Search Algorithms in Artificial Intelligence

Search Algorithm Terminologies

- **Search tree**: A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions**: It gives the description of all the available actions to the agent.
- **Transition model**: A description of what each action do, can be represented as a transition model.
- **Path Cost**: It is a function which assigns a numeric cost to each path.
- **Solution**: It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution**: If a solution has the lowest cost among all the possible solutions.

Properties of Search Algorithms

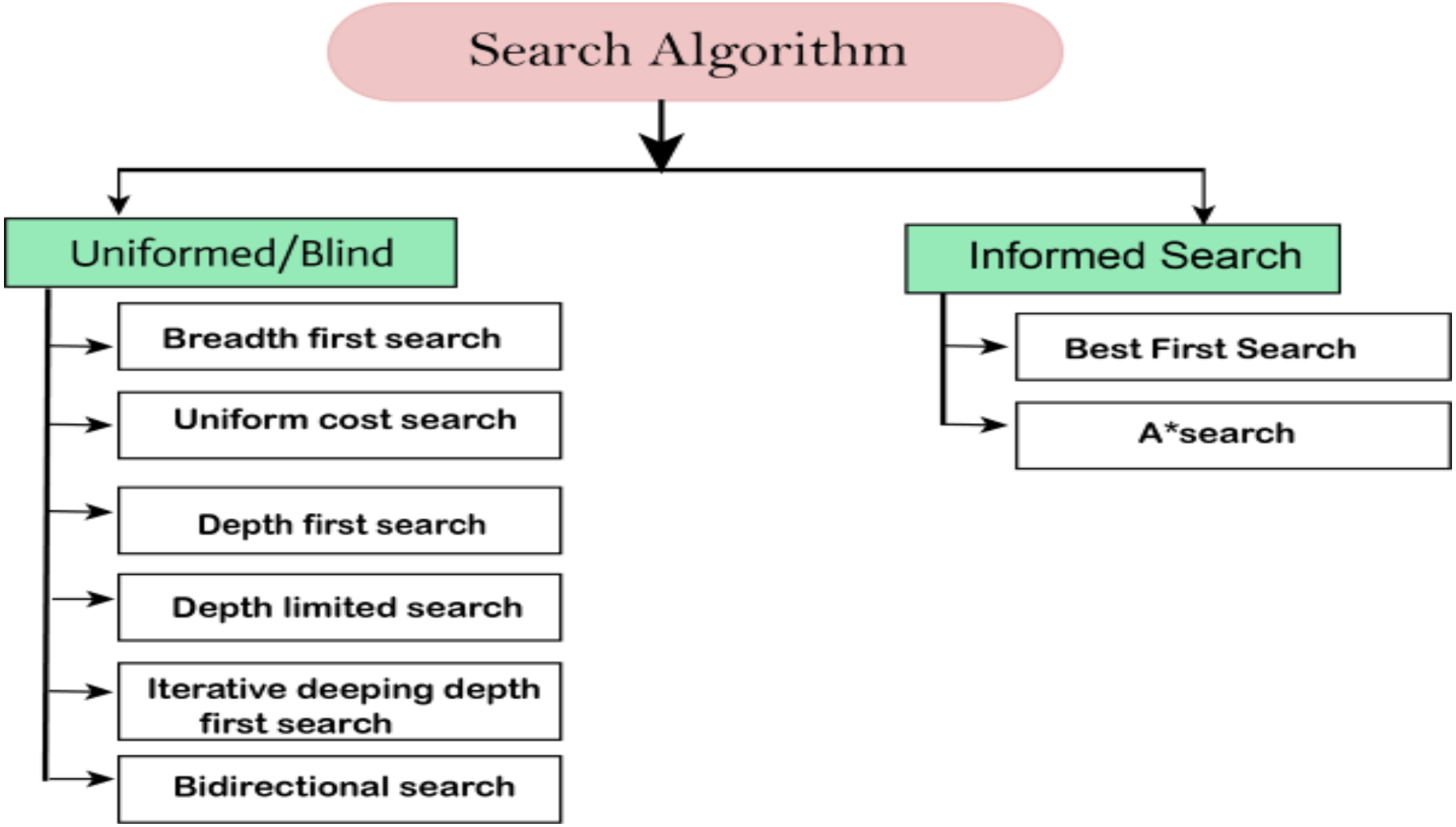
- **Completeness**: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

- **Optimality**: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for problem is said to be an optimal solution.

- **Time Complexity**: Time complexity is a measure of time for an algorithm to complete its task.

- **Space Complexity**: It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms



Uninformed/Blind Search

- Does not contain any domain knowledge such as closeness, the location of the goal.

- Operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes.

- No information about the search space, hence known as blind search.

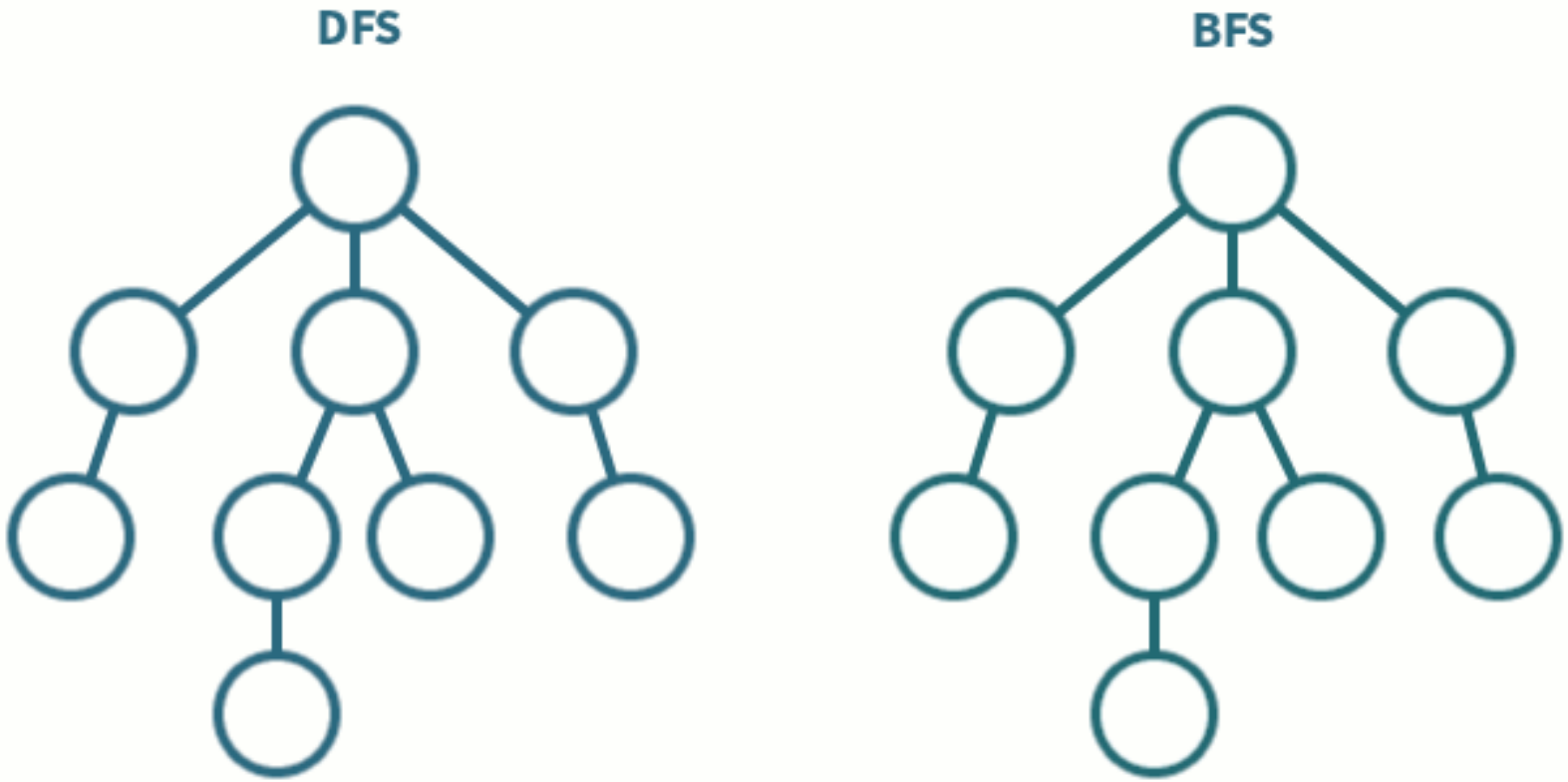
Informed Search

- Informed search algorithms use domain knowledge.
- Problem information is available which can guide the search.
- Informed search strategies can find a solution more efficiently than an uninformed search strategy.
- Informed search is also called a Heuristic search.
- A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.

Problem with uninformed/blind search

- Do not have any domain specific knowledge.
- Process of searching is drastically reduced and inefficient.

Difference Between BFS and DFS



Informed / Heuristics Search

- Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc.
- This knowledge help agents to explore less to the search space and find more efficiently the goal node.
- It is also called Heuristic search.
- Informed search algorithm is more useful for large search space.

Informed / Heuristics Search

- Two categories of problem are uses heuristics:

- Problem for which no exact algorithm are known, and one need to find an approximate and satisfying solution. E.g. Computer vision or speech recognition.

- Problem for which exact solution are known, but computationally infeasible. E.g. Rubric cube or Chess.

Heuristics function

- Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.
- The heuristic method does not always give the best solution, but it guaranteed to find a good solution in reasonable time.
- It is represented by h(n), and it calculates the cost of an optimal path between the pair of states.
- The value of the heuristic function is always positive.

Heuristics function

- Admissibility of the heuristic function is given as:

$$h(n) <= h^*(n)$$

- Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

- Some simple heuristic functions:
  - 8-tile puzzle: hamming distance is used
  - Chess Problem: Material Advantage is used

Best-first Search Algorithm

- Best-first search algorithm always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms.
- With the help of best-first search, at each step, we can choose the most promising node.
- In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

f(n)= g(n).

Best First Search Algorithm

Step 1. Put the initial node on a list START.

Step 2. If (START is empty) or (START=GOAL), terminate search.

Step 3. Remove the first node from START, call it node a.

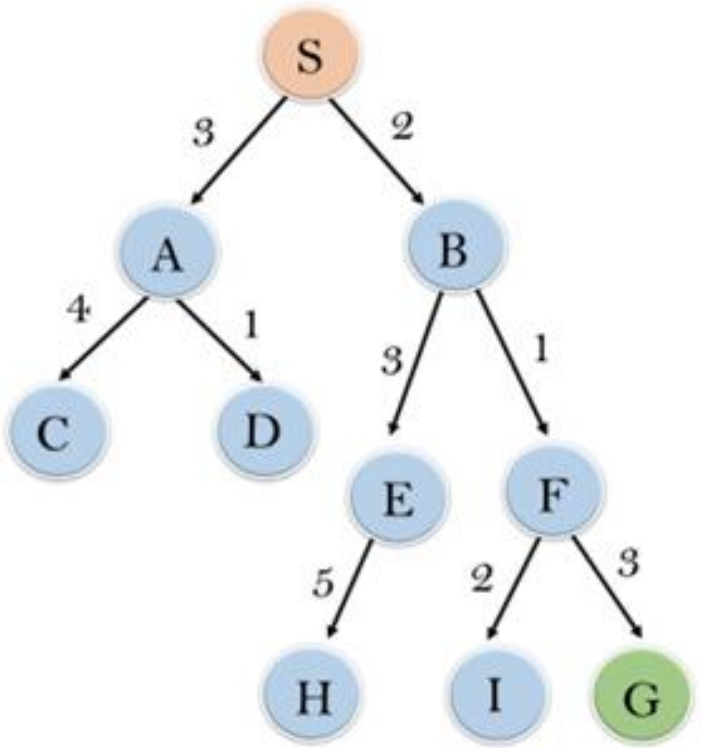
Step 4. If (a=GOAL) terminate search with success.

Step 5. Else if node a has successor, generate all of them. Find out how far they are from the goal node. Sort all the children generated so far by the remaining distance from the goal.

Step 6. Name the list as START 1 and replace it with list START.

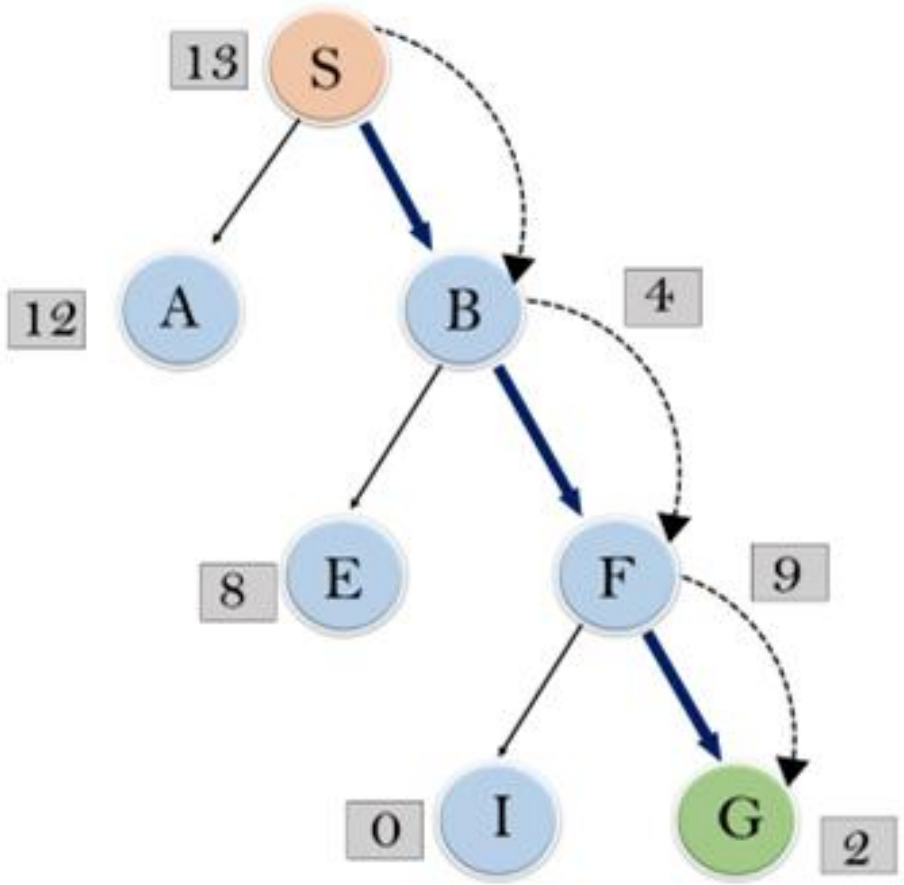
Step 7. Go to step 2.

Example



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

Example



Best-first Search Algorithm

- Expand the nodes of S and put in the CLOSED list
- Initialization: Open [A, B], Closed [S]
- Iteration 1: Open [A], Closed [S, B]
- Iteration 2: Open [E, F, A], Closed [S, B]
  : Open [E, A], Closed [S, B, F]
- Iteration 3: Open [I, G, E, A], Closed [S, B, F]
  : Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

Best-first Search Algorithm

- **Advantages:**
  - Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
  - This algorithm is more efficient than BFS and DFS algorithms.

- **Disadvantages:**
  - It can behave as an unguided depth-first search in the worst case scenario.
  - It can get stuck in a loop as DFS.
  - This algorithm is not optimal.

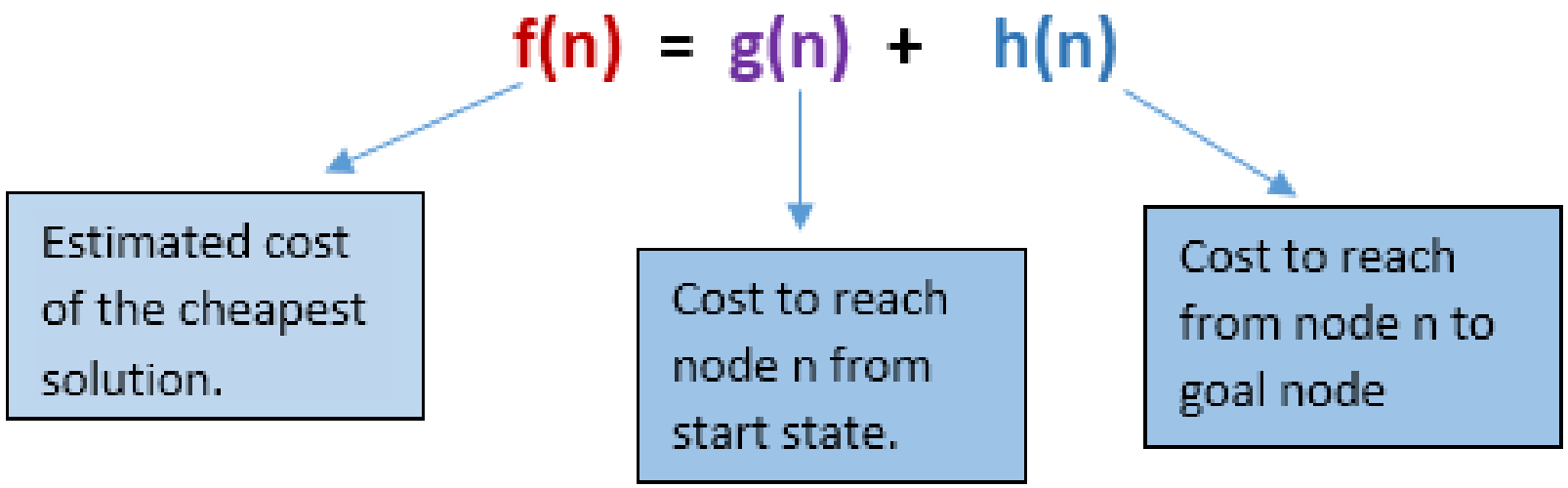
Best-first Search Algorithm

- Time Complexity: The worst-case time complexity of Greedy best first search is O(bm).
- Space Complexity: The worst-case space complexity of Greedy best first search is O(bm). Where, m is the maximum depth of the search space.
- Complete: Greedy best-first search is also incomplete, even if the given state space is finite.
- Optimal: Greedy best first search algorithm is not optimal.

A* Search Algorithm

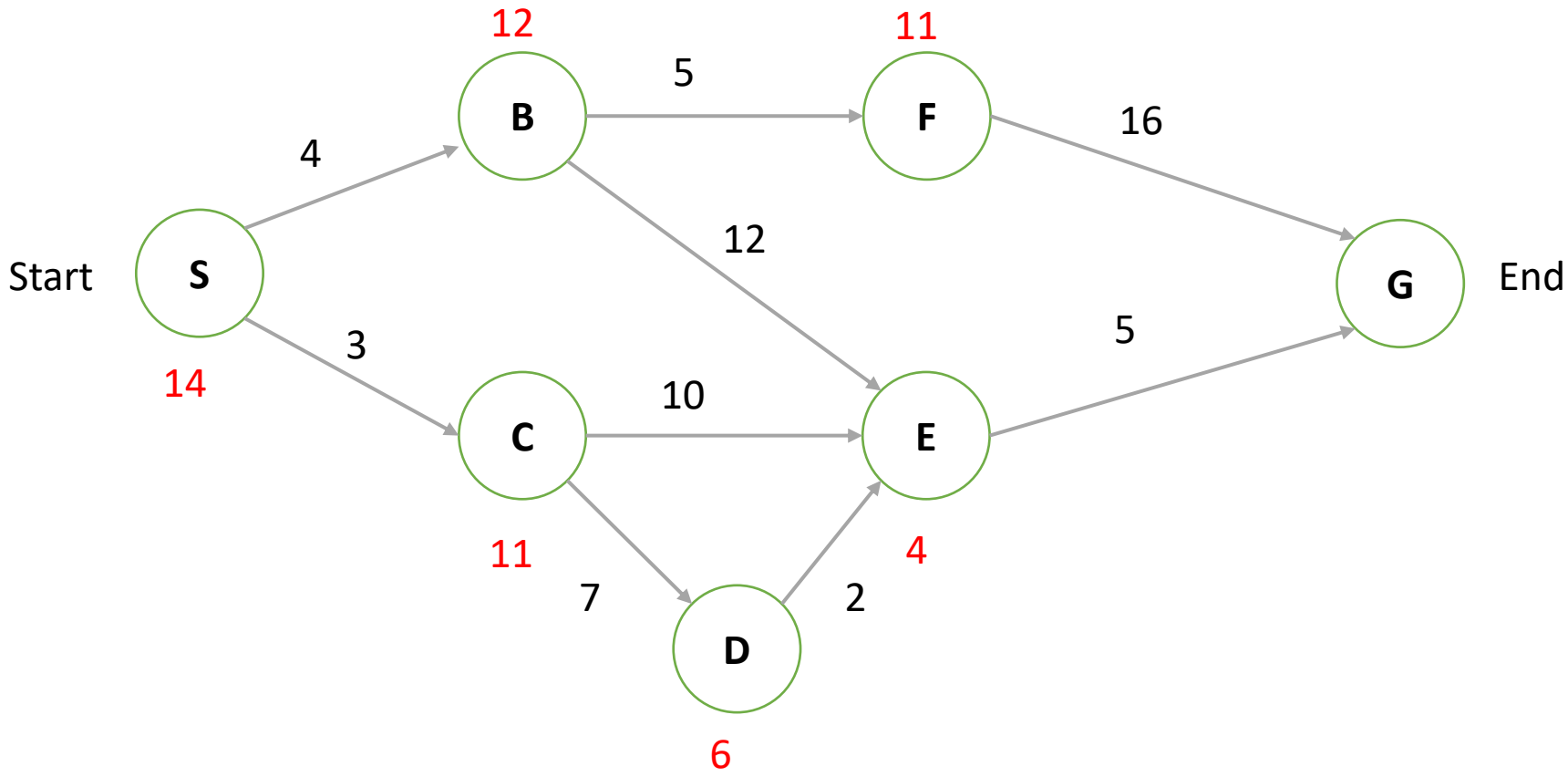
- A* uses heuristic function h(n), and cost to reach the node n from the start state g(n).
- A* search algorithm finds the shortest path through the search space using the heuristic function.
- This search algorithm expands less search tree and provides optimal result faster.
- * means admissible.
- A* is a admissible algorithm means it will always give optimal result.

A* Search Algorithm

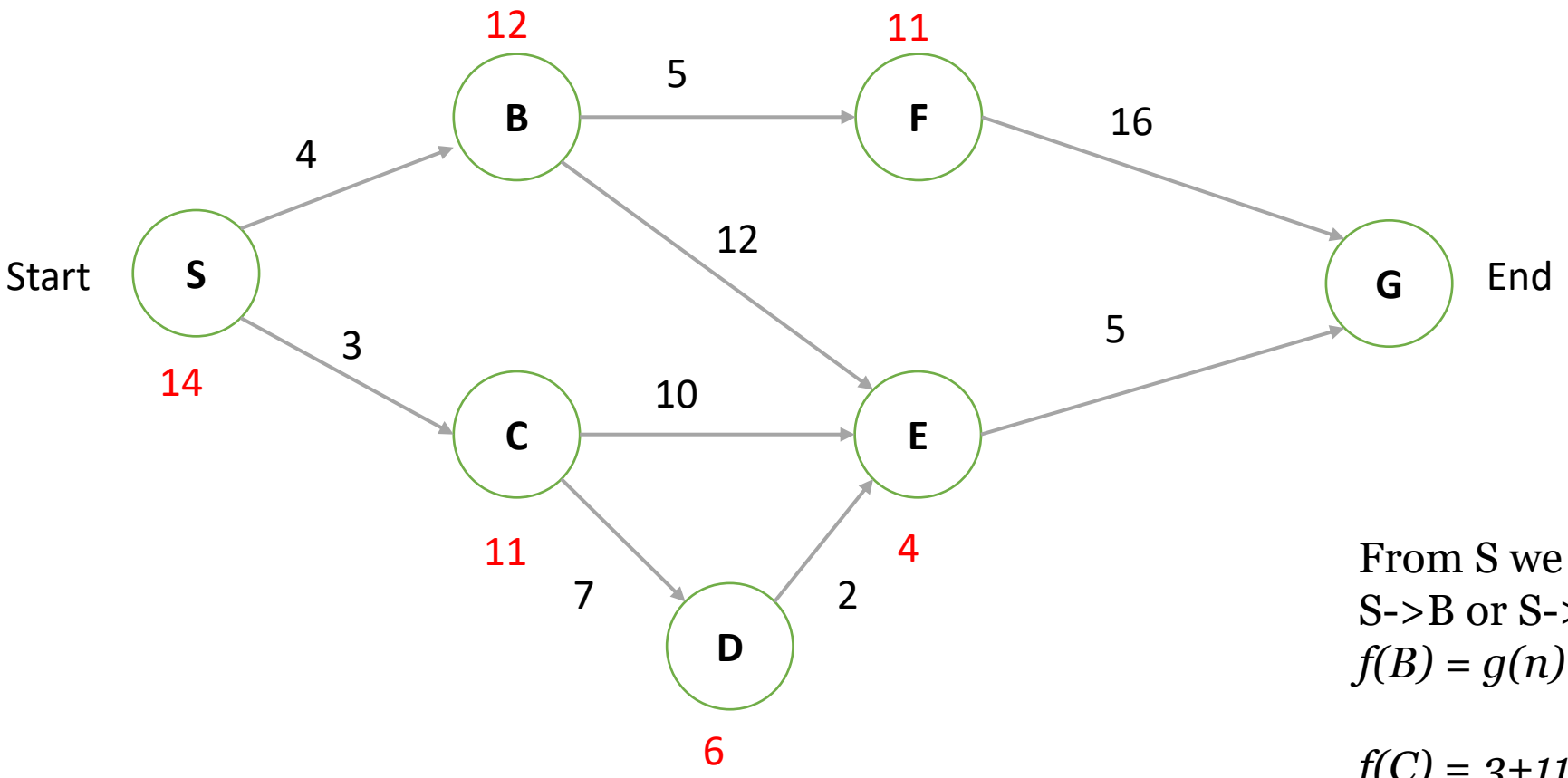


- In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.
- At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

Example



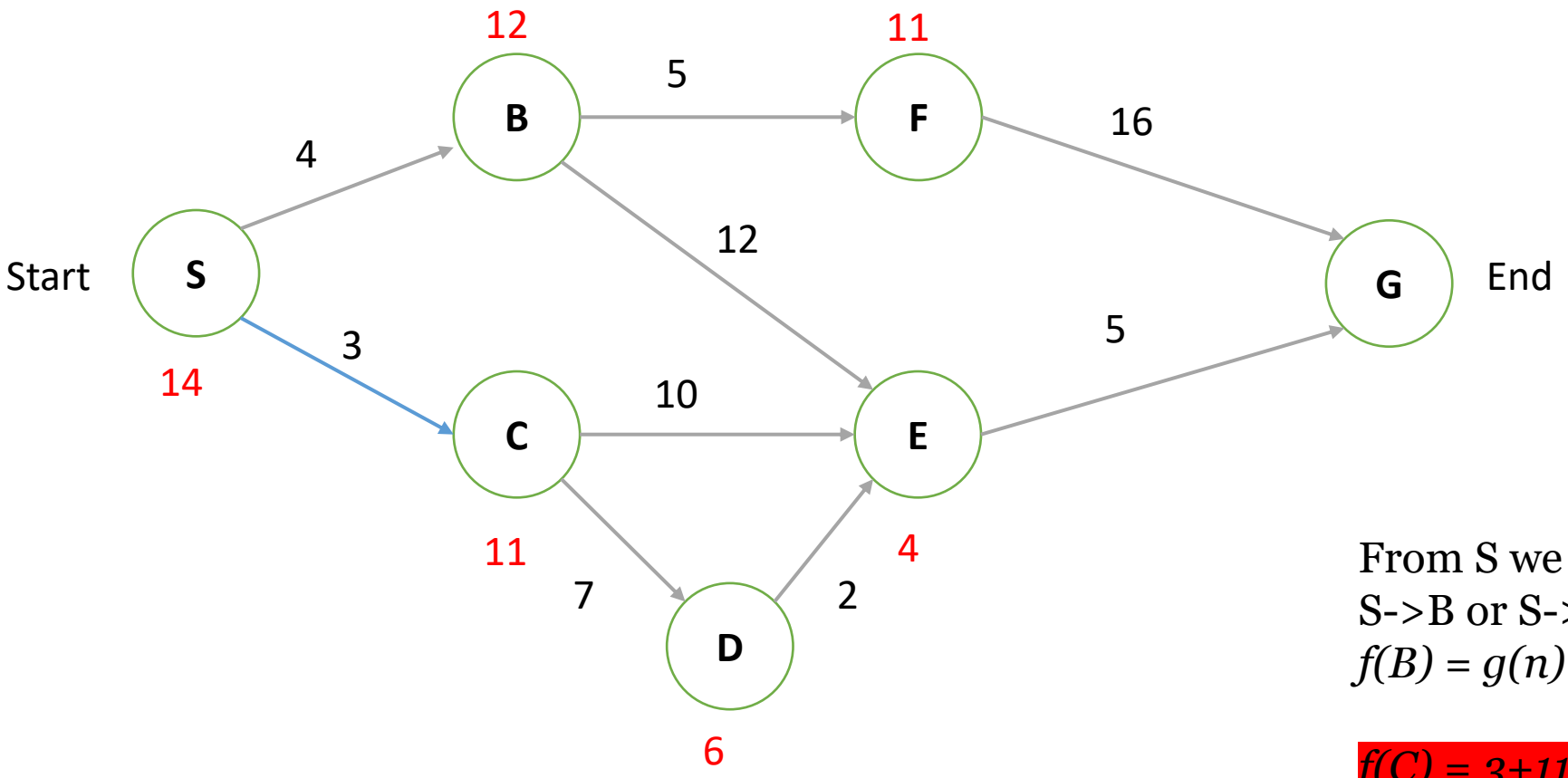
Example



From S we can move:
S->B or S->C
$f(B) = g(n) + h(n) = 4+12 = 16$

$f(C) = 3+11 = 14$

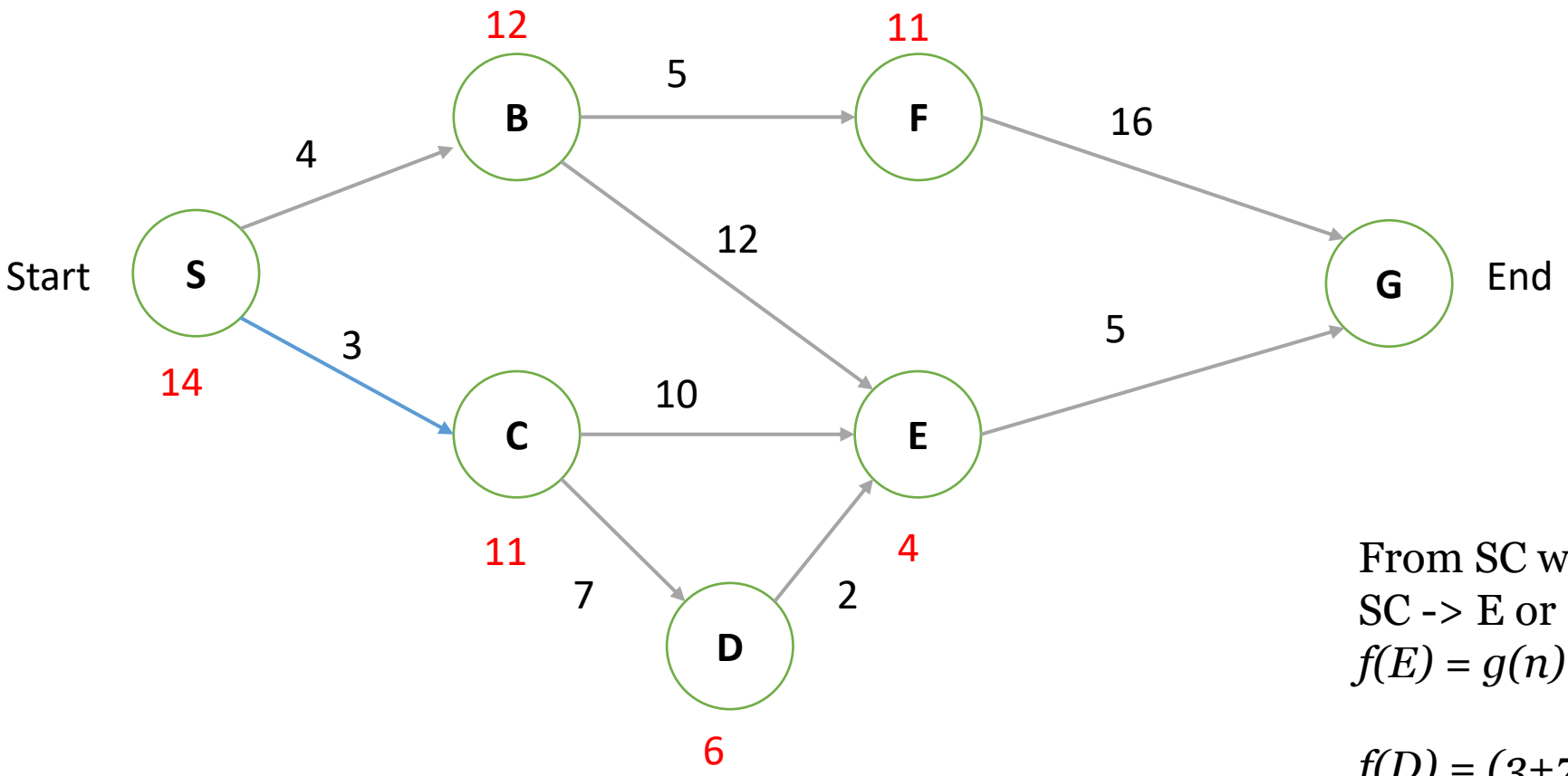
Example



From S we can move:
S->B or S->C
$f(B) = g(n) + h(n) = 4+12 = 16$

$f(C) = 3+11 = 14$

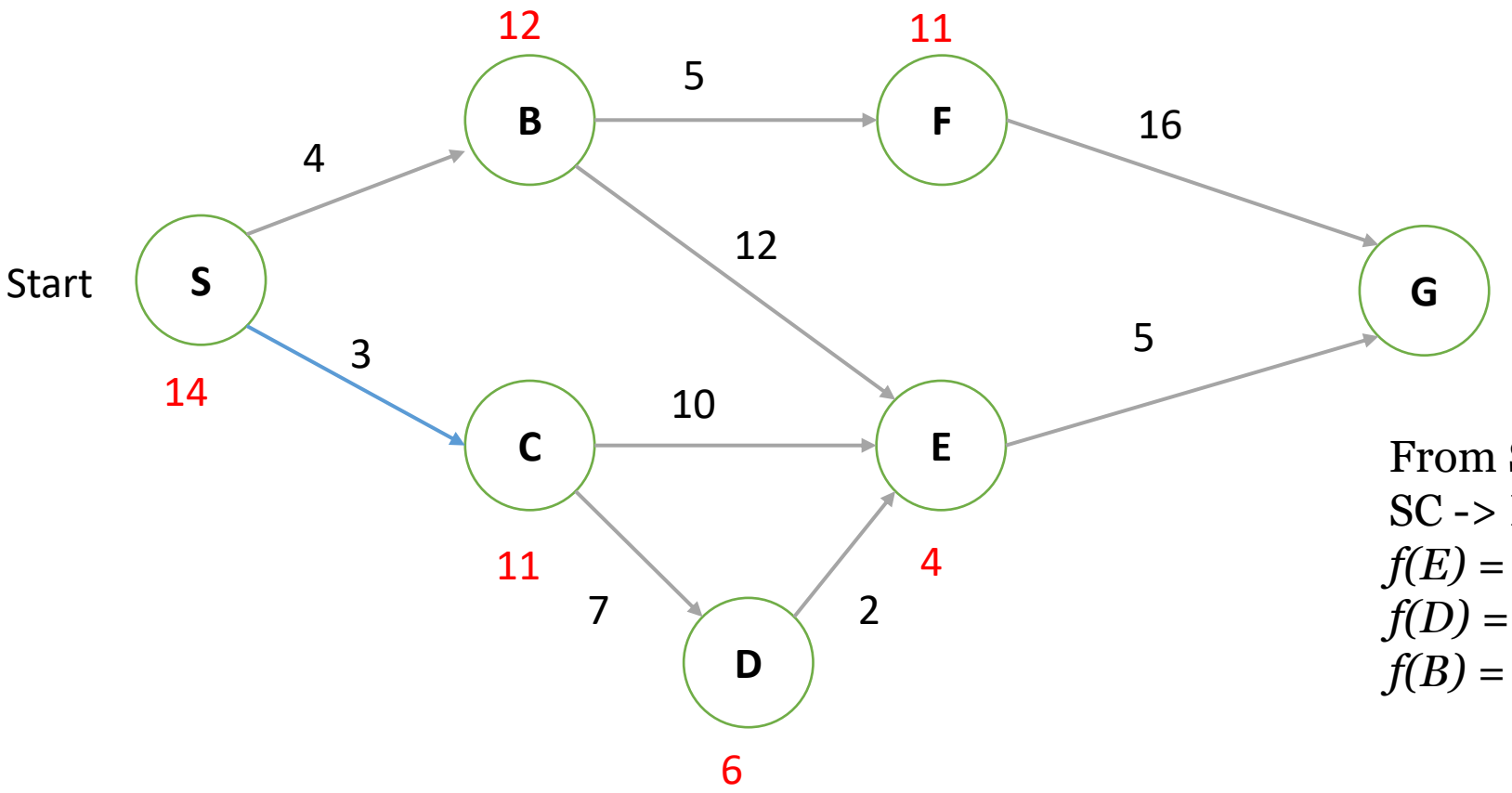
Example



From SC we can move:
SC -> E or SC -> D
$f(E) = g(n) + h(n) = (3+10)+4 = 17$

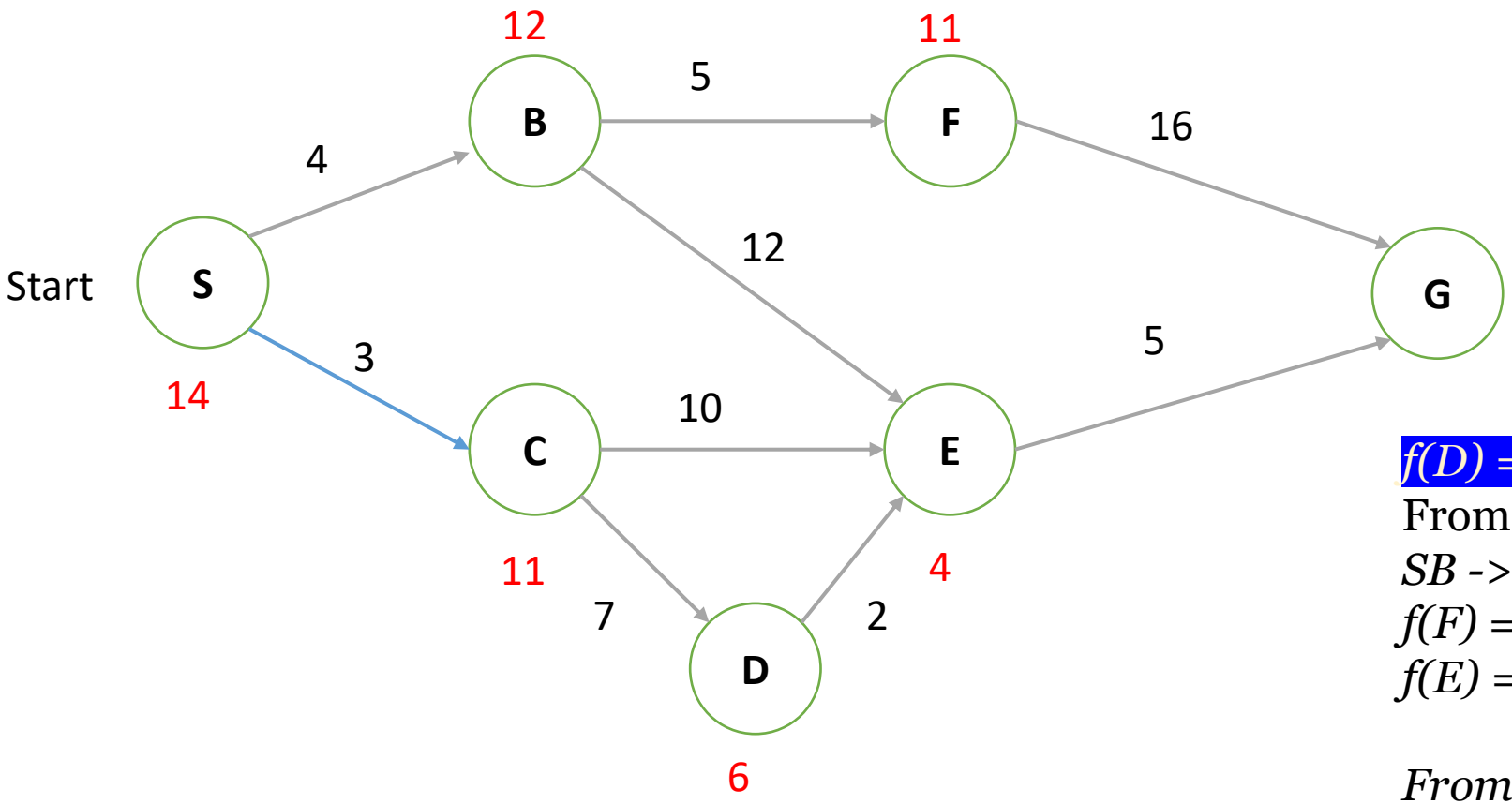
$f(D) = (3+7)+6 = 16$

Example



From SC we can move:
SC -> E or SC -> D
$f(E) = g(n) + h(n) = (3+10)+4 = 17$
$f(D) = (3+7)+6 = 16$
$f(B) = 16$

Example



f(D) = 16, f(B) = 16

From SB we can move:

SB -> F or SB -> E

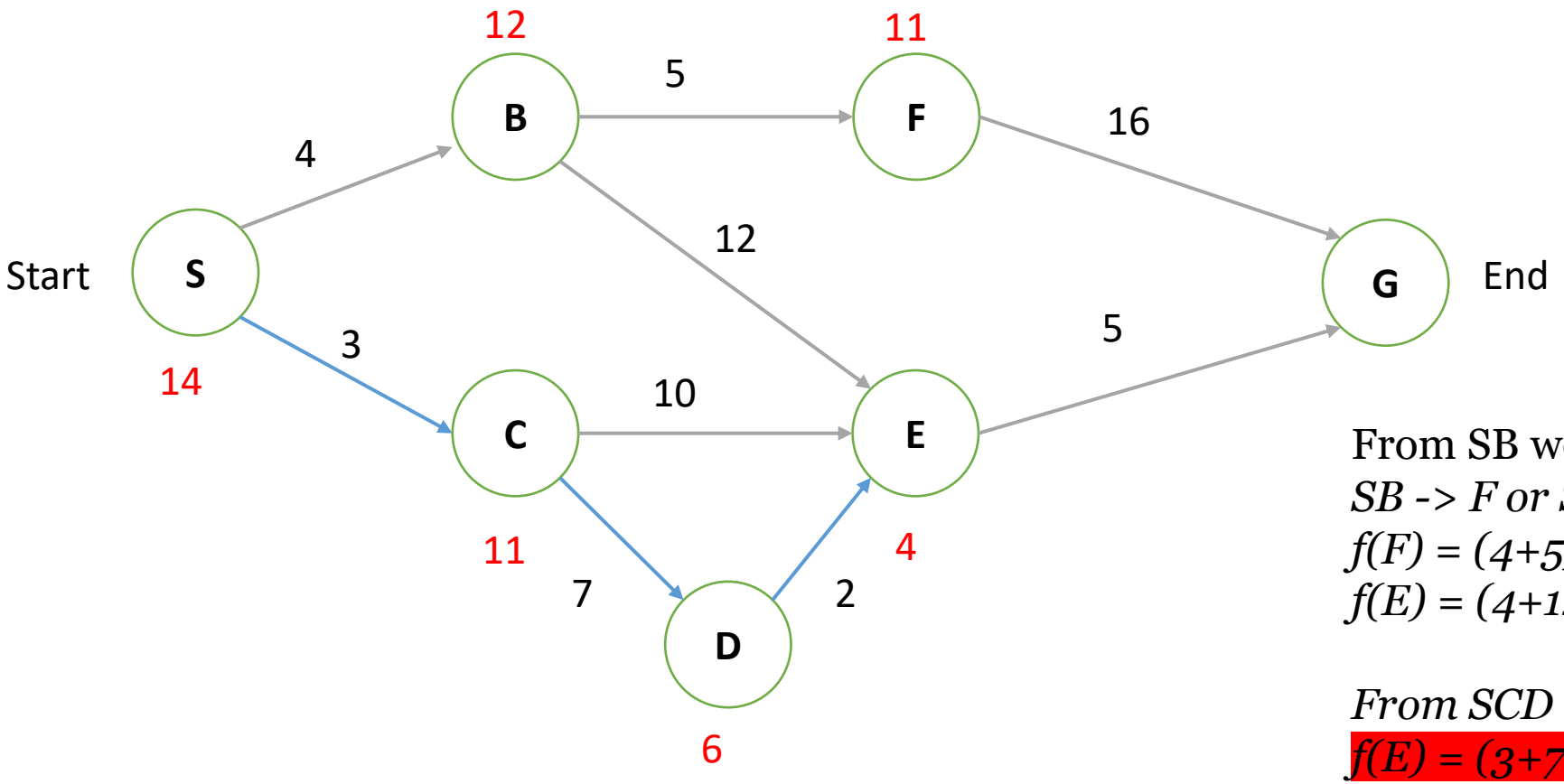
f(F) = (4+5)+11 = 20

f(E) = (4+12)+4 = 20

From SCD we can move to: E

f(E) = (3+7+2)+4 = 16

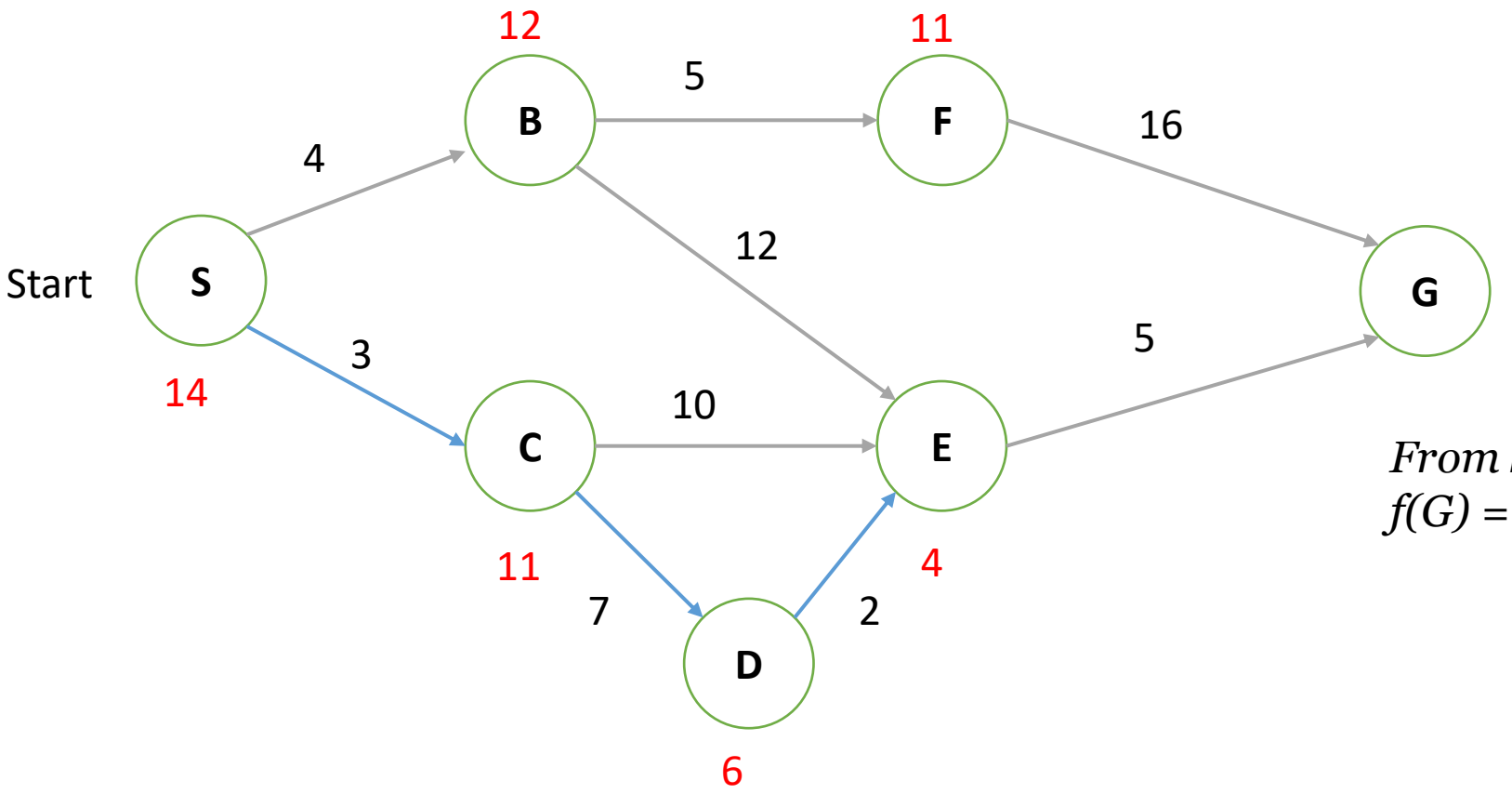
Example



From SB we can move:
SB -> F or SB -> E
f(F) = (4+5)+11 = 20
f(E) = (4+12)+4 = 20

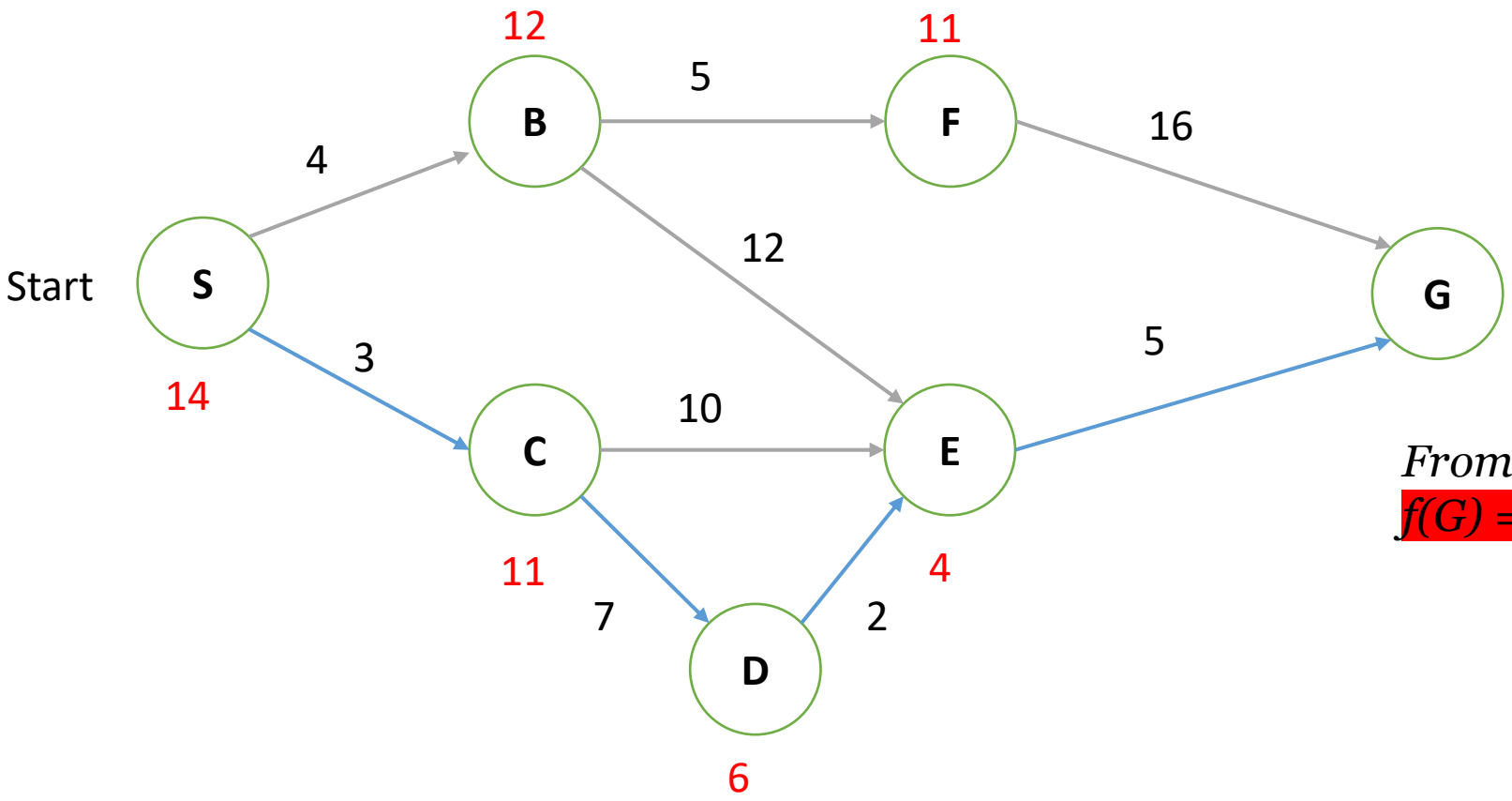
From SCD we can move to: E
f(E) = (3+7+2)+4 = 16

Example



From SCDE we can move to: G
f(G) = (3+7+2+5) + 0 = 17

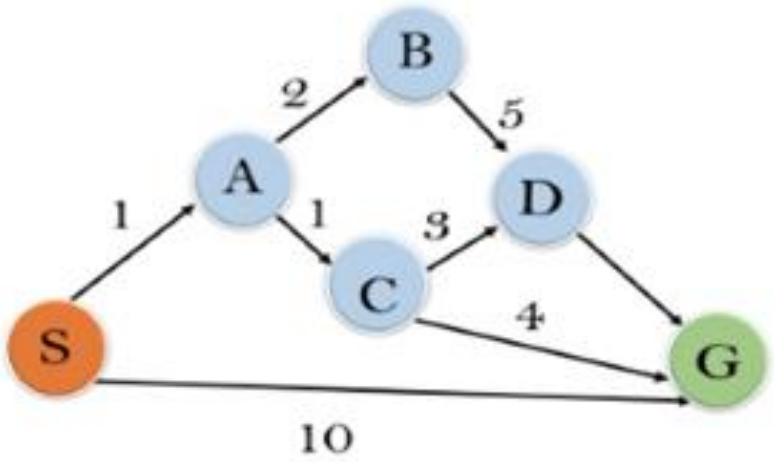
Example



From SCDE we can move to: G

f(G) = (3+7+2+5) + 0 = 17

Example



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

Algorithm of A* search

Step 1. Put the initial node on a list START.

Step 2. If (START is empty) or (START=GOAL), terminate search.

Step 3. Remove the first node from START, call it node a.

Step 4. If (a=GOAL) terminate search with success.

Step 5. Else if node a has successor, generate all of them. Estimate the fitness number of the successor by totalling the evaluation function value and cost function value. Sort the list by fitness number.

Step 6. Name the list as START 1 and replace it with list START.

Step 7. Go to step 2.

Time and Space Complexity

- **Complete**: A* algorithm is complete as long as **Branching factor is finite** and **Cost at every path is fixed**.
- **Optimal**: A* search algorithm is optimal if h(n) should be an admissible heuristic for A* tree search.
- **Consistency**: Second required condition is consistency for only A* graph-search.
- **Time Complexity**: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is **O(b^d)**, where b is the branching factor.
- **Space Complexity**: The space complexity of A* search algorithm is **O(b^d)**.

A* Search Algorithm

- **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. **(Underestimation and Overestimation)**
  - A* search algorithm has some complexity issues.
  - 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.

AO* Algorithm

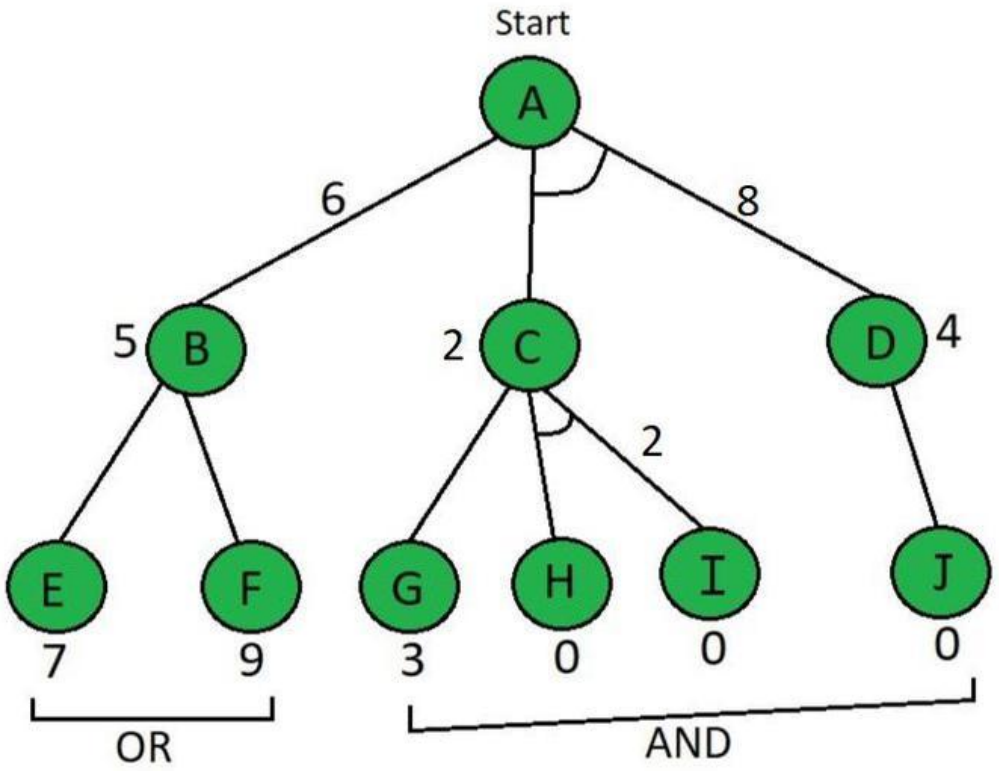
- The AO* method divides any given difficult problem into a smaller group of problems that are then resolved using the AND-OR graph concept.

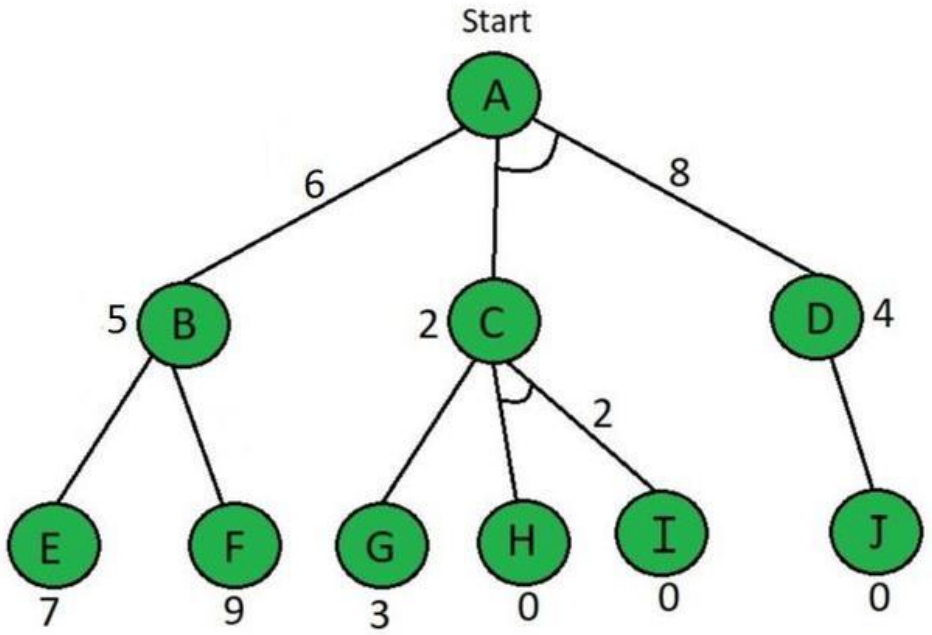
- The evaluation function in AO* looks like this:
- f(n) = g(n) + h(n)
- f(n) = Actual cost + Estimated cost

A* algorithm Vs AO* algorithm

- A* algorithm and AO* algorithm both works on the best first search.
- They are both informed search and works on given heuristics values.
- A* always gives the optimal solution but AO* doesn't guarantee to give the optimal solution.
- Once AO* got a solution doesn't explore all possible paths but A* explores all paths.
- When compared to the A* algorithm, the AO* algorithm uses less memory.
- opposite to the A* algorithm, the AO* algorithm cannot go into an endless loop.

Example



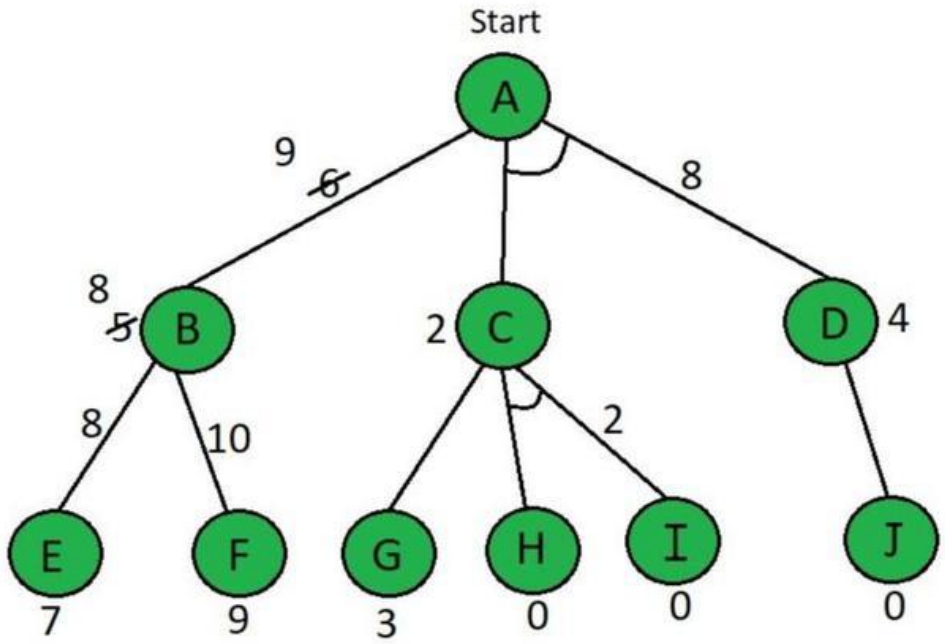


With help of f(n) = g(n) + h(n) evaluation function,
Start from node A,

f(A⇢B) = g(B) + h(B) = 1 + 5
here g(n)=1 is taken by default for path cost = 6

f(A⇢C+D) = g(c) + h(c) + g(d) + h(d) = 1 + 2 + 1 + 4 = 8

So, by calculation A⇢B path is chosen which is the minimum path,
i.e f(A⇢B)



Explore node B

Here the value of E & F are calculated as follows,

f(B⇢E) = g(E) + h(E) = 1 + 7 = 8

f(B⇢F) = g(F) + h(F) = 1 + 9 = 10

So, by above calculation B⇢E path is chosen which is minimum path, i.e f(B⇢E)

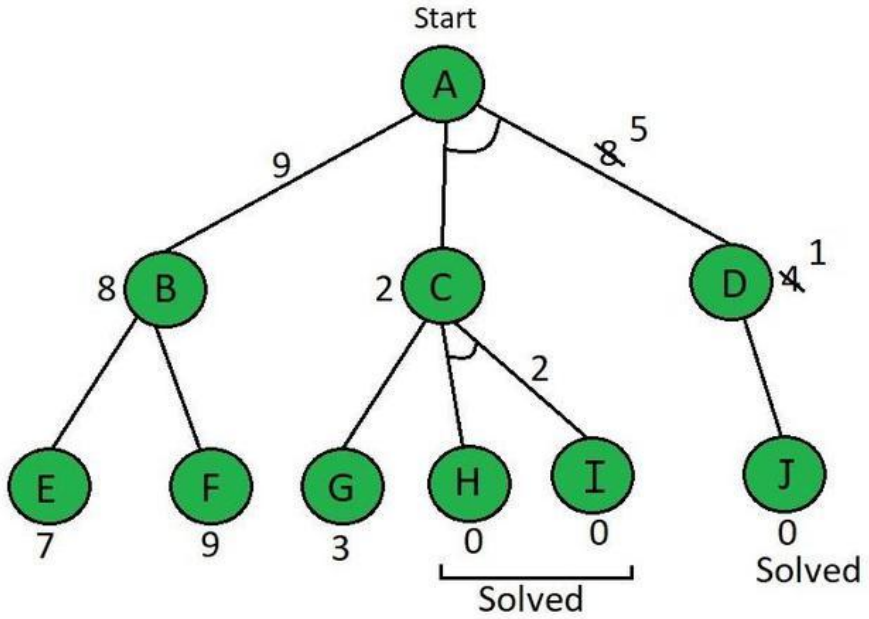
Because B's heuristic value is different from its actual value. The heuristic is updated and the minimum cost path is selected.

The minimum value in our situation is 8.

Therefore, the heuristic for A must be updated due to the change in B's heuristic.

f(A⇢B) = g(B) + updated h(B) = 1 + 8 = 9

We have Updated all values in the above tree.



By comparing f(A⇢B) & f(A⇢C+D), f(A⇢C+D) is shown to be smaller. i.e 8 < 9

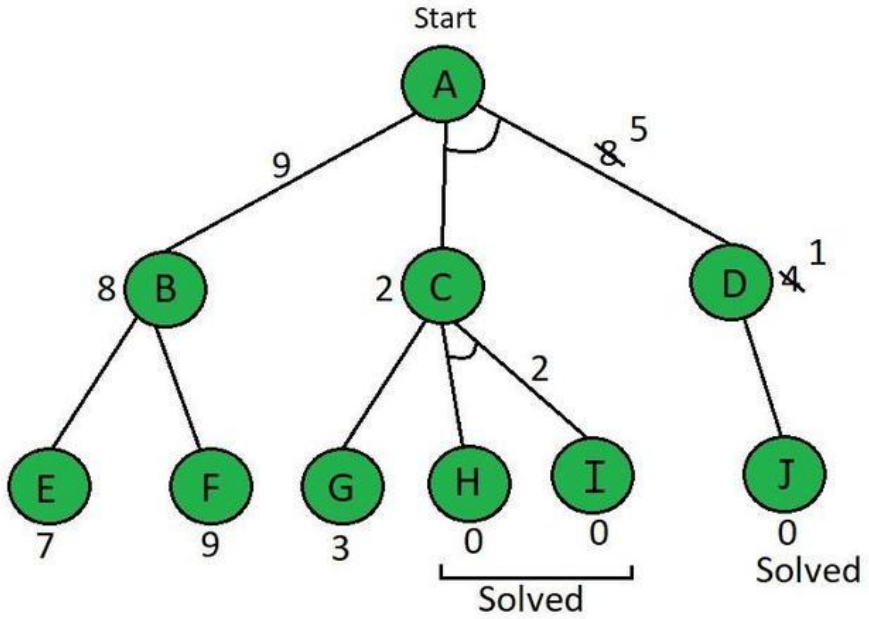
Now explore f(A⇢C+D). So, the current node is C.

f(C⇢G) = g(g) + h(g) = 1 + 3 = 4

f(C⇢H+I) = g(h) + h(h) + g(i) + h(i) = 1 + 0 + 1 + 0 = 2

f(C⇢H+I) is selected as the path with the lowest cost and the heuristic is also left unchanged.

Because it matches the actual cost. Paths H & I are solved because the heuristic for those paths is 0.



f(D⇢J) = g(j) + h(j) = 1 + 0 = 1

the heuristic of node D needs to be updated to 1.

f(A⇢C+D) = g(c) + h(c) + g(d) + h(d) = 1 + 2 + 1 + 1 = 5

as we can see that path f(A⇢C+D) is get solved and this tree has become a solved tree now.

In simple words, the main flow of this algorithm is that we have to find firstly level 1st heuristic value and then level 2nd and after that update the values with going upward means towards the root node.

In the above tree diagram, we have updated all the values.

Real-Life Applications of AO* algorithm

- Vehicle Routing Problem:
  - The vehicle routing problem is determining the shortest routes for a fleet of vehicles to visit a set of customers and return to the depot, while minimizing the total distance traveled and the total time taken. The AO* algorithm can be used to find the optimal routes that satisfy both objectives.
- Portfolio Optimization:
  - Portfolio optimization is choosing a set of investments that maximize returns while minimizing risks. The AO* algorithm can be used to find the optimal portfolio that satisfies both objectives, such as maximizing the expected return and minimizing the standard deviation.

Iterative deepening A* (IDA*)

- It perform depth first search with limited to some f-bound.
- Uses the formula: f(n) = h(n) + g(n)

- Algorithm:
- Perform depth-first search limited to some f-bound.
- If goal found: OK
- Else: increase the f-bound and restart.

Small Memory A* (SMA*)

- Like A* search, SMA* search is an optimal and complete algorithm for finding a least-cost path.
- Unlike A*, SMA* will not run out of memory, unless the size of the shortest path exceeds the amount of space in available memory.
- SMA* addresses the possibility of running out of memory by pruning the portion of the search-space that is being examined.

Heuristic Techniques

- Heuristic techniques are problem-solving methods that prioritize speed and efficiency over optimality. These methods are commonly used in situations where finding an exact solution is computationally expensive or impractical. Heuristic techniques provide approximate solutions that are often "good enough" for practical purposes. Here are some common heuristic techniques:

Greedy Algorithms:

- Greedy algorithms make decisions based on the current best option without considering the global optimal solution. At each step, the algorithm selects the locally optimal choice, hoping that it will lead to a good solution overall. Greedy algorithms are easy to implement and often efficient but may not always produce the best possible solution.

Local Search:

- Local search algorithms explore a solution space by iteratively moving from one solution to a neighboring solution that improves some objective function. These algorithms start with an initial solution and repeatedly make small modifications to improve it. Examples include hill climbing, simulated annealing, and genetic algorithms.

Rule-Based Systems:

- Rule-based systems use a set of predefined rules to make decisions or perform tasks. These systems are often used in expert systems, where human expertise is encoded into a set of rules that guide problem-solving or decision-making processes.

Generate and Test

- **Description:** In the generate and test approach, a solution is generated and then tested to determine if it meets the desired criteria. If the solution is satisfactory, it is accepted; otherwise, the process is repeated until an acceptable solution is found.
- **Example:** In a software development context, when designing an algorithm to solve a particular problem, developers may generate multiple solutions based on different approaches (e.g., brute force, divide and conquer, dynamic programming) and test each solution against a set of test cases to evaluate its performance and effectiveness.

Hill Climbing:

- **Description:** Hill climbing is a local search algorithm that iteratively improves a solution by making incremental changes at each step. The algorithm evaluates neighboring solutions and selects the one that maximizes (or minimizes) an objective function, moving in the direction of the steepest ascent (or descent) until a peak (or valley) is reached.
- **Example:** In route optimization, hill climbing can be used to find the shortest path between two points on a map by iteratively adjusting the route based on the distance to the destination. At each step, the algorithm evaluates neighboring routes and selects the one that brings it closer to the destination until an optimal path is found.

State Space Search:

- **Description:** State space search involves exploring the set of possible states of a problem to find a solution. Each state represents a configuration or arrangement of elements, and transitions between states are determined by applying operators or actions. The search proceeds by systematically exploring the state space until a goal state is reached.
- **Example:** In puzzle-solving, such as the eight puzzle or Rubik's Cube, state space search is used to find a sequence of moves that transforms the initial configuration of the puzzle into the goal configuration. The algorithm explores different states of the puzzle by applying valid moves (e.g., sliding puzzle tiles or rotating cube faces) until the goal state is achieved.

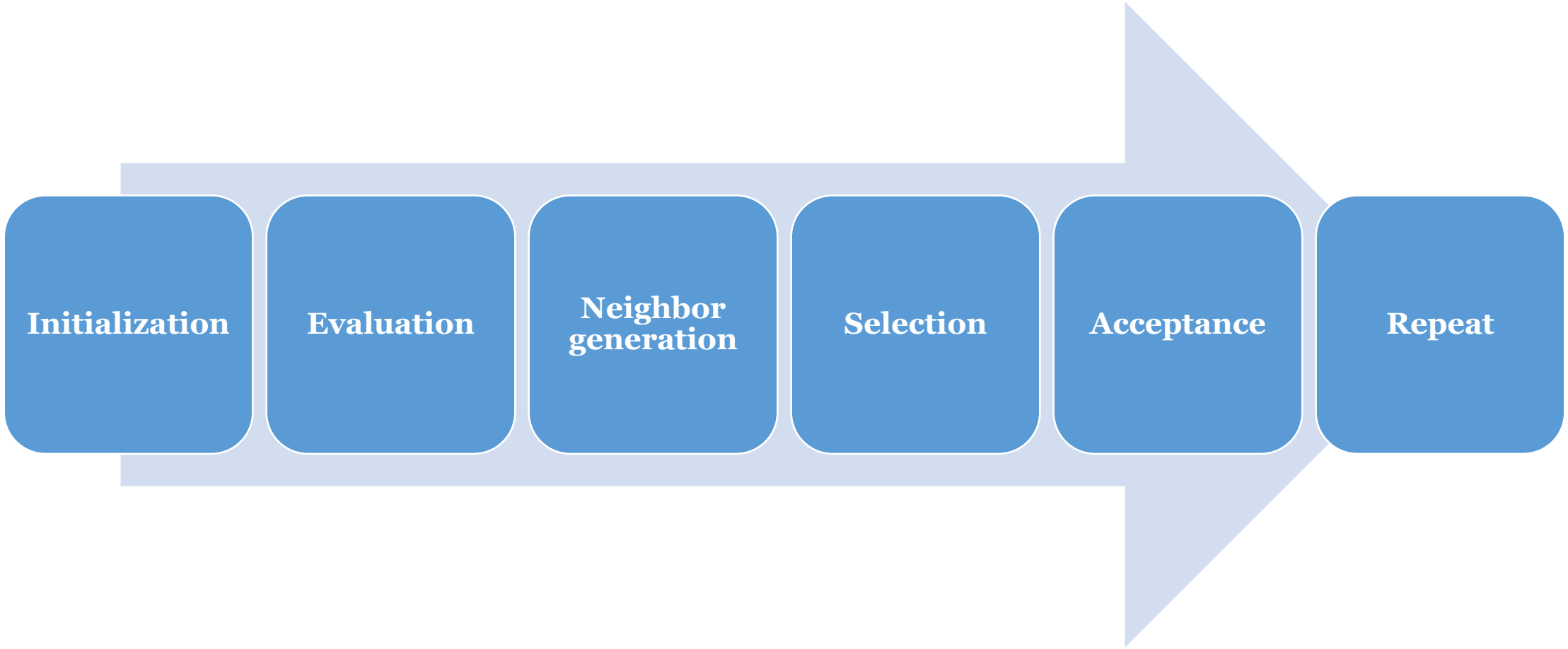
Constraint Satisfaction Problem (CSP):

- **Description:** A constraint satisfaction problem involves finding a solution that satisfies a set of constraints or conditions. The problem typically consists of a set of variables, each with a domain of possible values, and a set of constraints that restrict the allowable combinations of variable values. The goal is to find an assignment of values to variables that satisfies all constraints.
- **Example:** Sudoku puzzles can be formulated as constraint satisfaction problems, where the goal is to fill in the empty cells of the grid with digits from 1 to 9 such that each row, column, and 3x3 subgrid contains each digit exactly once. The constraints are the rules of Sudoku, and the solution is a valid assignment of digits to cells that satisfies all constraints.

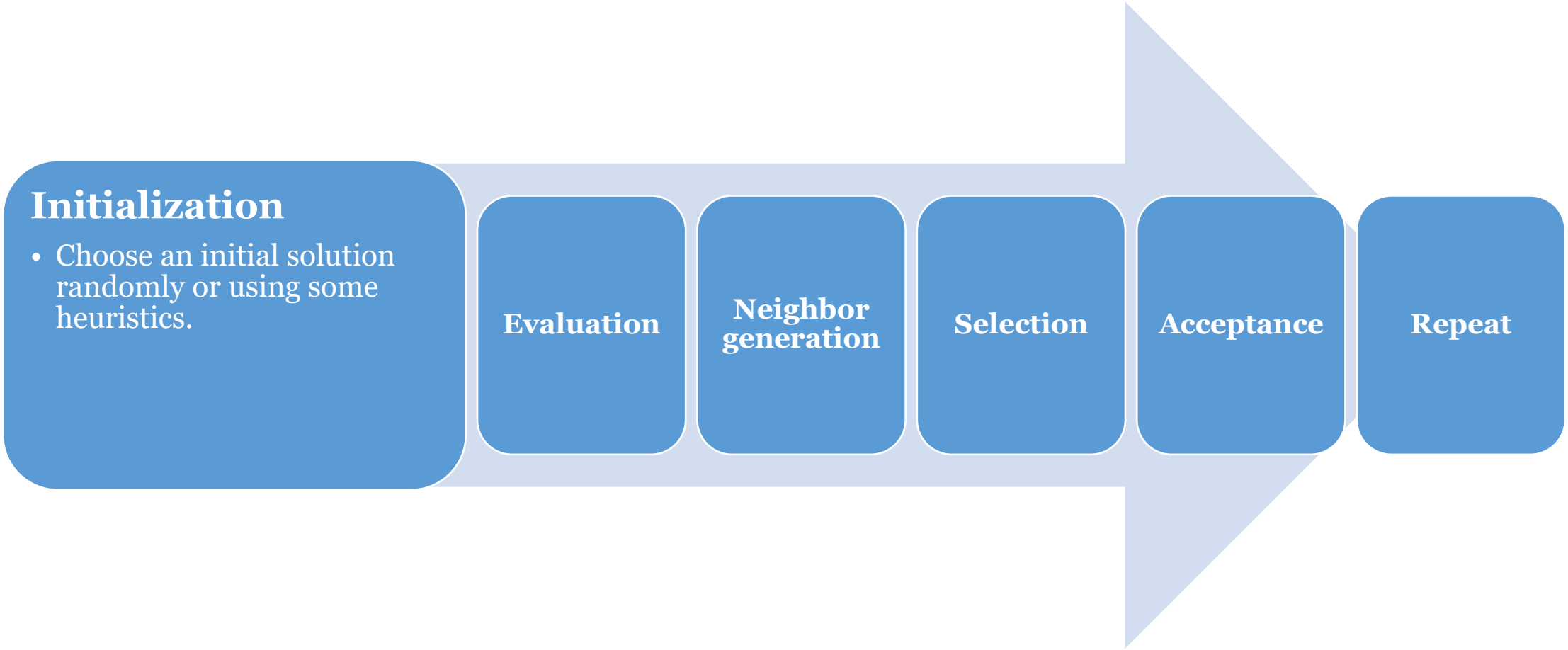
Hill Climbing Algorithm

- The Hill Climbing Algorithm is a local search algorithm that can be used in Artificial Intelligence (AI) to solve optimization problems.
- It is a simple and intuitive algorithm that iteratively searches for the best solution in the local neighborhood of the current solution.
- The algorithm starts with an initial solution and evaluates the neighboring solutions by making small changes to the current solution.
- The algorithm accepts a new solution if it has a better score than the current solution.
- The algorithm continues until it reaches a local optimum, i.e., a solution that is better than its neighbors but not better than other solutions that may be further away.
- The Hill Climbing Algorithm can be used for a wide range of optimization problems in AI, including machine learning, robotics, and game-playing.

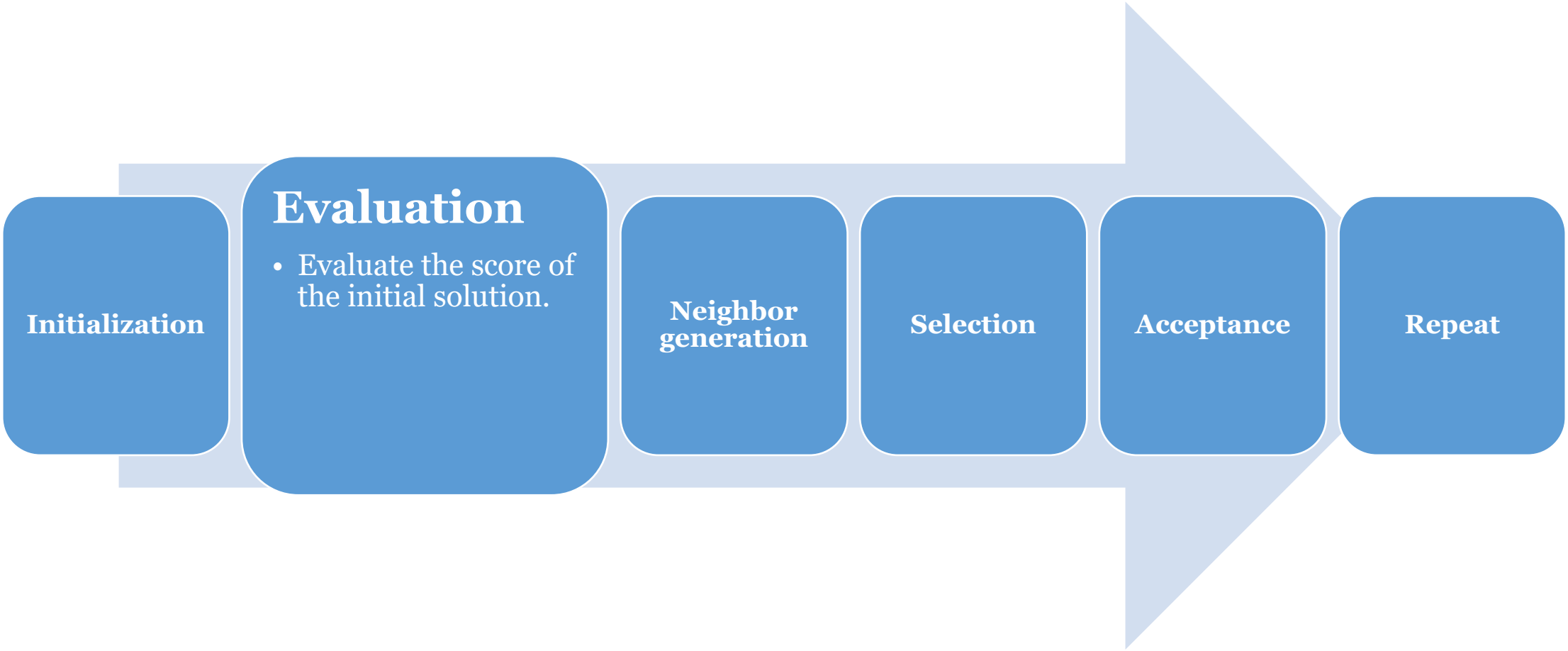
General outline of Hill Climbing Algorithm working



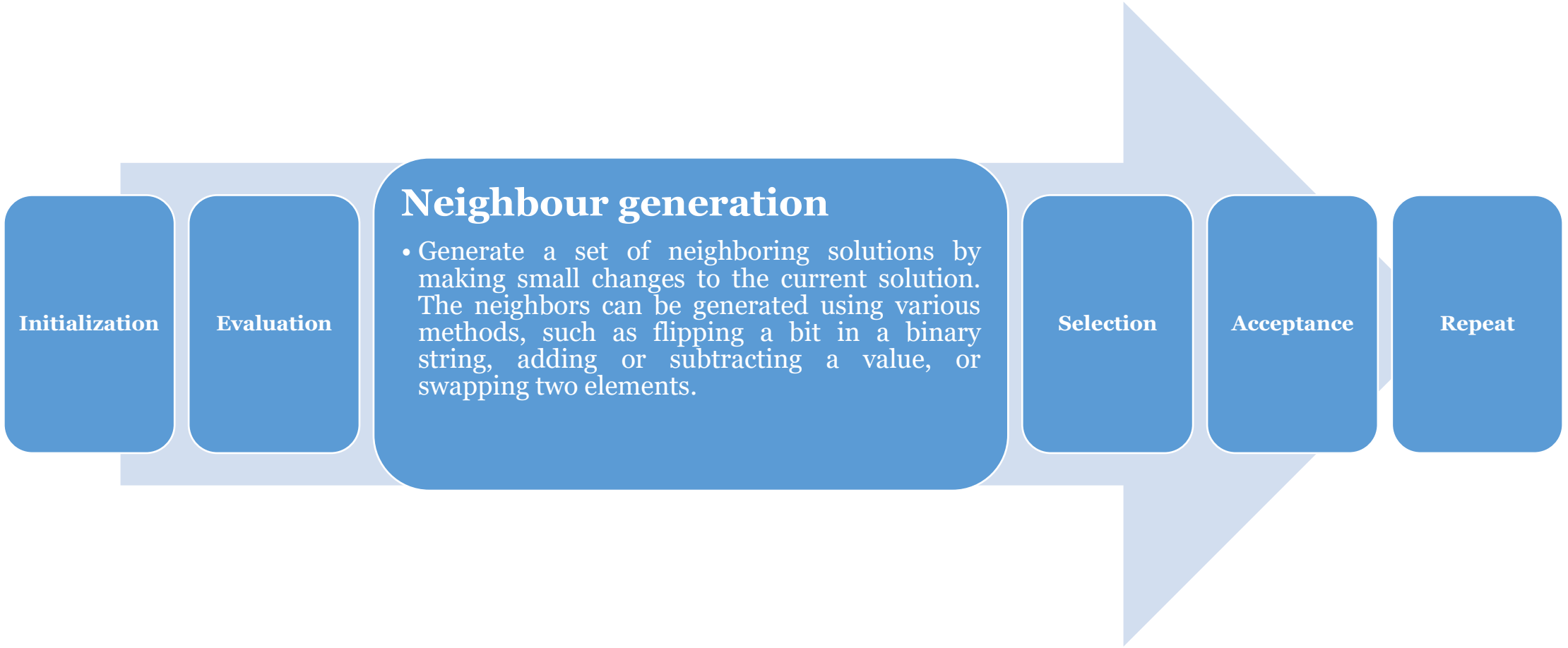
General outline of Hill Climbing Algorithm working



General outline of Hill Climbing Algorithm working

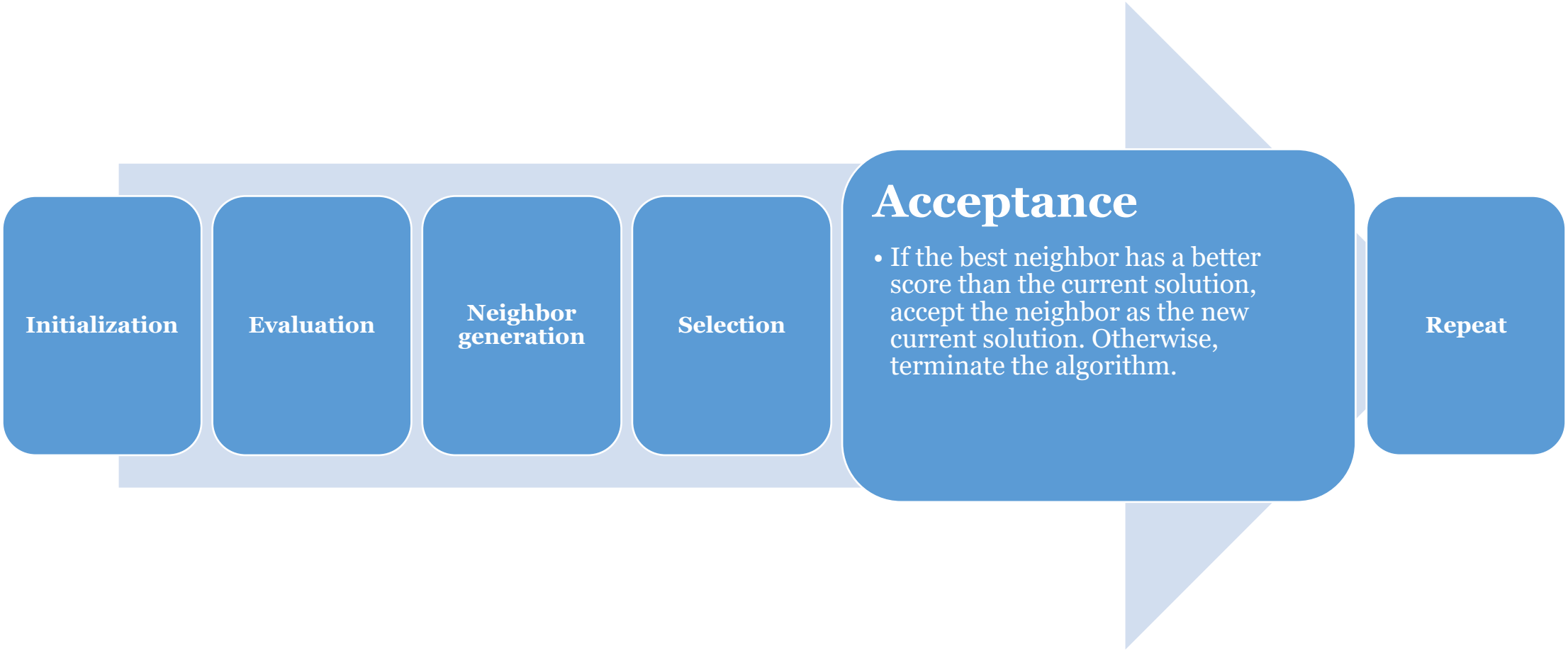


General outline of Hill Climbing Algorithm working

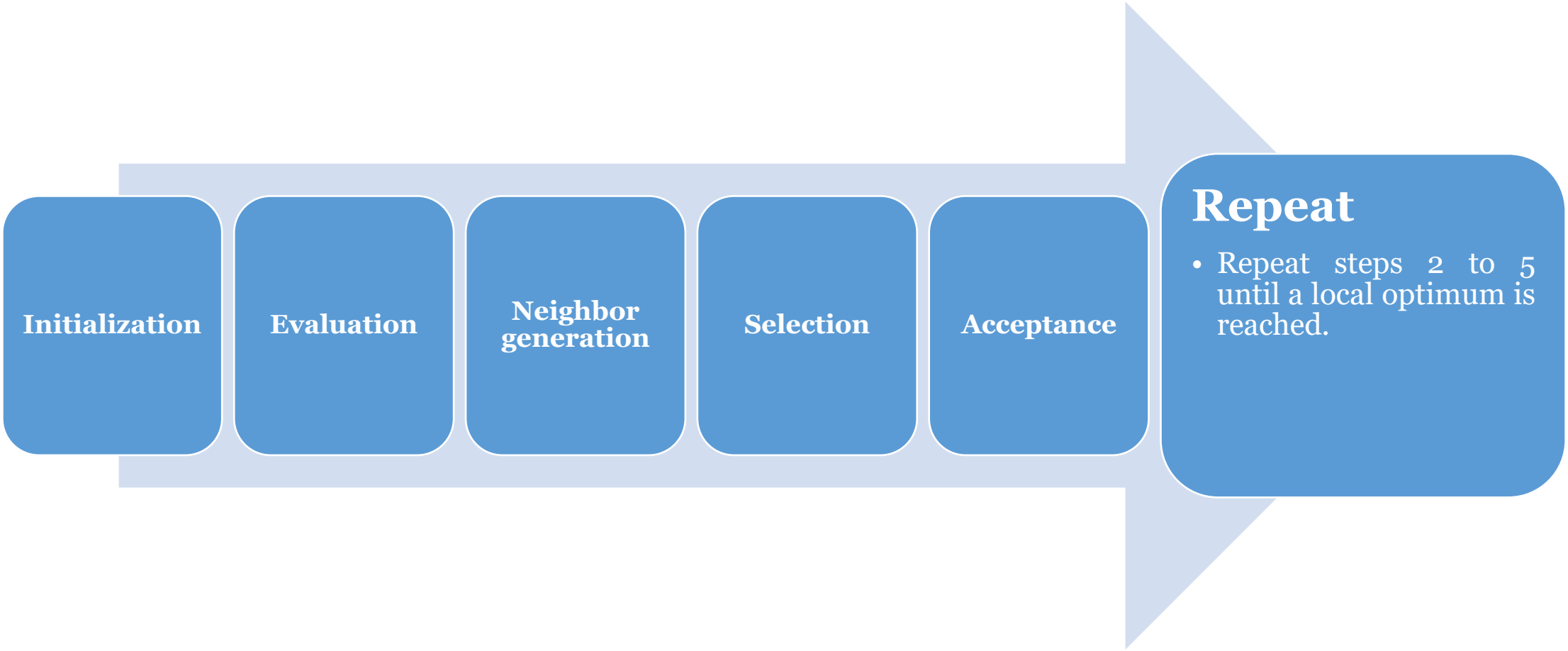


General outline of Hill Climbing Algorithm working

General outline of Hill Climbing Algorithm working



General outline of Hill Climbing Algorithm working



Example

- Let's consider an example of how the Hill Climbing Algorithm can be used to solve an optimization problem. Suppose we want to find the maximum value of the function f(x) = x^2 in the interval [0, 5]. We can use the Hill Climbing Algorithm to iteratively search for the best value of x that maximizes the function f(x).

Example

Step 1. Initialization: Choose an initial value of x randomly or using some heuristics. Let's choose x=2 as the initial value.

Step 2. Evaluation: Evaluate the value of f(x) at x=2. f(2) = 4.

Step 3. Neighbor generation: Generate a set of neighboring solutions by making small changes to the current solution. Let's generate two neighbors by adding and subtracting a small value, such as 0.1. The neighbors are x=2.1 and x=1.9.

Step 4. Selection: Select the best neighbor from the set of neighbors. Evaluate the values of f(x) at x=2.1 and x=1.9. f(2.1) = 4.41 and f(1.9) = 3.61. The best neighbor is x=2.1.

Example

Step 6. Acceptance: If the best neighbor has a better score than the current solution, accept the neighbor as the new current solution. In this case, the value of f(x) at x=2.1 is better than the value of f(x) at x=2, so we accept x=2.1 as the new current solution.

Step 7. Repeat: Repeat steps 2 to 5 until a local optimum is reached. Let's repeat the algorithm starting from x=2.1. The process continues until we reach the local optimum at x=5. The final solution is x=5, which maximizes the function f(x) in the interval [0, 5].

Types of optimization problems

➢**Function optimization**

- This type of problem involves finding the maximum or minimum value of a given function. The goal is to find the input value(s) that yield the best output value(s) of the function.
- The Hill Climbing Algorithm is well-suited for function optimization problems because it can iteratively search for the best solution in the search space.

➢**Application:** Finding the maximum profit of a business given a set of input variables such as production costs, sales price, and demand.

Types of optimization problems

➢**Constraint optimization**

- This type of problem involves finding the best solution that satisfies a set of constraints. The goal is to find the input value(s) that satisfy the constraints and optimize the objective function.
- The Hill Climbing Algorithm can be used for constraint optimization problems by incorporating the constraints into the neighbor generation and acceptance steps.

➢**Application:** Scheduling a set of tasks given constraints such as available resources, deadlines, and dependencies.

Types of optimization problems

➢ **Combinatorial optimization**

- This type of problem involves finding the best combination of discrete elements from a given set. The goal is to find the combination that maximizes or minimizes a given objective function. Combinatorial optimization problems are common in fields such as logistics, scheduling, and resource allocation.
- The Hill Climbing Algorithm can be used for combinatorial optimization problems by generating neighboring solutions that involve adding or removing elements from the current solution.

➢ **Application:** Packing a set of items into a container to maximize the total value, subject to constraints such as weight and volume limits.

Search strategies Used in Hill Climbing

➢**Random mutation:**

- A random perturbation is made to the current solution to generate a new solution. This approach can be effective for exploring a large search space, but it can also lead to suboptimal solutions if the perturbations are too large or not well-guided.

➢**Gradient descent:**

- The search process is guided by the gradient of the objective function, which indicates the direction of steepest increase or decrease. This approach can be effective for finding a local optimum, but it can also get stuck in local optima or saddle points.

Search strategies Used in Hill Climbing

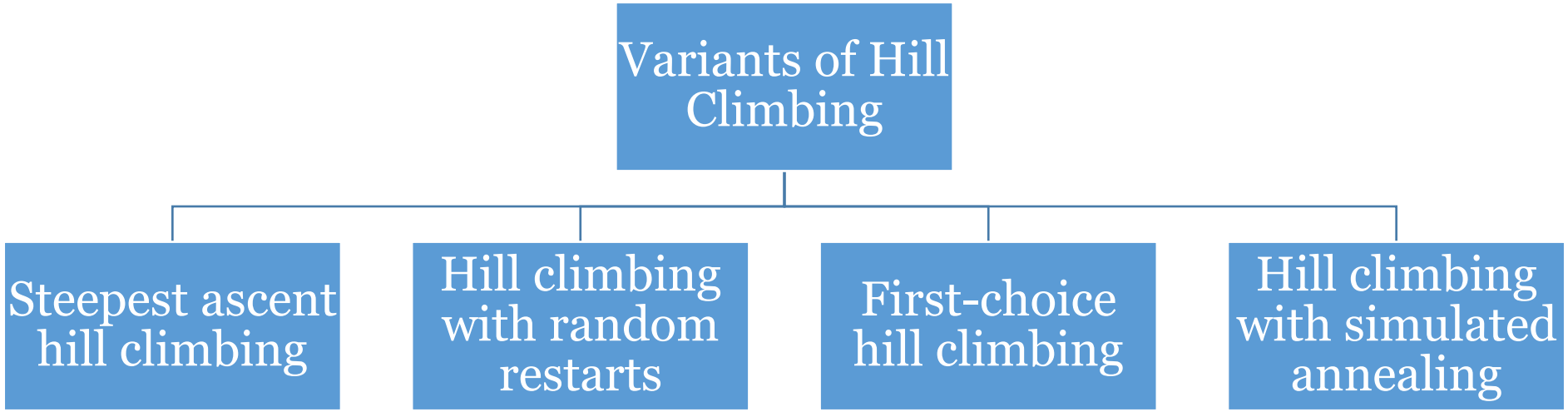
➢**Simulated annealing:**

- The search process is guided by a temperature parameter that controls the likelihood of accepting worse solutions. This approach can be effective for escaping local optima, but it can also be computationally expensive and sensitive to the choice of parameters.

➢**Tabu Search:**

- A tabu list is used to keep track of recently visited solutions and prevent backtracking. This approach can be effective for avoiding cycles and exploring diverse areas of the search space, but it can also get stuck in local optima if the tabu list is too strict.

Variants of Hill Climbing Algorithm



Steepest ascent hill climbing

- The search process always selects the neighbor with the highest objective function value, regardless of whether it is better than the current solution.
- This approach can be effective for finding the global optimum, but it can also be computationally expensive and prone to oscillations.
- **Application:** Finding the optimal parameters of a support vector machine for a binary classification problem.

Hill climbing with random restarts

- The search process is repeated from multiple random starting points, in the hope of finding a better solution.
- This approach can be effective for escaping local optima, but it can also be computationally expensive and require a large number of restarts.
- **Application:** Finding the optimal combination of hyperparameters for a deep learning model for image recognition.

First-choice hill climbing

- The search process evaluates a random subset of the neighbors and selects the first one that is better than the current solution.
- This approach can be effective for avoiding cycles and exploring diverse areas of the search space, but it can also lead to suboptimal solutions if the subset is too small or not well-guided.
- **Application:** Finding the optimal sequence of moves in a chess game.

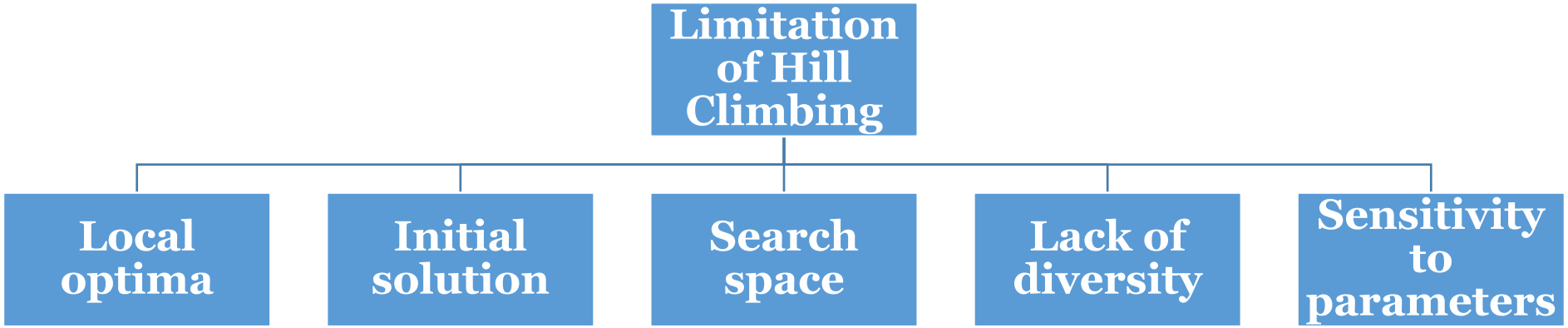
Hill climbing with simulated annealing

- The search process is guided by a temperature parameter that controls the likelihood of accepting worse solutions, similar to the simulated annealing search strategy.
- This approach can be effective for escaping local optima and exploring diverse areas of the search space, but it can also be computationally expensive and sensitive to the choice of parameters.
- **Application:** Finding the optimal placement of facilities in a logistics network with multiple objectives.

Advantages of Hill Climbing

- **Simplicity:** Hill climbing is easy to implement and understand, and requires few parameters to tune.
- **Efficiency:** Hill climbing can be computationally efficient, especially for problems with simple objective functions and low-dimensional search spaces.
- **Local search:** Hill climbing is a local search algorithm, which can be beneficial for problems where the objective function is smooth and the search space is well-behaved.

Limitations of Hill Climbing



Limitations of Hill Climbing

➢**Local optima:**

- Hill climbing can get stuck in local optima, which can lead to suboptimal solutions.
- Suboptimal solutions that are better than their immediate neighbors, but worse than the global optimum.
- This happens when the algorithm chooses a neighboring solution that improves the objective function, but does not lead to the global optimum.
- Once the algorithm is stuck in a local optimum, it cannot escape from it, and may not find the global optimum.
- This limitation can be particularly problematic for problems with many local optima and few global optima.

Limitations of Hill Climbing

- **Initial solution:**
- Hill climbing is sensitive to the initial starting point, which can affect the quality of the final solution.
- If the initial starting point is far from the global optimum, the algorithm may converge to a suboptimal solution.
- This limitation can be mitigated by using random or heuristic methods to generate a set of starting points, and selecting the best one as the initial solution.

Limitations of Hill Climbing

- **Search space:**
- Hill climbing is limited by the search space, which can be problematic for problems with complex or discontinuous objective functions.
- If the search space is large or contains regions with low objective function values, the algorithm may require a large number of iterations to find a good solution.
- Moreover, if the objective function is not smooth or continuous, the algorithm may not be able to accurately estimate the gradient, and may get stuck in regions with large gradients or noisy values.

Limitations of Hill Climbing

- **Lack of diversity:**
- The Algorithm focuses on improving the objective function value by iteratively selecting the best neighboring solution.
- This approach can be effective for problems where the objective function is smooth and well-behaved, but may not be appropriate for problems with multiple objectives or conflicting constraints.
- In these cases, the algorithm may converge to a single solution that is not diverse or does not reflect the trade-offs between different objectives or constraints.

Limitations of Hill Climbing

- **Sensitivity to parameters:**
- There are several parameters that can affect the performance, such as the step size, the stopping criterion, and the number of iterations.
- Choosing the right parameters can be challenging, and may require a trial-and-error process or a careful analysis of the problem characteristics.
- Moreover, the performance of the algorithm may be sensitive to small changes in the parameters, which can make it difficult to generalize to new problems or datasets.