CS324
Intelligent Systems
(Elective)

S.Y.

2020 – 2021

By:

Christian Q. Panuncillo

BSCS - 4

Table of Contents

Breadth-first search

Breadth-first search (BFS) is a traversing algorithm which starts from a selected node (source or starting node) and explores all the neighbor nodes at the present depth before moving on to the nodes at the next level of depth. It must be ensured that each vertex of the graph is visited exactly once to avoid getting into an infinite loop with cyclic graphs or to prevent visiting a given node multiple times when it can be reached through more than one path.

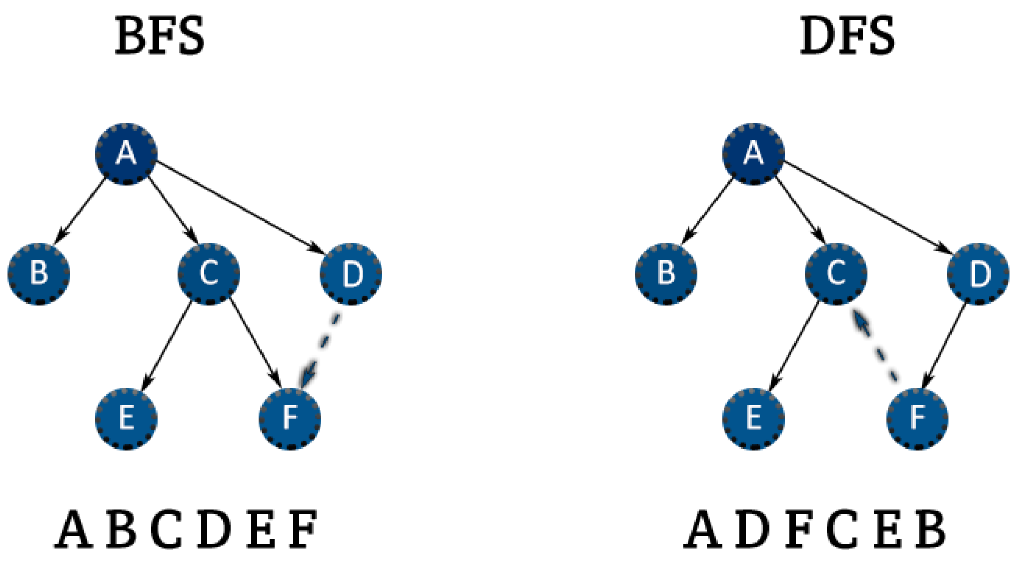
Breadth-first search can be implemented using a queue data structure, which follows the first-in-first-out (FIFO) method – i.e., the node that was inserted first will be visited first, and so on.

Algorithm:

We start the process by considering any random node as the starting vertex. We enqueue (insert) it to the queue and mark it as visited. Then we mark and enqueue all its unvisited neighbors at the current depth or continue to the next depth level if there is any. The visited vertices are removed from the queue. The process ends when the queue becomes empty.

Example of applications:

- Check if a graph is connected.
- Generating spanning tree
- Finding the shortest path in a graph



Depth-first search

Depth-first search (DFS) is a traversing algorithm that uses the opposite strategy of breadth-first search. It explores the highest-depth nodes first before backtracking and expanding shallower nodes.

Depth-first search can be implemented using a stack data structure, which follows the last-in-first-out (LIFO) method – i.e., the node that was inserted last will be visited first.

Algorithm:

First, we select any random node as a starting vertex. We push it onto the stack and mark it as visited. Then we explore it as far as possible in a branch and push its unvisited neighbor nodes onto the stack. If there are not any unvisited nodes left, we come back to a fixed point (backtrack). The visited vertices are pushed onto the stack and later when there is no vertex further to visit those are popped-off. The process ends when the stack becomes empty.

Examples of application:

- Testing connectivity
- Finding a path between two nodes
- Solving puzzles

Key Differences

BFS visits all the neighbors before visiting children nodes. DFS visits all children nodes before visiting neighbors.

For implementation, BFS uses a queue data structure, while DFS uses a stack.

BFS uses a larger amount of memory because it expands all children of a vertex and keeps them in memory. It stores the pointers to a level's child nodes while searching each level to remember where it should go when it reaches a leaf node. DFS has much lower memory requirements than BFS because it iteratively expands only one child until it cannot proceed anymore and backtracks thereafter. It has to remember a single path with unexplored nodes.

Both have the same O(n) time complexity, but BFS is "better" at finding the shortest path in a graph, while DFS is "better" at answering connectivity queries (determining if every pair of vertices can be connected by two disjoint paths).

Choosing between both algorithms depends on the structure of the search tree and the number and location of the solutions. If the solution is mostly in the neighborhood of the current node it is better to choose BFS. If it is most likely the farthest descendant of a node then choosing a DFS is a better option. If the graph is dense and your average number of neighbors is high relative to the number of nodes, a breadth-first search will have long queues while depth-first will have small stacks. In sparse graphs, the situation is reversed. In conclusion, if memory is a limiting factor, the shape of the graph may have to inform your choice of a search strategy.

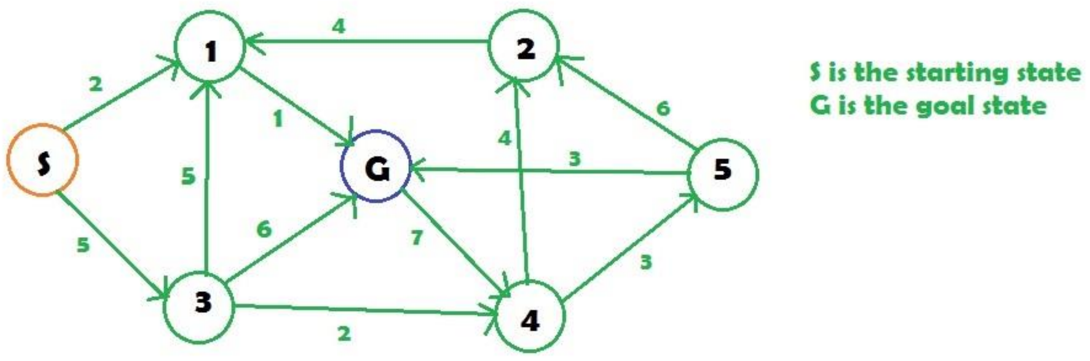
Uniform-Cost Search (Dijkstra for large Graphs)

Uniform-Cost Search is a variant of Dijikstra's algorithm. Here, instead of inserting all vertices into a priority queue, we insert only source, then one by one insert when needed. In every step, we check if the item is already in priority queue (using visited array). If yes, we perform decrease key, else we insert it.

This variant of Dijsktra is useful for infinite graphs and those graphs which are too large to represent in the memory. Uniform-Cost Search is mainly used in Artificial Intelligence.

Examples:

Input:



Output:

Minimum cost from S to G is =3

Recommended:

Please try your approach on {IDE} first, before moving on to the solution.

Uniform-Cost Search is similar to Dijikstra's algorithm. In this algorithm from the starting state we will visit the adjacent states and will choose the least costly state then we will choose the next least costly state from the all un-visited and adjacent states of the visited states, in this way we will try to reach the goal state (note we won't continue the path through a goal state), even if we reach the goal state we will continue searching for other possible paths(if there are multiple goals) . We will keep a priority queue which will give the least costliest next state from all the adjacent states of visited states.

Best-first Search Algorithm (Greedy Search):

Greedy 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. It uses the heuristic function and search. Best-first search allows us to take the advantages of both 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.

$f(n)= g(n)$

Were, $h(n)$= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm:

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both lists, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

Advantages:

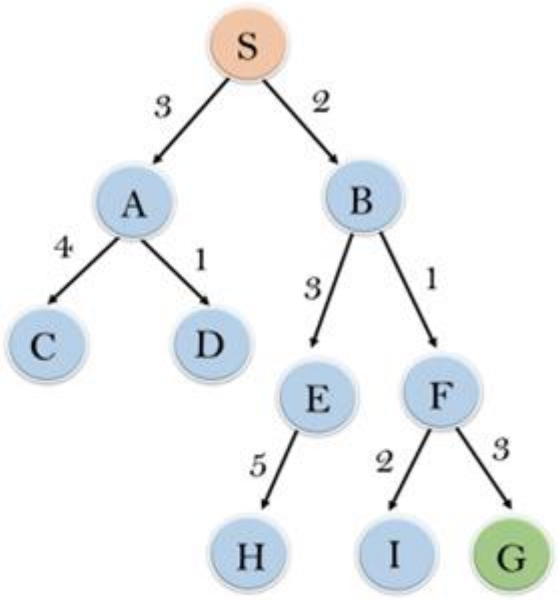
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

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.

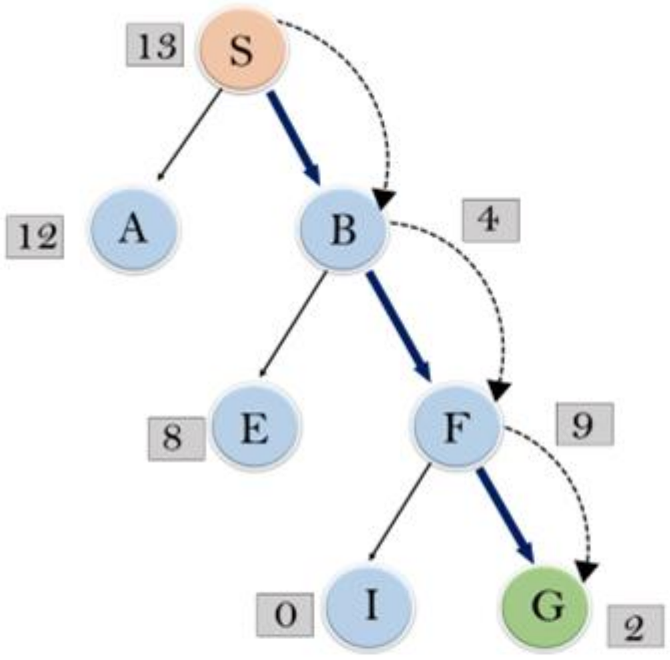
Example:

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n), which is given in the below table.



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

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.



- **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]

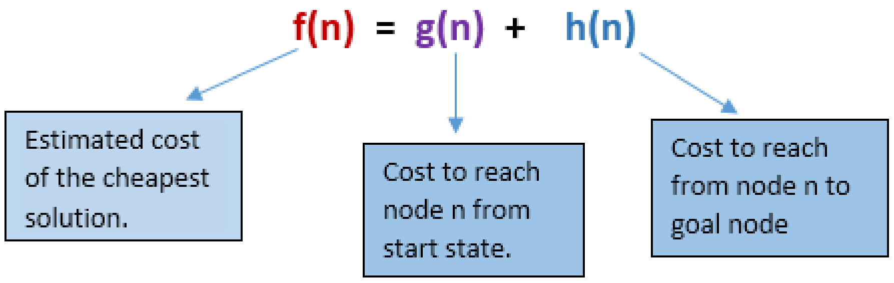
  : Open [I, E, A], Closed [S, B, F, G]

- Hence the final solution path will be: S----> B----->F----> G.
- **Time Complexity:** The worst-case time complexity of Greedy best first search is O(b^m).
- **Space Complexity:** The worst-case space complexity of Greedy best first search is O(b^m). 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* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solves the problem efficiently. 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. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

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**.


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

At each point in the search space, only those nodes are expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

Algorithm of A* search:

- **Step1:** Place the starting node in the OPEN list.
- **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise.
- **Step 4:** Expand node n and generate all of its successors and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
- **Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- **Step 6:** Return to **Step 2**.

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.

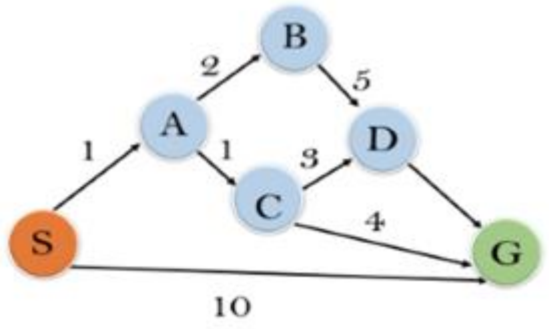
Disadvantages:

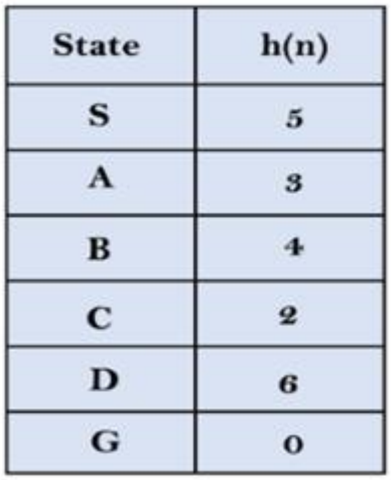
- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example:

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

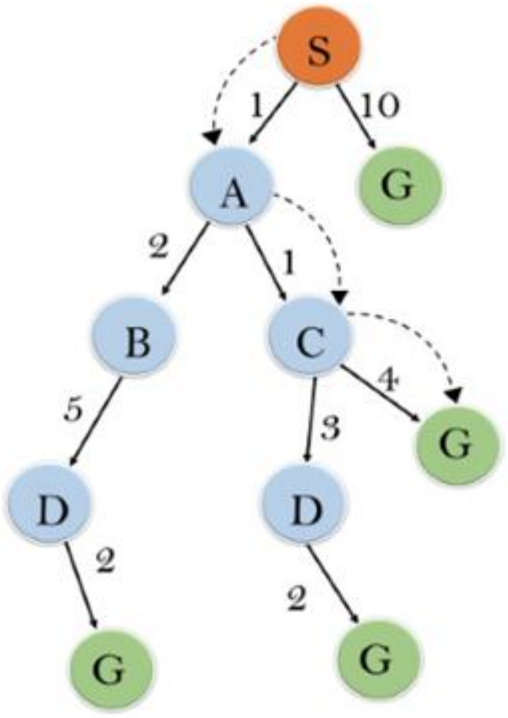
Here we will use OPEN and CLOSED list.





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

Solution:



- **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.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n) <="" li="">

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- Admissible: the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- Consistency: Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

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)

Constraint satisfaction problem

Constraint satisfaction problems (CSPs) are mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research since the regularity in their formulation provides a common basis to analyze and solve problems of many seemingly unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. Constraint Programming (CP) is the field of research that specifically focuses on tackling these kinds of problems. Additionally, boolean satisfiability problem (SAT), the satisfiability modulo theories (SMT), mixed integer programming (MIP) and answer set programming (ASP) are all fields of research focusing on the resolution of particular forms of the constraint satisfaction problem.

Example: Problems that can be modeled as a constraint satisfaction problem include:

- Type inference
- Eight queens puzzle
- Map coloring problem
- Sudoku, Crosswords, Futoshiki, Kakuro (Cross Sums), Numbrix, Hidato and many other logic puzzles

These are often provided with tutorials of CP, ASP, Boolean SAT and SMT solvers. In the general case, constraint problems can be much harder, and may not be expressible in some of these simpler systems. "Real life" examples include automated planning, lexical disambiguation, musicology, product configuration, and resource allocation.

The existence of a solution to a CSP can be viewed as a decision problem. This can be decided by finding a solution or failing to find a solution after exhaustive search (stochastic algorithms typically never reach an exhaustive conclusion, while directed searches often do, on sufficiently small problems). In some cases, the CSP might be known to have solutions beforehand, through some other mathematical inference process.

Genetic Algorithms

Genetic algorithms use an iterative process to arrive at the best solution. Finding the best solution out of multiple best solutions (best of best). Compared with Natural selection, it is natural for the fittest to survive in comparison with others.

Now let us try to grab some pointers from the evolution side to clearly correlate with genetic algorithms.

- Evolution usually starts from a population of randomly generated individuals in the form of iteration. (Iteration will lead to a new generation).
- In every iteration or generation, the fitness of each individual is determined to select the fittest.
- Genome fittest individuals selected are mutated or altered to form a new generation, and the process continues until the best solution has reached.

The process terminates under 2 scenarios-

1. When maximum number of generations have been created
2. Fitness level reached is sufficient.

Relating it to the Optimization scenario, we need to identify the **Genetic Representation** of our solution domain or business problem we need to solve. Evaluation criteria i.e., **Fitness Function** to decide the worth of a solution.

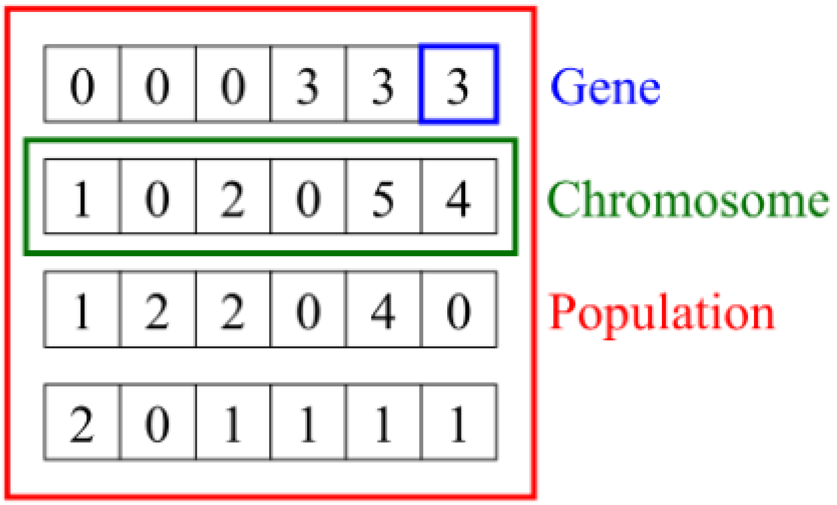
For example:

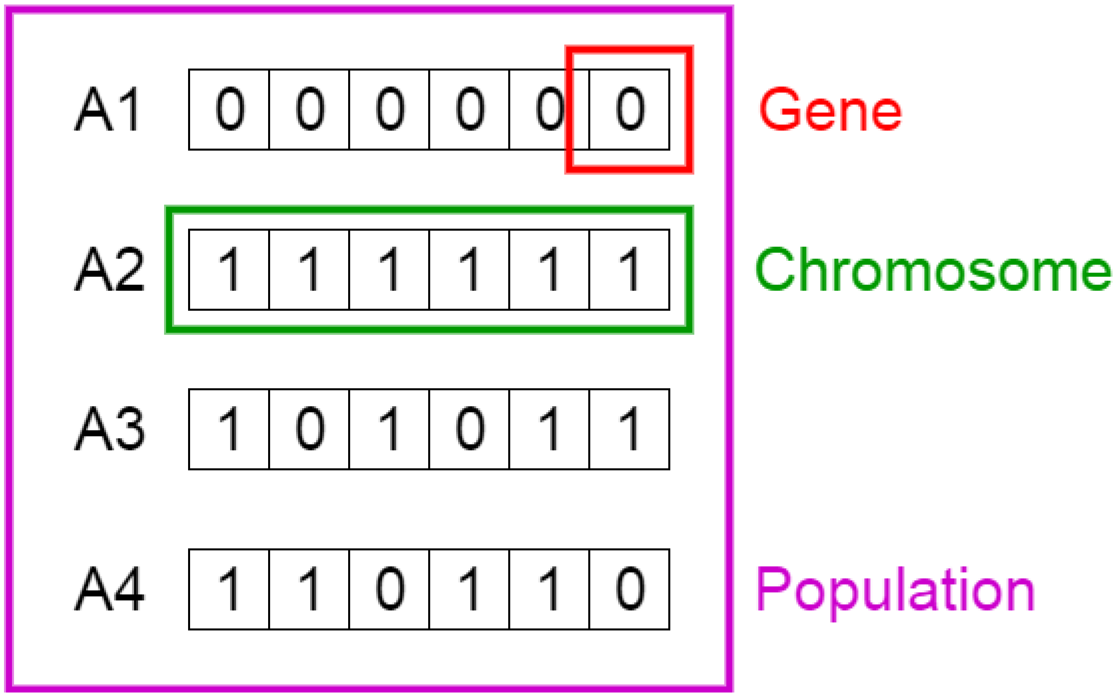
- We need to maximize the profit (Fitness Function) by increasing sales (Genetic representation) of the product.
- We need to find the best model hyperparameters (Fitness function) for the classification algorithms i.e., Fine-tuning to yield the best prediction.
- Optimum number of feature (fitness function) selection for building the machine learning model (Genetic representation).

The process can be broadly divided as following:

1. Initialization:

Randomly generate a population with multiple chromosomes. Gene is the smallest unit and can be referred to as a set of characteristics (variables). We aim to join the Genes to obtain the Chromosomes(solution). The chromosome itself represents one candidate solution abstractly. The generation of Chromosome is user-defined (combination of numbers between 0 and 5 or only binary numbers).





2. Defining the fit function:

Now we need to define the evaluation criteria for best chromosomes(solution). Each chromosome is assigned with a fitness score by the fitness function, which represents the goodness of the solution. Let us say the fitness function is the sum of all the genes. Hence, the chromosome with the maximum sum is the fittest. In our case, the chromosome has a sum of 12.

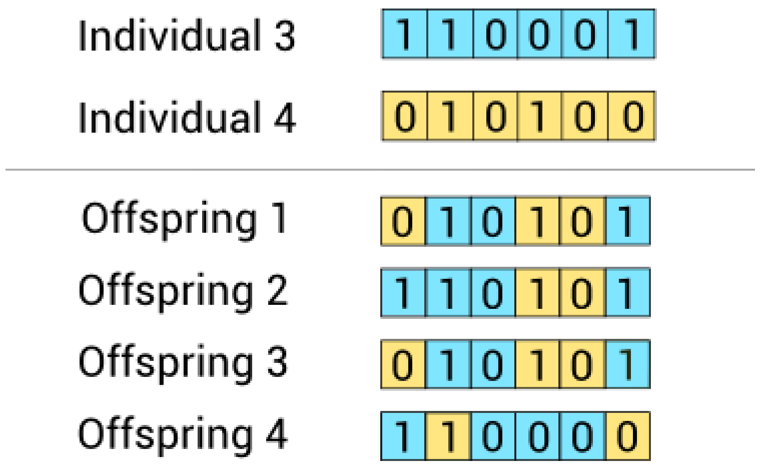
3. Selection:

Selecting the top 2 fittest chromosomes for creating the next generation. These will act as parents to generate offspring for the next generation which will naturally inherit the strong features. Two pairs of individuals (**parents**) are selected based on their fitness scores. Other chromosomes are dropped. Here are some of the methods of parent selection-

1. Roulette Wheel Selection
2. Rank Selection
3. Steady State Selection
4. Tournament Selection
5. Elitism Selection

4. Crossover:

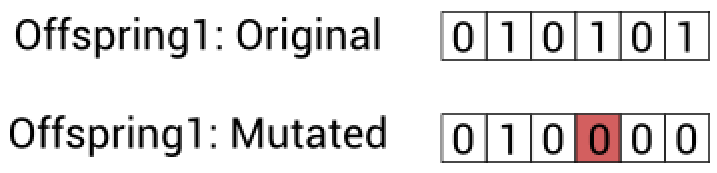
Crossover is the equivalent of two parents having a child. Each chromosome contributes a certain number of genes to the new individual. **Offspring** are created by exchanging the genes of parents among themselves until the crossover point is reached.



1. Single point crossover.
2. k-point crossover (k ≥ 1)
3. Uniform crossover.

5. Mutation:

To avoid the duplicity(crossover generates offspring similar to parents) and to enhance the diversity in offspring we perform mutation. The mutation operator solves this problem by changing the value of some features in the offspring at random.



These steps are repeated until the termination criteria is met.

When to apply Genetic Algorithm:

- There are multiple local optima.
- The objective function is not smooth (so derivative methods cannot be applied)
- Number of parameters is very large.
- Objective function is noisy or stochastic.

Advantages of Genetic Algorithm:

- Concept is easy to understand.
- Modular, separate from application.
- Answer gets better with time.
- Inherently parallel; easily distributed.
- Genetic algorithms work on the Chromosome, which is an encoded version of potential solutions' parameters, rather the parameters themselves.
- Genetic algorithms use fitness score, which is obtained from objective functions, without another derivative or auxiliary information

Disadvantages:

- Genetic Algorithms might be costly in computational terms since the evaluation of each individual requires the training of a model.
- These algorithms can take a long time to converge since they have a stochastic nature.

Conclusion:

As feature selection and Tuning hyperparameters are vital for any model building process, using advanced techniques like Genetic algorithms gives a great boost to your results.