National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Spring 2023, Lab Manual – 05

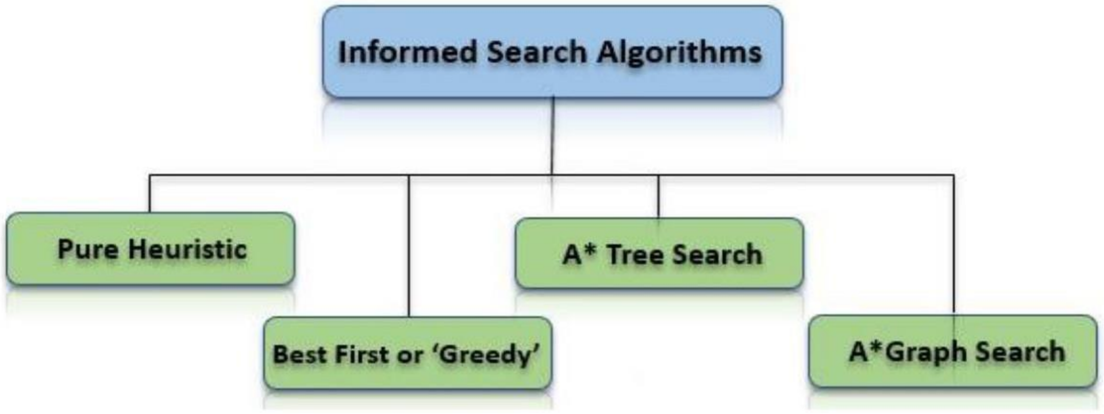
Course Code: AI-2002	**Course: Artificial Intelligence Lab**
Instructor(s):	**Kariz Kamal, Saeeda Kanwal, Sania Urooj, Shakir Hussain, Omer Qureshi, Muhammad Ali Fatmi**

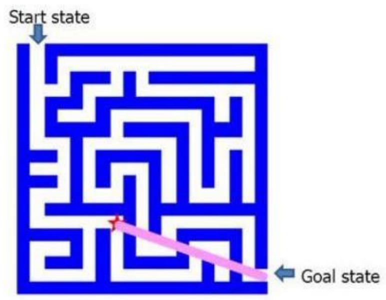
Contents:

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.





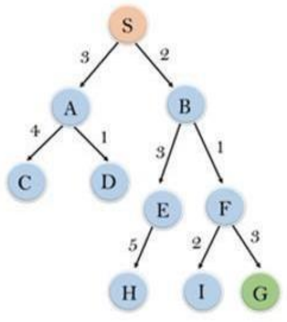
Heuristic function:

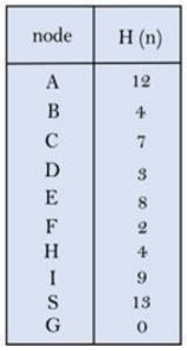
Heuristic function h(n) estimates the cost of reaching goal from node n.
They calculate the cost of optimal path between two states.

- 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.





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

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

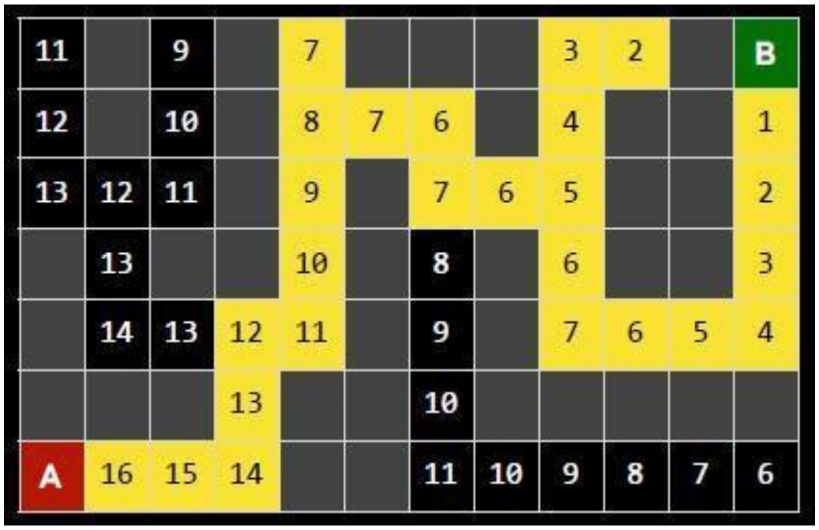
Best first search algorithm:

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

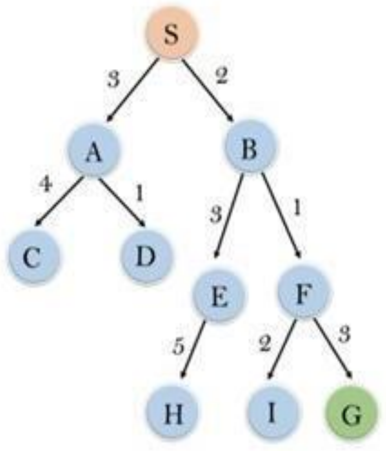
3. If N is a GOAL node, then move the node to the Closed list and exit the loop returning 'True'. The solution can be found by backtracking the path
4. If N is not the GOAL node, expand node N to generate the 'immediate' next nodes linked to node N and add all those to the OPEN list
5. Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n)

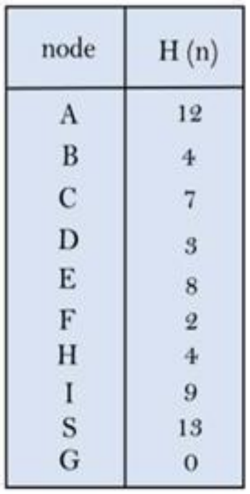
Greedy Best first search algorithm:

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).

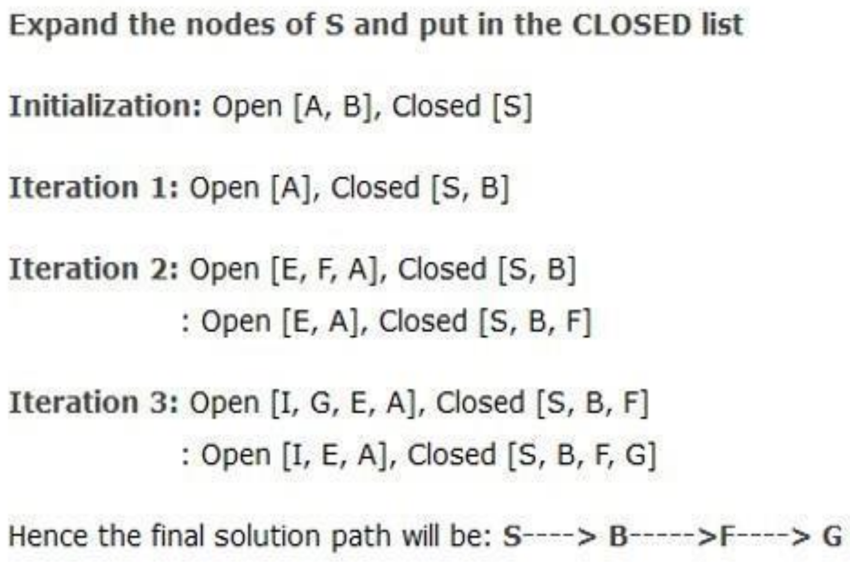


- Evaluation function *f(n) = h(n)*
- h(n) = estimated cost of the cheapest path from the state at node n to a goal state
- Greedy best-first search expands the node that appears to be closest to goal
- It is implemented using priority queue.





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



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

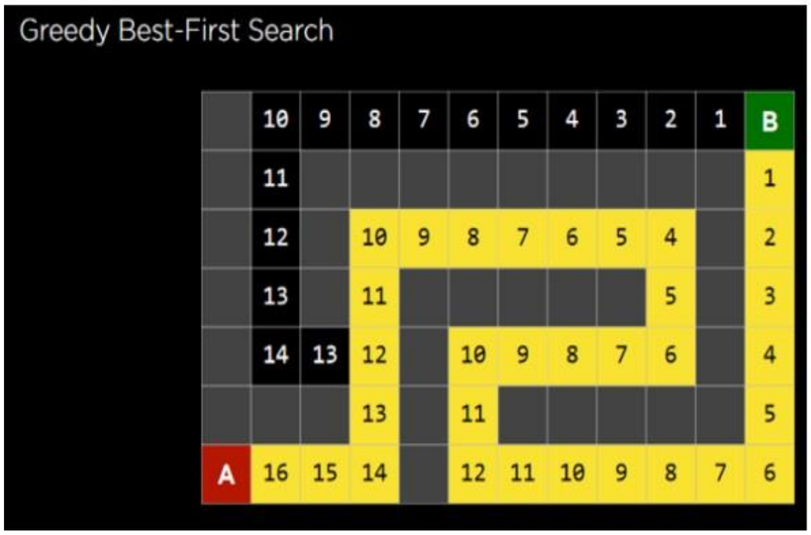
Iteration 1: Open [A], Closed [S, B]

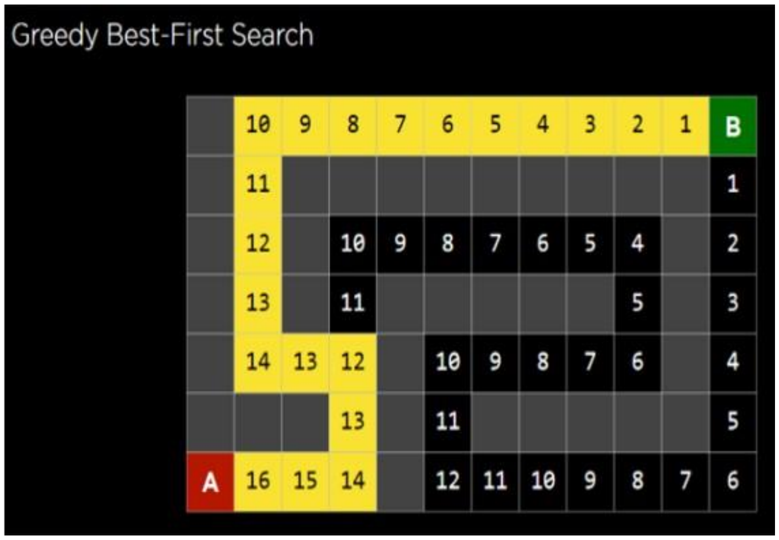
Iteration 2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]

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

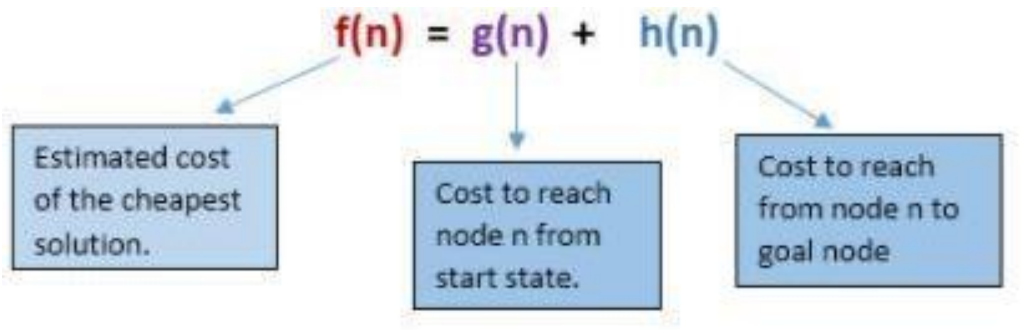
Disadvantage − It can get stuck in loops. It is not optimal.



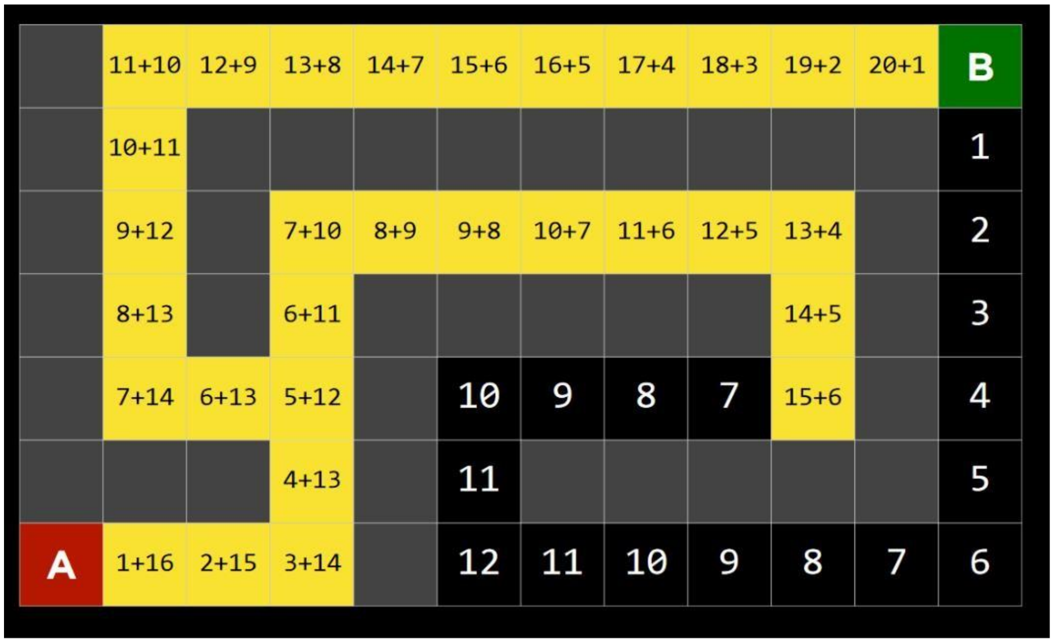


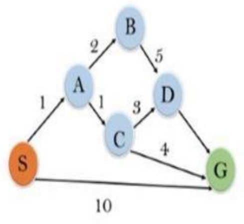
A* search

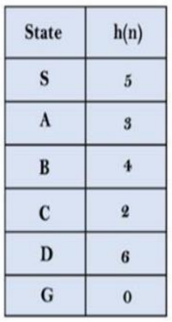
It is best-known form of Best First search. It avoids expanding paths that are already expensive, but expands most promising paths first.


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

•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







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

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

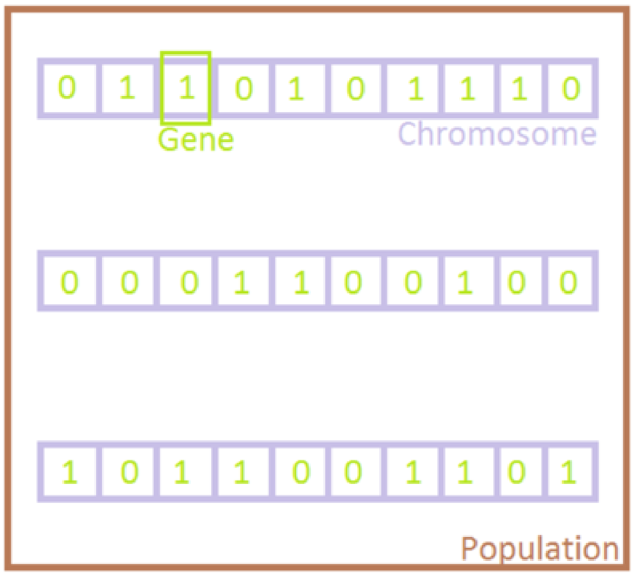
Genetic Algorithm:

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

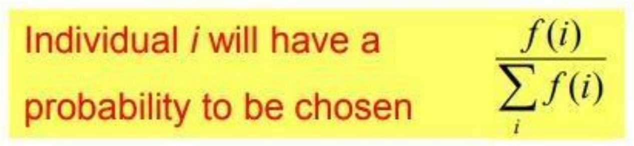
s4=0100010011(s4) = 4

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.



Individual *i* will have a probability to be chosen $\frac{f(i)}{\sum_i f(i)}$

Suppose that, after performing selection, we get the following population:

s1 ` = 1111010101 (s1)

s2 ` = 1110110101 (s3)

s3 ` = 1110111101 (s5)

s4 ` = 0111000101 (s2)

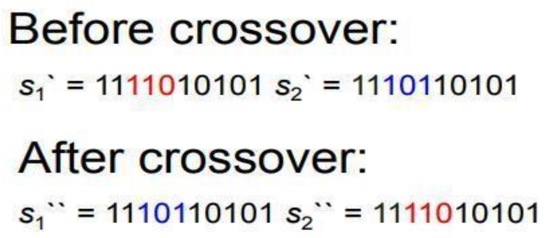
s5 ` = 0100010011 (s4)

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.

3. 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.



Before crossover:

s_1` = 1111010101 $s_2$` = 1110110101

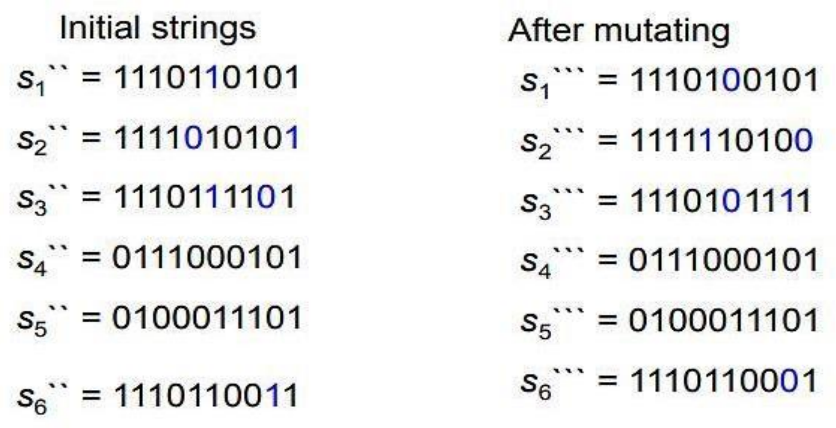
After crossover:

s_1`` = 1110110101 $s_2$`` = 1111010101

4. 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.



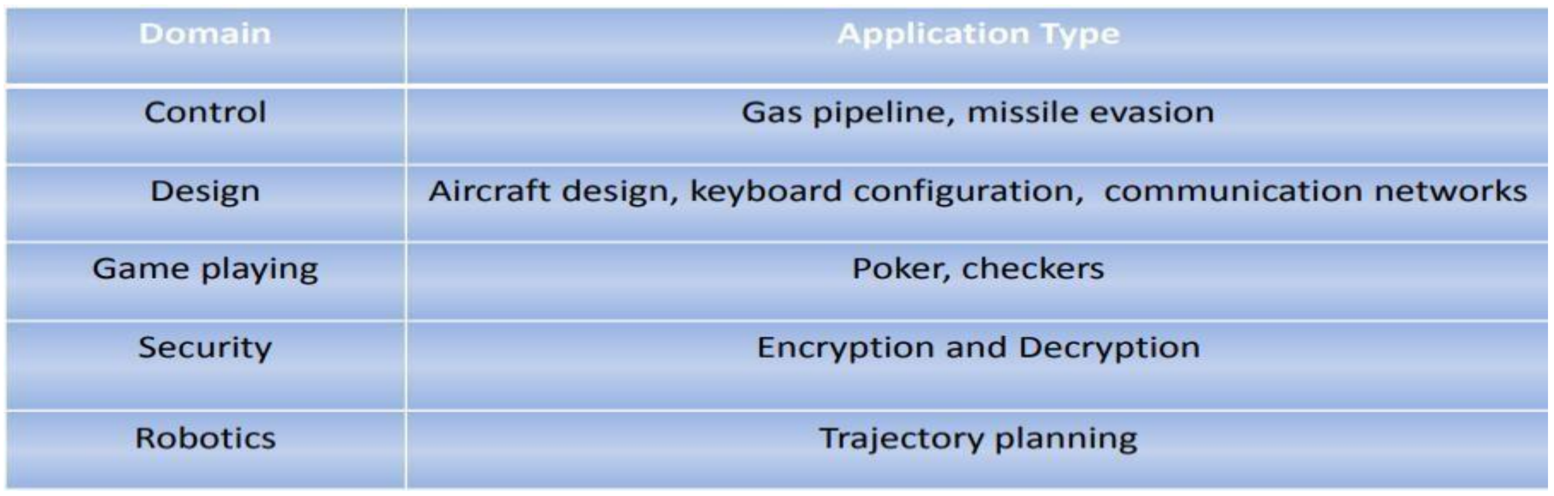
Initial strings	After mutating
s_1`` = 1110110101	s_1``` = 1110100101
s_2`` = 1111010101	s_2``` = 1111110100
s_3`` = 1110111101	s_3``` = 1110101111
s_4`` = 0111000101	s_4``` = 0111000101
s_5`` = 0100011101	s_5``` = 0100011101
s_6`` = 1110110011	s_6``` = 1110110001

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:

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

GA Applications:

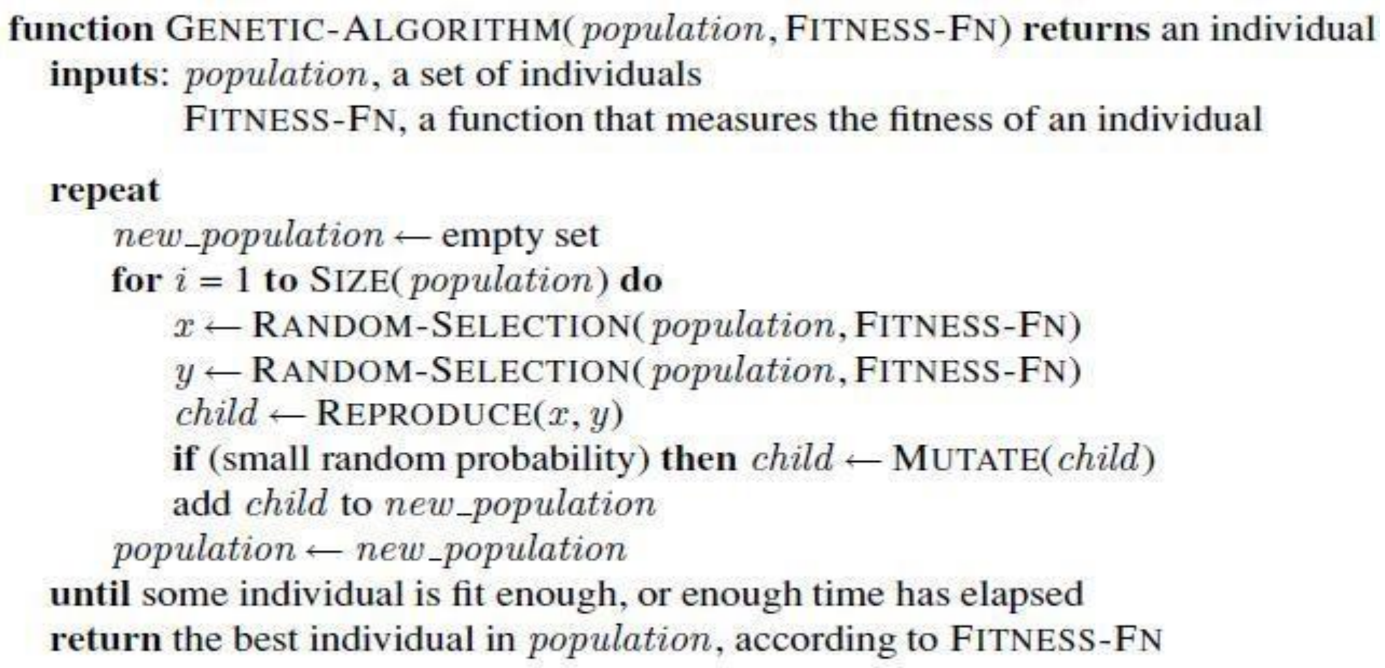


Domain	Application Type
Control	Gas pipeline, missile evasion
Design	Aircraft design, keyboard configuration, communication networks
Game playing	Poker, checkers
Security	Encryption and Decryption
Robotics	Trajectory planning

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

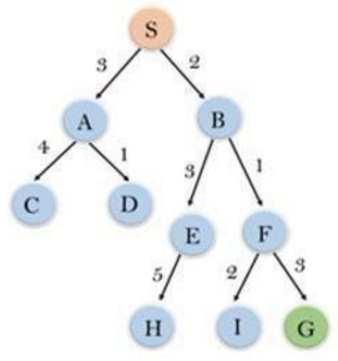
Pseudo Code of Genetic Algorithm:

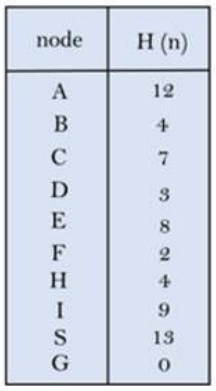


```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for $i$ = 1 to SIZE(population) do
        $x$ ← RANDOM-SELECTION(population, FITNESS-FN)
        $y$ ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE($x$, $y$)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

Example 1:

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





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

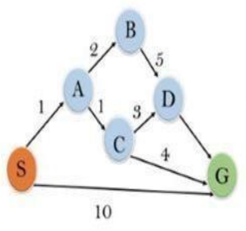
Code

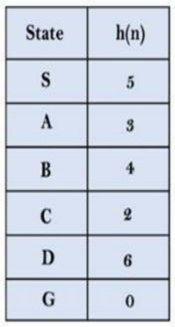
```
import queue
from queue import PriorityQueue

class Node:
    def __init__(self, data,value,value2):

        self.left = None
        self.right = None
        self.data = data
        self.hn = value
        self.gn = value2

    def PrintTree(self):
        print( self.data)
        if self.left:
            self.left.PrintTree()
        if self.right:
            self.right.PrintTree()

    def insert(p,root):
        p.put(self.data,rself.value)
        if self.left:
            self.left.PrintTree()
        if self.right:
            self.right.PrintTree()
    def check(self):
        found = False
        p = PriorityQueue()
        p.put((self.hn,self))
        while not found:
            k = p.get()
            n = k[1]
            print("\nSelected node : " + n.data + "\n")
            if( n.data  == 'H' ):
```

```
            print("Found")
            found=True
            return True;
        if n.left:
            print("at " + n.left.data + " ,h(n) : "+ str(n.left.hn))
            p.put((n.left.hn,n.left))
        if n.right:
            print("at " + n.right. data + " ,h(n) : "+ str(n.right.hn))
            p.put((n.right.hn,n.right))


root = Node('S',13,0)
root.left = Node('A',12,3)
root.right = Node('B',4,2)
root.left.left = Node('C',7,4)
root.left.right = Node('D',3,1)
root.right.left = Node('E',8,3)
root.right.right = Node('F',2,1)
root.right.left.left = Node('H',4,5)
root.right.right.left = Node('I',9,2)
root.right.right.right = Node('G',0,3)
```

Example 2:

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





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

Code:

```
from collections import deque

class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    def h(self, n):
        H = {
            'S': 0,
            'A': 5,
            'B': 4,
            'C': 2,
            'D': 6,
            'G': 0
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
        closed_list = set([])

        g = {}

        g[start_node] = 0

        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path
                for (m, weight) in self.get_neighbors(n):

                    if m not in open_list and m not in closed_list:
                        open_list.add(m)
                        parents[m] = n
                        g[m] = g[n] + weight

                    else:
                        if g[m] > g[n] + weight:
                            g[m] = g[n] + weight
                            parents[m] = n

                            if m in closed_list:
                                closed_list.remove(m)
                                open_list.add(m)

                open_list.remove(n)
                closed_list.add(n)

            print('Path does not exist!')
            return None

    adjacency_list = {
        'S': [('A', 1), ('G', 10),],
        'A': [('B', 2),('C', 1)],
        'B': [('D', 5)],
        'C': [('D', 5), ('G', 4)],
        'D': [('G', 6)],
        'G': [('G', 0)]
    }


    graph1 = Graph(adjacency_list)
    graph1.a_star_algorithm('S', 'G')
```