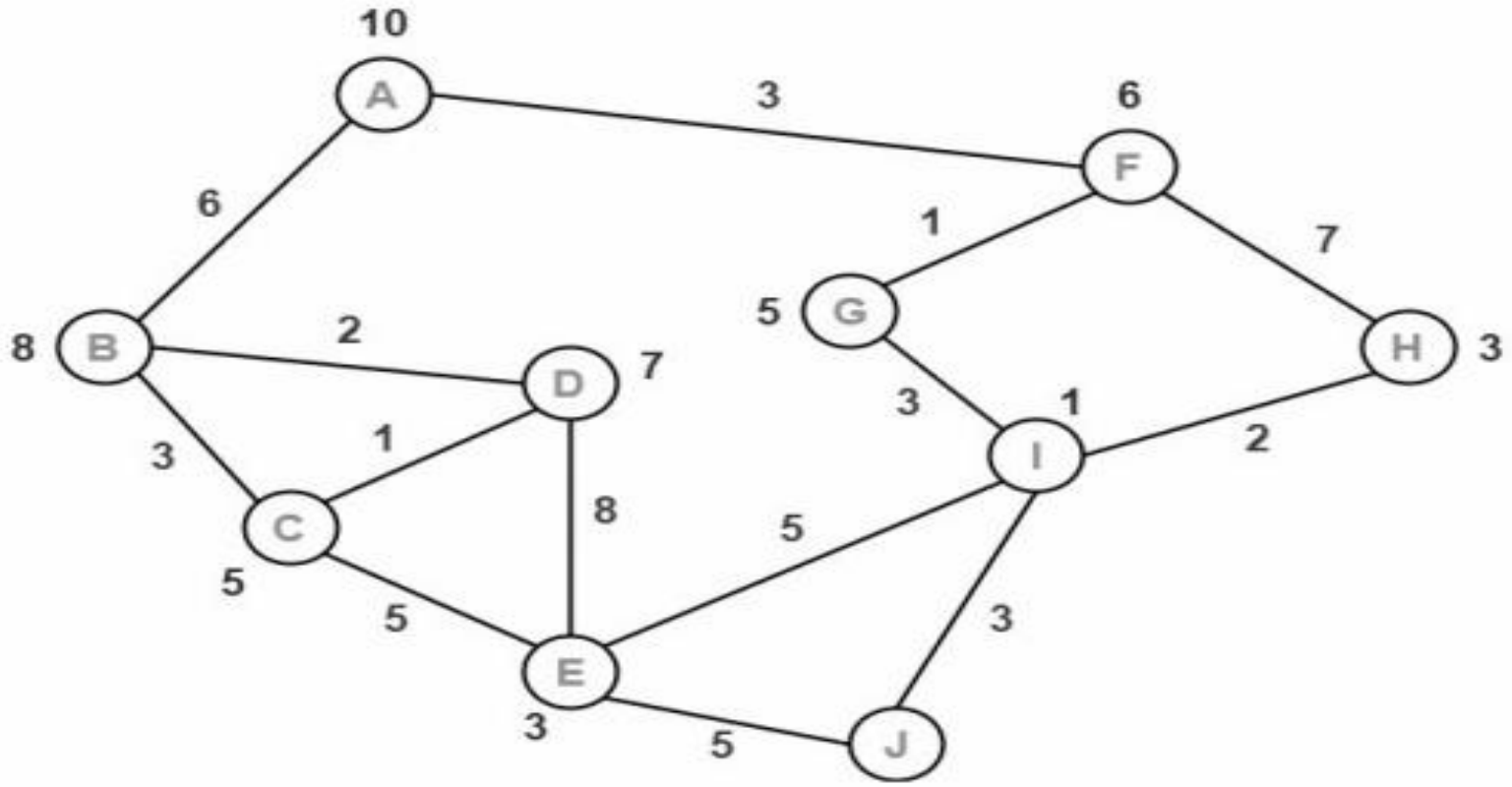
Use A* algorithm to find a path from S to C in the graph



Node n	S	A	B	C	D
$h(n)$	6	2	3	0	2

Example 2

- Find the most cost-effective path to reach from start state A to final state J using A* Algorithm



A*- Advantages

- **Guaranteed Optimality** – Always finds the shortest path if h(n) is admissible.
- **Efficient** – Faster than Dijkstra's algorithm in many cases.
- **Widely Used** – Used in games, robotics, and navigation systems.

A*-Disadvantages

- **High Memory Usage** – Stores many nodes in the open and closed lists.
- **Performance Depends on Heuristic** – A poor heuristic can slow down A*.

Sl. No.	Greedy best first	A^* best first
1	The greedy best first is not complete, that is, the algorithm may not find the goal always.	If the heuristic function is admissible, the A^* best first is complete.
2	In general, the greedy best first may not find the optimal path from the initial to the goal state.	If the heuristic function is admissible, the A^* best first is always yields the optimal path from the initial state to goal state.
3	In general, greedy best first uses less memory than A^* best first.	In general, A* best first uses more memory than greedy best first.

Hill Climbing

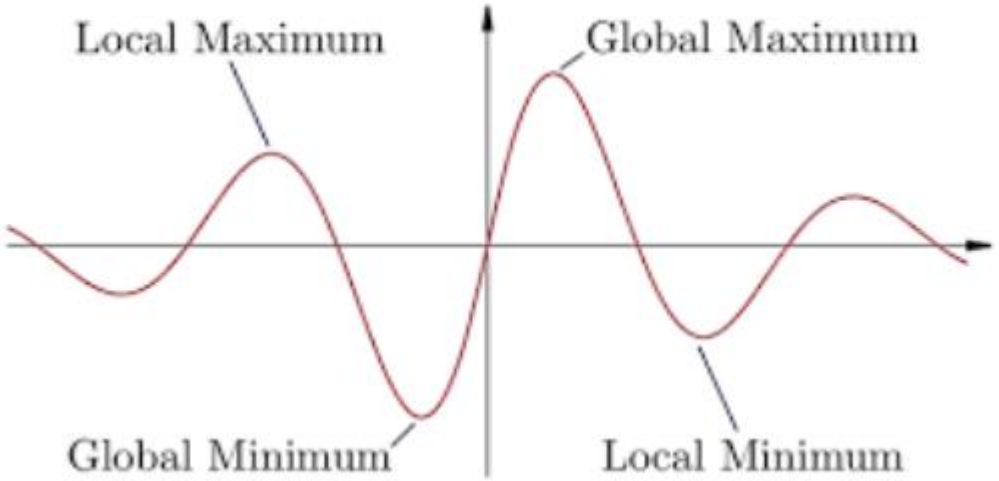
- Hill climbing is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an incremental change to the solution.
- If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found.
- The solution obtained by this method may not be the global optimum; it will only be a local optimum.
- In this sense, it is a local search method.
- Hill Climbing is a **simple and efficient** algorithm for **local search and optimization**.

Key Features of Hill Climbing

- **Uses a Heuristic Function** – Evaluates each state and moves towards a better one.
- **Greedy Approach** – Always chooses the best neighboring state.
- **No Backtracking** – It does not revisit previous states.
- **Efficient for Local Search** – Suitable for problems like route optimization, scheduling, and neural network tuning.

Hill Climbing

- The difference between a local and a global maxima is illustrated in Figure.
- In the figure, the x-axis denotes the nodes in the state space and the y-axis denotes the values of the objective function corresponding to particular states.



Hill Climbing

- When we perform hill-climbing, we are short-sighted. We cannot really see far. We make local best decisions with the hope of finding a global best solution

Types of Hill climbing

1. **Simple Hill Climbing**

- Evaluates one successor at a time.
- Moves to the first better state found.
- Fast but may miss the best solution (gets stuck in local maxima).

2. **Steepest-Ascent Hill Climbing**

- Evaluates all possible successors and moves to the best one.
- More effective but computationally expensive.

Simple Hill Climbing

1. **Start with an initial solution** (randomly chosen or given)
2. **Evaluate the heuristic value** of the current state.
3. **Generate a single neighbor**.
4. **Compare the heuristic value** of the neighbor with the current state:
    1. If the neighbor is **better**, move to it.
    2. Otherwise, **stop** (local maximum reached).
    3. **Repeat until no improvement is possible**.

Example

- Given the 8-puzzle, use the hill-climbing algorithm with the Manhattan distance heuristic to find a path to the goal state.

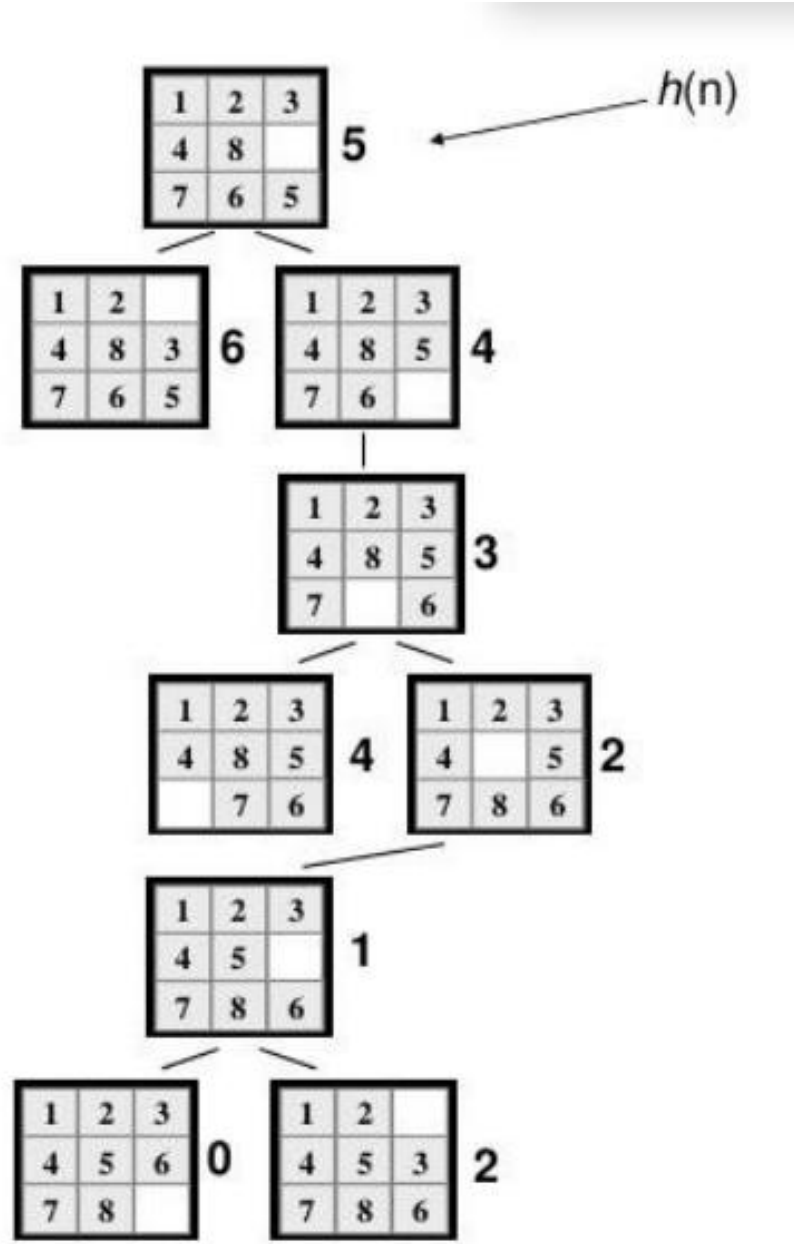
1	2	3
4	8	
7	6	5

Initial state

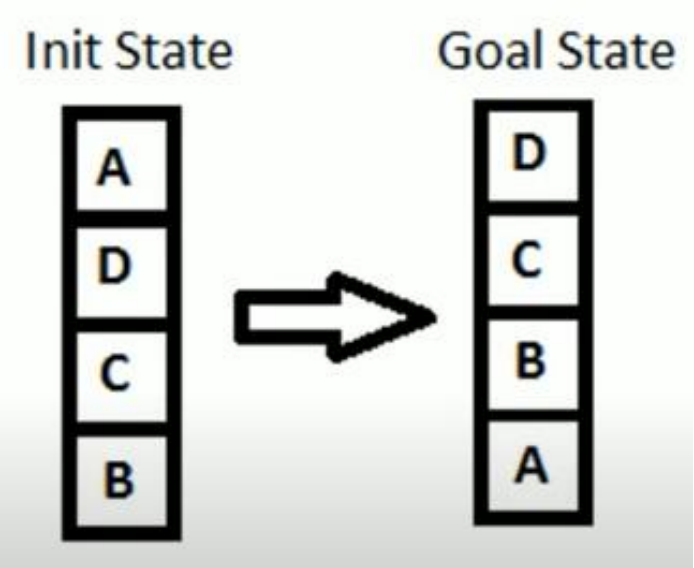
1	2	3
4	5	6
7	8	

Goal state

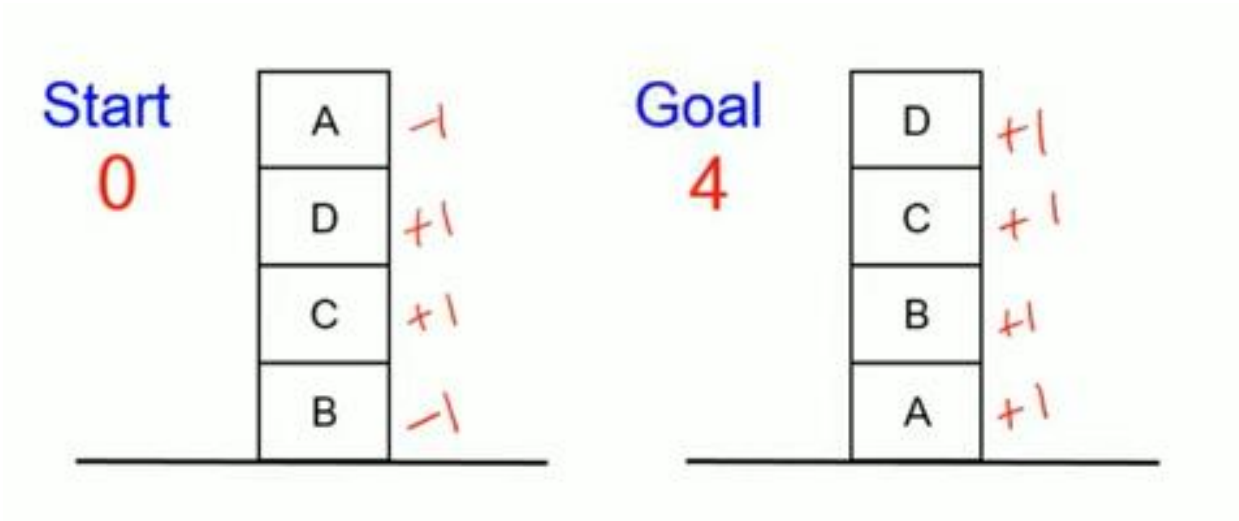
- h(Initial state) = 2 + 2 + 1 = 5

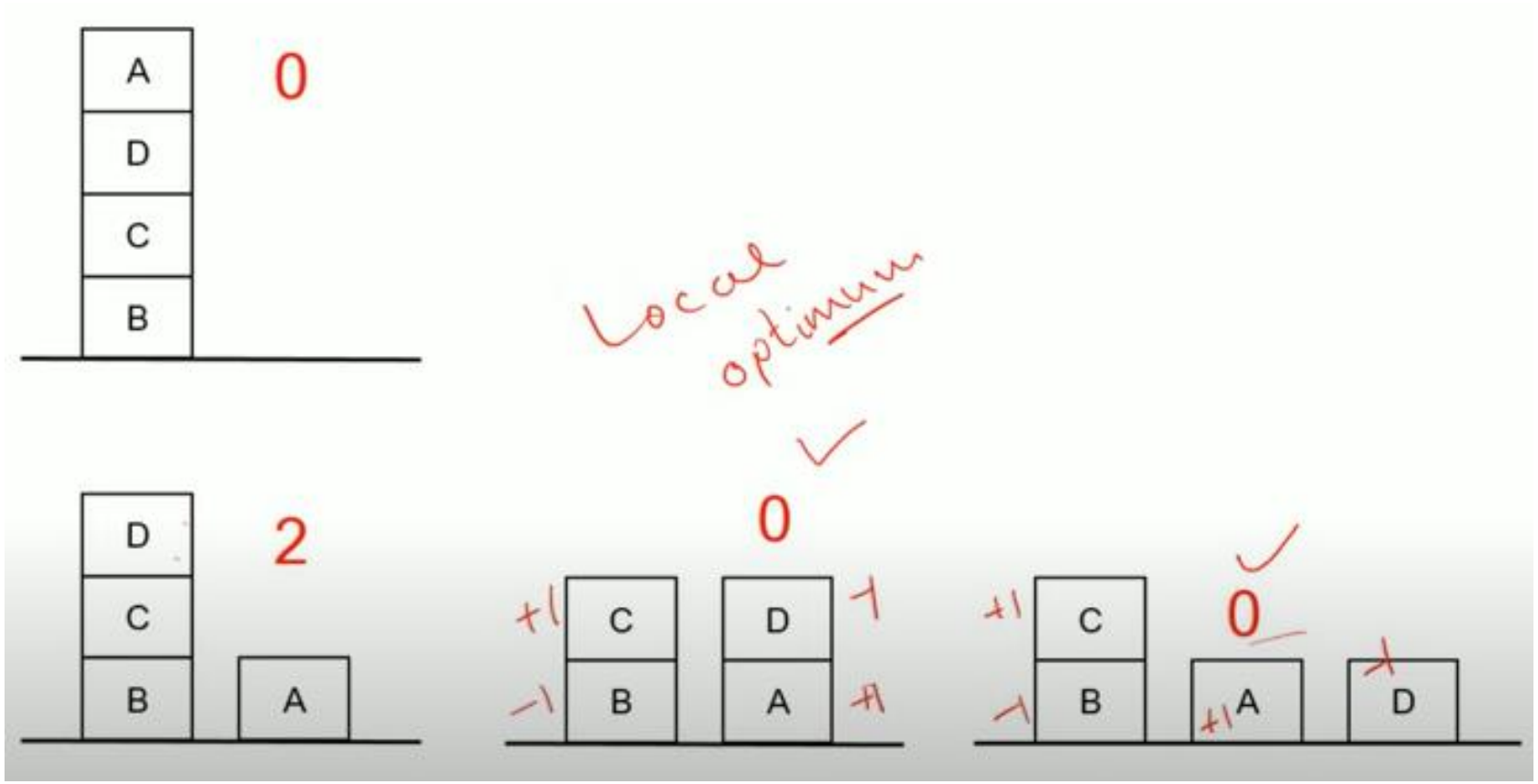

h(n)
1 2 3
4 8
7 6 5
5
1 2
4 8 3
7 6 5
6
1 2 3
4 8 5
7 6
4
1 2 3
4 8 5
7 6
3
1 2 3
4 8 5
7 6
4
1 2 3
4 5
7 8 6
2
1 2 3
4 5
7 8 6
1
1 2 3
4 5 6
7 8
0
1 2
4 5 3
7 8 6
2

Example Blocksworld problem



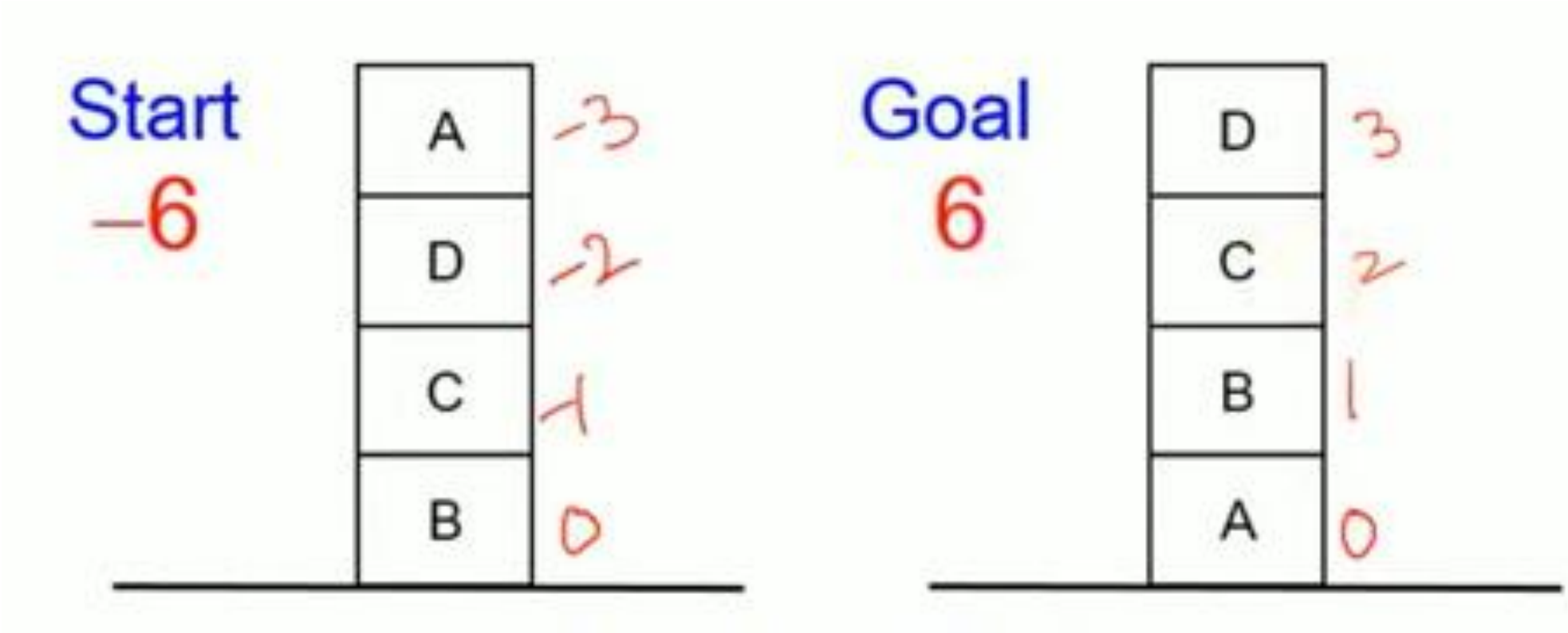
- h(n) = Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing

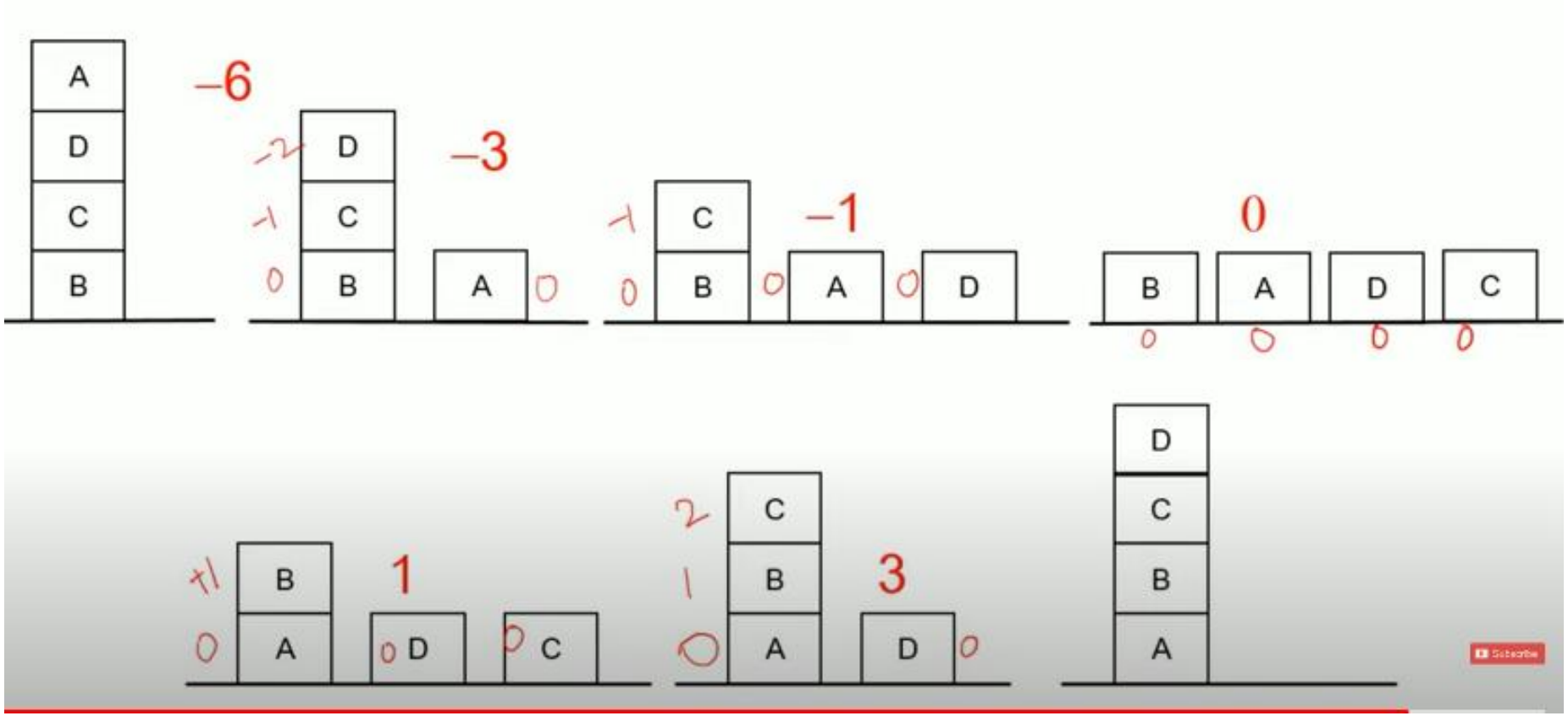

Start
0
A
D
C
B
-1
+1
+1
-1
Goal
4
D
C
B
A
+1
+1
+1
+1


A
D
C
B
0
D
C
B
A
2
Local optimum
0
+1
C
D
-1
B
A
0
+1
C
-1
B
+1
A
D

Blocksworld- another heuristic function

- For each block that has the correct support structure: +1 to every block in the support structure
- For each block that has the wrong support structure: -1 to every block in the support structure


Start
−6
A
D
C
B
-3
-2
-1
0
Goal
6
D
C
B
A
3
2
1
0


A
D
C
B
-6
D
C
B
A
-3
C
B
A
D
-1
B
A
D
C
0
B
A
D
C
B
1
D
C
C
B
A
3
D
D
C
B
A
Subscribe

Simple Hill Climbing

- Once we find an operation which produces a better value than the value in current state we do proceed to that state without considering whether there is any other move which produces a state having a still better value

Steepest-Ascent Hill Climbing

- In steepest-ascent hill climbing, we consider all the moves from the current state and selects the best as the next state. In the basic hill climbing, the first state that is better than the current state is selected.

Steepest-Ascent Hill Climbing

1. **Start with an initial solution** (randomly chosen or given).
2. **Evaluate the heuristic value** of the current state.
3. **Generate all possible neighbors** of the current state.
4. **Select the best neighbor** (the one with the highest heuristic value).
5. **Move to the best neighbor** if it improves the heuristic value.
6. **Repeat until** no better neighbor exists (**local maximum reached**).

Advantages of Hill Climbing

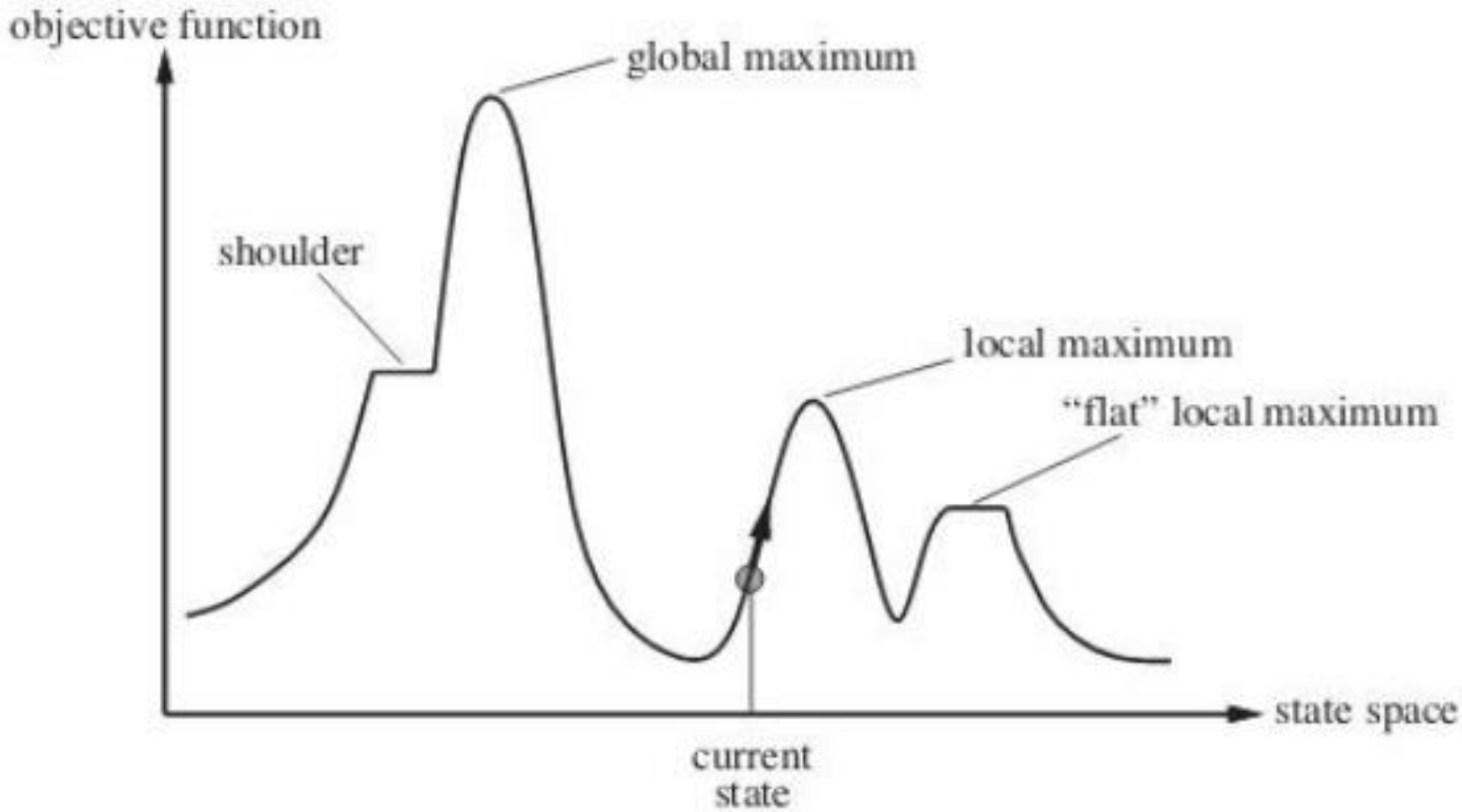
- Hill climbing is very useful in routing-related problems like travelling salesmen problem, job scheduling, chip designing, and portfolio management.
- It is good in solving optimization problems while using only limited computation power.
- It is sometimes more efficient than other search algorithms.
- Even though it may not give the optimal solution, it gives decent solutions to computationally challenging problems.

Problems with Hill Climbing

- **Local Maxima** – Can get stuck at a peak that is **not the global maximum**.
- **Plateau (Flat Region)** – Can get stuck when there is **no improvement** in heuristic value.
- **Ridges** – Cannot move forward if the best solution requires moving **down before going up**.

Solutions to These Problems

- **Random Restarts** – Run the algorithm multiple times with different starting points.
- **Simulated Annealing** – Introduces randomness to escape local maxima.
- **Better Heuristics** – Improve the function for more accurate guidance.


objective function
global maximum
shoulder
local maximum
"flat" local maximum
state space
current
state

Applications of Hill climbing

- **Artificial Intelligence** – Game AI, machine learning model tuning.
- **Robotics** – Path planning for robots and self-driving cars.
- **Scheduling** – Exam timetables, job scheduling, resource allocation.
- **Computer Vision** – Feature selection, image recognition.
- **Cryptography** – Decoding encrypted messages.

Simulated annealing

- Simulated Annealing (SA) is an advanced optimization algorithm that improves upon Hill Climbing by allowing downhill moves to escape local maxima.
- It is inspired by the annealing process in metallurgy, where metals are heated and then slowly cooled to reach a more stable structure.

Key features of Simulated annealing

- **Avoids local maxima** by allowing some bad moves.
- **Uses a probability function** to decide whether to accept a worse solution.
- **Gradually reduces randomness** over time (controlled by temperature T).
- **Useful for global optimization problems**.

Simulated Annealing Algorithm

- **Start with an initial solution**.
- **Initialize temperature T (high at the start)**.
- **Repeat until stopping condition is met**:
  - **Generate a random neighbor**.
  - **If the new state is better**, move to it.
  - **If the new state is worse**, accept it with a probability $P = e^{-\Delta E/T}$ where **ΔE** is the difference in heuristic value.
  - **Decrease the temperature gradually**.
  - **Return the best solution found**.

Simulated Annealing

- At high temperatures, it allows more bad moves to escape local maxima.
- As the temperature decreases, it behaves like Hill Climbing, focusing on better solutions.
- The probability of accepting bad moves reduces over time, leading to convergence.

Applications of Simulated Annealing

- **Route Optimization** – Finding the shortest path (e.g., Traveling Salesman Problem).
- **Neural Networks** – Optimizing weights for better learning.
- **Scheduling Problems** – Exam timetables, job scheduling, etc.
- **Circuit Design** – Minimizing wire length in VLSI design.

General principle of simulated annealing