**National University of Computer & Emerging Sciences, Karachi Computer Science Department**

**Summer 2023, Lab Manual – 05**

|  |  |
| --- | --- |
| **Course Code: AI-2002** | **Course: Artificial Intelligence Lab** |
| **Instructor(s):** | **Sania Urooj** |

**Contents:**

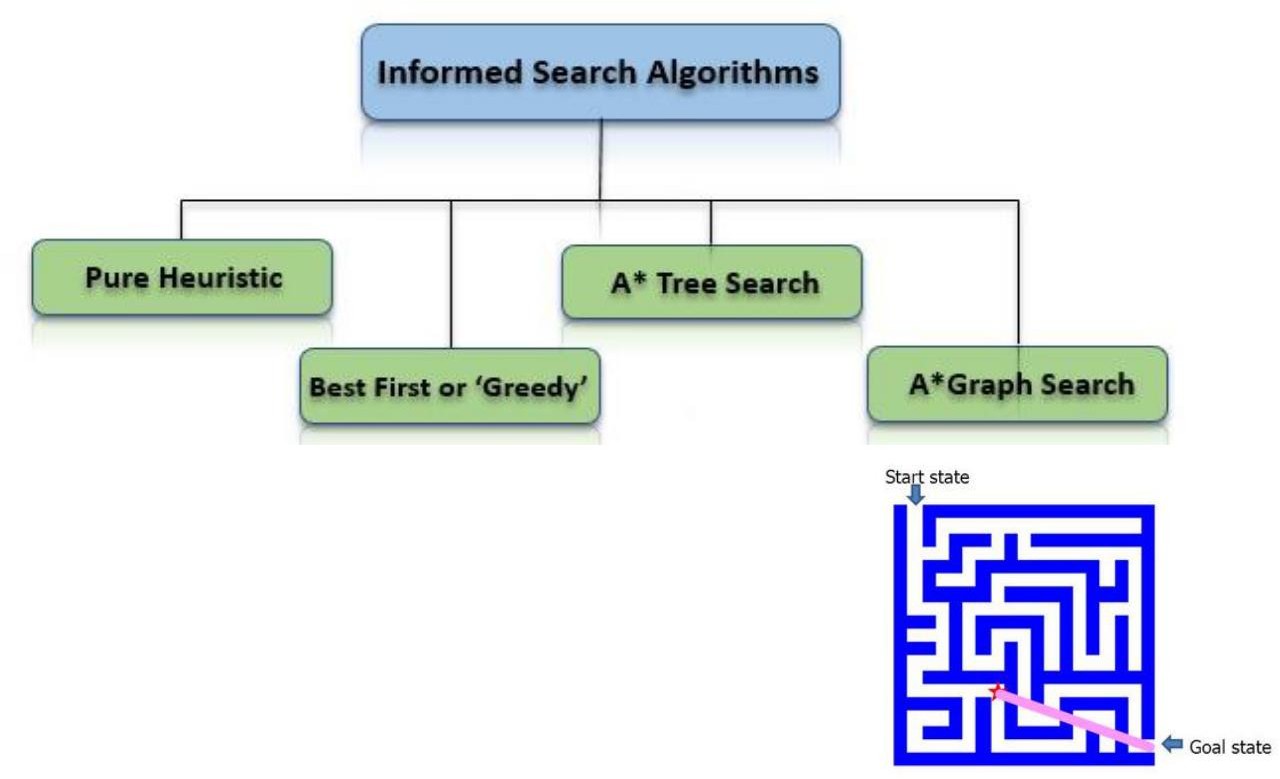
1. Informed or Heuristic Search Algorithm
   1. Pure Heuristic Search
   2. Best First Search
   3. A\* Search
2. Genetic Algorithm
3. Phases in GA
4. Limitations in GA
5. Applications of GA

# Heuristic (or informed) search algorithms:

* A solution cost estimation is used to guide the search.
* The optimal solution, or even a solution, are not guaranteed.

Some information about problem space (heuristic) is used to compute preference among the children for exploration and expansion.

To solve large problems with large number of possible states, problem-specific knowledge needs to be added to increase the efficiency of search algorithms.



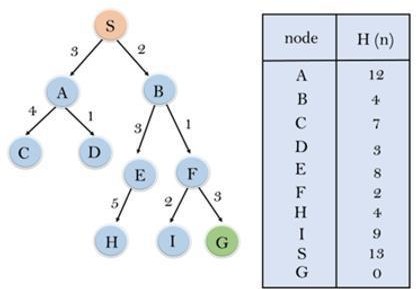
# Heuristic function:

Heuristic function h(n) estimates the cost of reaching goal from node n.

They calculate the cost of optimal path between two states.

Examples: Manhattan distance, Euclidean distance for path finding.

# Pure Heuristic Search

It expands nodes in the order of their heuristic values. It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes.

In each iteration, a node with a minimum heuristic value is expanded, all its child nodes are created and placed in the closed list. Then, the heuristic function is applied to the child nodes and they are placed in the open list according to their heuristic value. The shorter paths are saved and the longer ones are disposed.

# Best-First Search

If we consider searching as a form of traversal in a graph, an uninformed search algorithm would blindly traverse to the next node in a given manner without considering the cost associated with that step. An informed search, like Best first search, on the other hand would use an evaluation function to decide which among the various available nodes is the most promising (or ‘BEST’) before traversing to that node.

The Best first search uses the concept of a Priority queue and heuristic search. To search the graph space, the BFS method uses two lists for tracking the traversal. An ‘Open’ list which keeps track of the current ‘immediate’ nodes available for traversal and ‘CLOSED’ list that keeps track of the nodes already traversed.

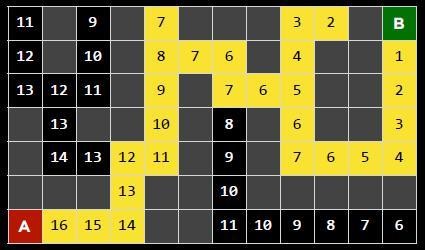
Variants of Best First Search Greedy best-first search A\* best-first search



# 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
   1. If OPEN list is empty, then EXIT the loop returning ‘False’
   2. Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also capture the information of the parent node
   3. 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
   4. 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
   5. Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n)

# Greedy Best first search algorithm:

A search method of selecting the best local choice at each step in hopes of finding an optimal solution.

It is the combination of depth-first search and breadth-first search algorithms.

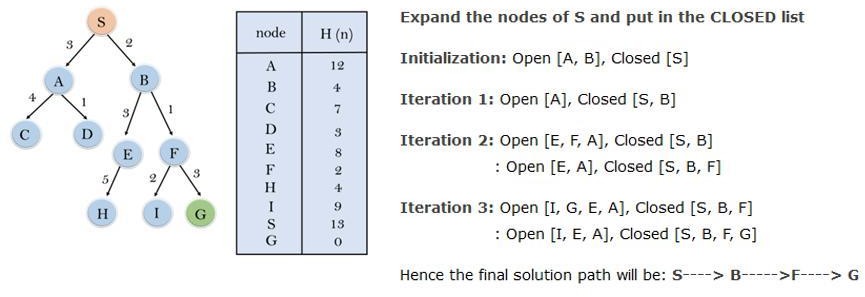
At each step, we choose the most promising node. In the greedy search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e. f(n)= h(n).

Evaluation function *f(n) = h(n)*

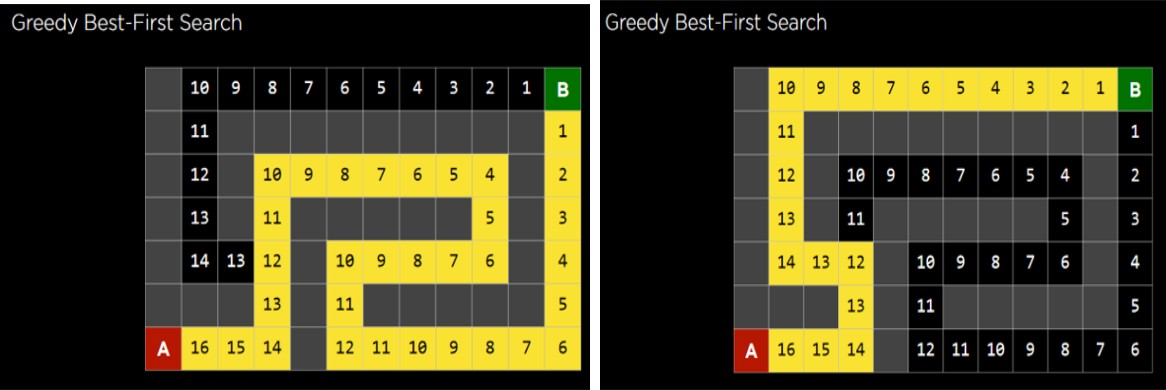


h(n) = estimated cost of the cheapest path from the state at node n to a goal state Greedy best-first search expands the node that appears to be closest to goal

It is implemented using priority queue.

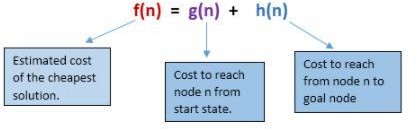


**Disadvantage** − It can get stuck in loops. It is not optimal.



# A\* search

It is best-known form of Best First search. It avoids expanding paths that are already expensive, but expands most promising paths first.

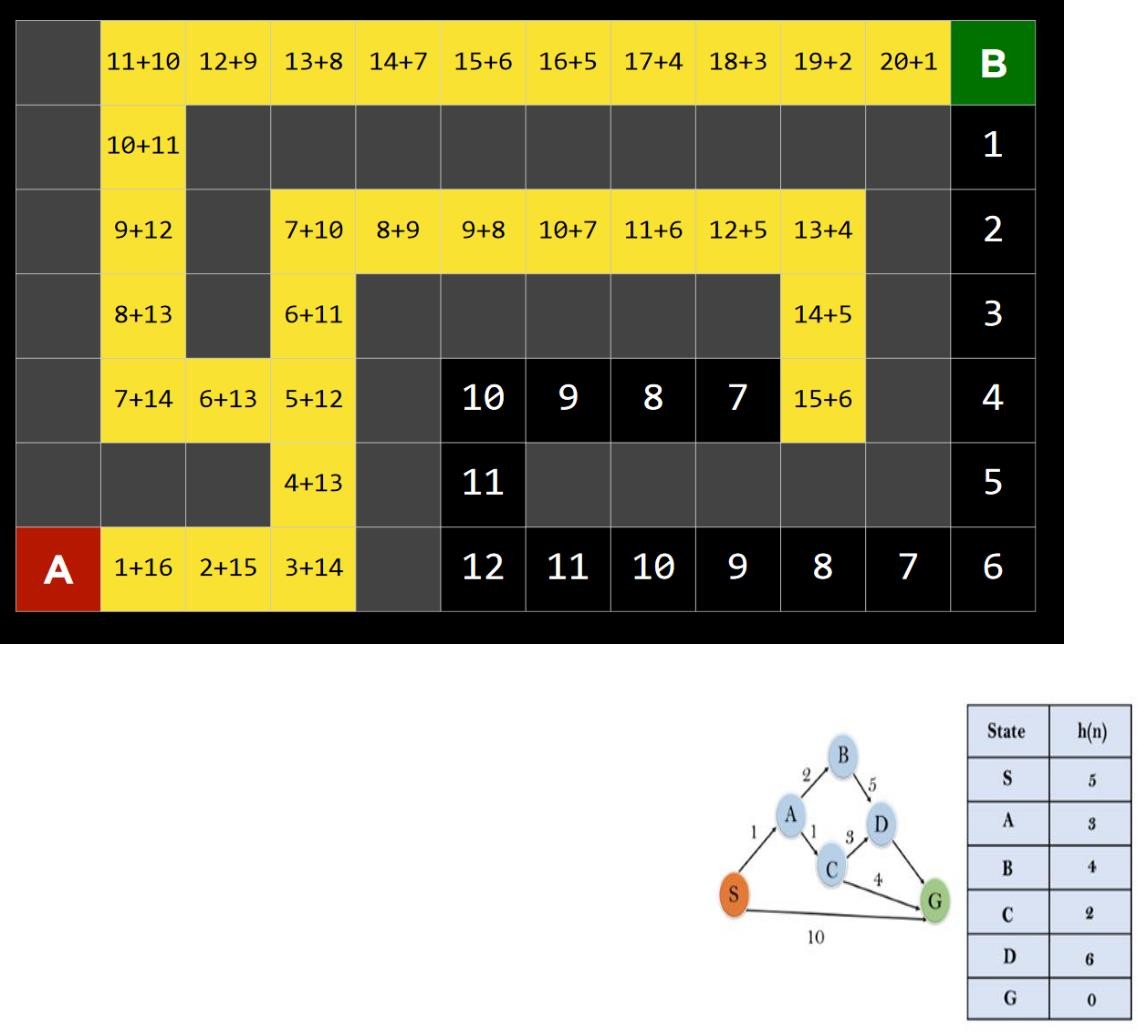


•Evaluation function f(n) = g(n) + h(n)

•g(n) = cost so far to reach n

•h(n)= estimated cost from n to goal

•f(n) = estimated total cost of path through n to goal

Initialization: {(S, 5)}

Iteration1: {(S --> A, 4), (S -->G, 10)}

Iteration2: {(S --> A-- >C, 4), (S--> A-- >B, 7), (S-- >G, 10)}

Iteration3: {(S --> A-- >C--- >G, 6), (S-- > A-->C--- >D, 11), (S--> A-->B, 7), (S-- >G, 10)}

Iteration 4 will give the final result, as S--->A--->C--->G it provides the optimal path with cost 6

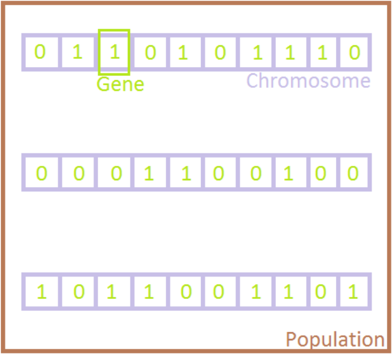
**Genetic Algorithm:**

A Genetic Algorithm (GA) is a meta-heuristic inspired by natural selection and is a part of the class of Evolutionary Algorithms (EA). We use these to generate high-quality solutions for optimization and search problems, for which, these use bio-inspired operators like mutation, crossover, and selection. In other words, using these, we hope to achieve optimal or near-optimal solutions to difficult problems.

This algorithm work in four steps:

* Individuals in population compete for resources, mate.
* Fittest individuals mate to create more off-springs than others.
* Fittest parent propagates genes through generation; parents may produce off- springs better than either parent.
* Each successive generation evolves to suit its ambience.

In optimization, we try to find within this search space the point or set of points that gives us the optimal solution. Each individual is like a string of characters/integers/floats and the strings are like chromosomes.



# Phases in Genetic Algorithms:

Five phases are considered in a genetic algorithm.

Initial population Fitness function Selection Crossover Mutation

# Example Of GA:

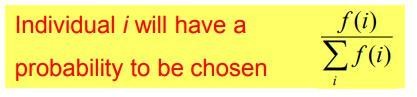
1. Initialization & Fitness

We toss a fair coin 10 times and get the following initial population:

|  |  |
| --- | --- |
| s1=1111010101 | (s1) = 7 |
| s2=0111000101 | (s2) = 5 |
| s3=1110110101 | (s3) = 7 |
| s4=0100010011 | (s4) = 4 |
| s5=1110111101 | (s5) = 8 |
| s6=0100110000 | (s6) = 3 |

Selection:

We randomly (using a biased coin) select a subset of the individuals based on their fitness.



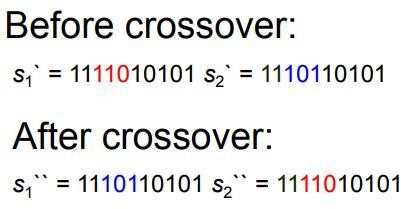
Suppose that, after performing selection, we get the following population: s1 ` = 1111010101 (s1)

s2 ` = 1110110101 (s3) s3 ` = 1110111101 (s5) s4 ` = 0111000101 (s2) s5 ` = 0100010011 (s4) s6 ` = 0100110000 (s6)

You can analyze here the fi values 7,7,8 for respective populations have the highest and nearest fitness function calculations we select the starting two pairs S1’ and S2’ for crossover process according to their fitness calculation.

1. Crossover:

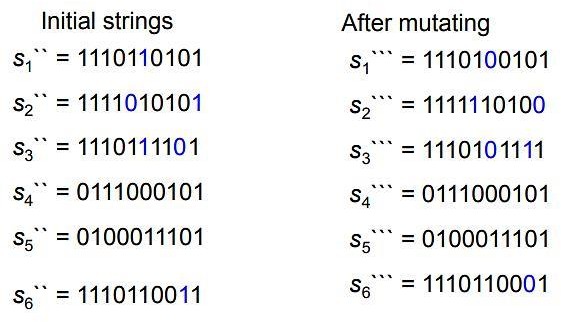
Next we mate strings for crossover. For each couple we first decide whether to actually perform the crossover or not. If we decide to actually perform crossover, we randomly extract the crossover points, for instance from point 2 and 5.



1. Mutation:

The final step is to apply random mutations: for each bit that we are to copy to the new population we allow a small probability of error (for instance 0.1).

Here, we also perform the crossover between the remaining pairs at a certain instance points.



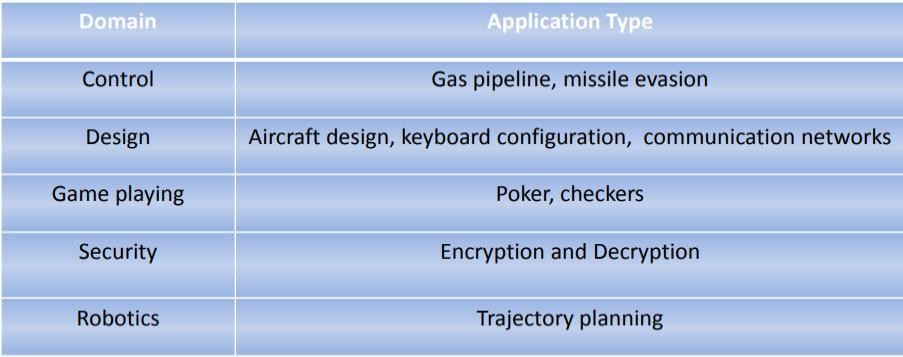
In one generation, the total population fitness changed from 34 to 37, thus improved by

~9%. At this point, we go through the same process all over again, until a stopping criterion is met.

# Benefits of Genetic Algorithms:

1. Concept is easy to understand.
2. Modular, separate from application.
3. Supports multi-objective optimization.
4. Always an answer; answer gets better with time.
5. Easy to exploit previous or alternate solutions.
6. Flexible building blocks for hybrid applications.

# GA Applications:



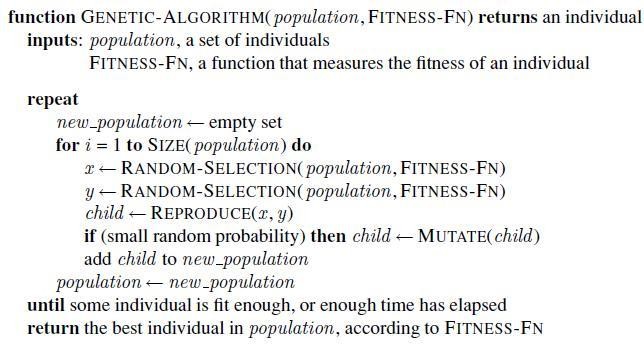
**Limitations of Genetic Algorithms**

Not suitable for simple problems with available derivative information Stochastic; no guarantee of the result solution being optimal



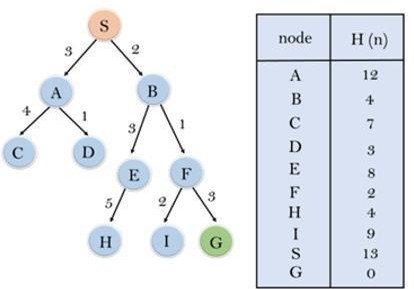
Frequent calculation of fitness value is computationally expensive for some problems No guarantee of convergence to the optimal solution if not implemented properly

# Pseudo Code of Genetic Algorithm:



**Example 1:**

Implement the following tree using greedy algorithm having a destination node H.



**Code**

class Node:

    def \_\_init\_\_(self, data):

        self.left = None

        self.right = None

        self.data = data

root = Node('S')

root.left = Node('A')

root.right = Node('B')

A = root.left

A.left = Node('C')

A.right = Node('D')

B = root.right

B.left = Node('E')

B.right = Node('F')

B.left.left = Node('H')

F = B.right

F.left = Node('I')

F.right = Node('G')

Actual\_Cost = {'S':0,'A':3,'B':2,'C':4,'D':1,'E':3,'F':9,'G':3,'H':5,'I':2}

Heuristic = {'S':13,'A':12,'B':4,'C':7,'D':3,'E':8,'F':2,'G':0,'H':4,'I':9}

Goal = 'G'

def run():

    opened = []

    closed = []

    visited = []

    temp = []

    visited.append("S")

    node = root

    closed.append(node.data)

    while node.data != Goal:

        if node.left and node.left.data not in visited:

            opened.append(node.left)

            visited.append(node.left.data)

            temp.append((Heuristic[node.left.data], node.left.data))

        if node.right and node.right.data not in visited:

            opened.append(node.right)

            visited.append(node.right.data)

            temp.append((Heuristic[node.right.data], node.right.data))

        temp.sort(reverse=True)

        x = temp.pop()

        y=0

        for i in opened:

            if i.data==x[1]:

                node=i

                closed.append(i.data)

                break

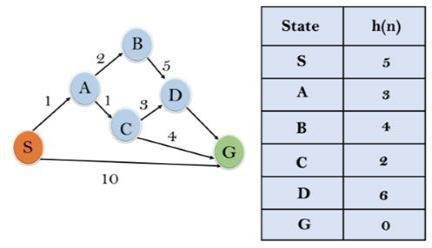
            y = y + 1

        opened.pop(y)

    return closed

print(run())

**Example 2:**

Implement the following graph using A\* search algorithm starting from Node S to Node G

**Code:**

import heapq

import heapq

class Graph:

def \_\_init\_\_(self):

self.adjacency\_list = {

'S': {'A': 1, 'G':10},

'A': {'B': 2,'C':1},

'B': {'D': 5},

'C': {'D': 3,'G':4},

'D': {'G': 4}

}

def heuristic(self, node):

# Define the heuristic function for estimating the cost from a node to the goal (Node D)

heuristic\_values = {'S': 5, 'A': 3, 'B': 4, 'C': 2, 'D': 6, 'G': 0}

return heuristic\_values[node]

def a\_star\_search(self, start, goal):

# Initialize the priority queue with the start node and its total cost

priority\_queue = [(0, start)]

# Initialize the dictionary to store the cost of reaching each node

cost = {start: 0}

# Initialize the dictionary to store the parent node for each node in the path

parent = {start: None}

while priority\_queue:

\_, current\_node = heapq.heappop(priority\_queue)

if current\_node == goal:

# Reconstruct and return the path

path = self.construct\_path(parent, goal)

return path

for neighbor, edge\_cost in self.adjacency\_list[current\_node].items():

new\_cost = cost[current\_node] + edge\_cost

if neighbor not in cost or new\_cost < cost[neighbor]:

cost[neighbor] = new\_cost

total\_cost = new\_cost + self.heuristic(neighbor)

heapq.heappush(priority\_queue, (total\_cost, neighbor))

parent[neighbor] = current\_node

# No path found from the start to the goal

return []

def construct\_path(self, parent, goal):

path = []

current = goal

while current:

path.append(current)

current = parent[current]

path.reverse()

return path

# Example usage

graph = Graph()

start\_node = 'S'

goal\_node = 'G'

path = graph.a\_star\_search(start\_node, goal\_node)

print("Shortest path:", path)