**Genetic Algorithms & Online Search Agents**

**Genetic Algorithms**

A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state. The analogy to natural selection is the same as in stochastic beam search, except that now we are dealing with sexual rather than asexual reproduction.

**Advantages of GAs**

GAs have various advantages which have made them immensely popular. These include −

1. Does not require any derivative information (which may not be available for many real-world problems).
2. Is faster and more efficient as compared to the traditional methods.
3. Has very good parallel capabilities.
4. Optimizes both continuous and discrete functions and also multi-objective problems.
5. Provides a list of “good” solutions and not just a single solution.
6. Always gets an answer to the problem, which gets better over the time.
7. Useful when the search space is very large and there are a large number of parameters involved.

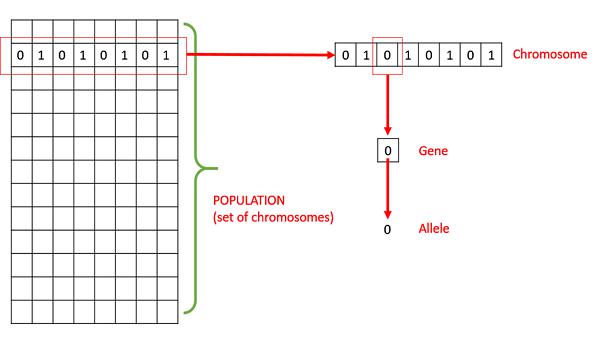
**Limitations of GAs**

Like any technique, GAs also suffer from a few limitations. These include −

1. GAs are not suited for all problems, especially problems which are simple and for which derivative information is available.
2. Fitness value is calculated repeatedly which might be computationally expensive for some problems.
3. Being stochastic, there are no guarantees on the optimality or the quality of the solution.
4. If not implemented properly, the GA may not converge to the optimal solution.

**Basic Terminology:**

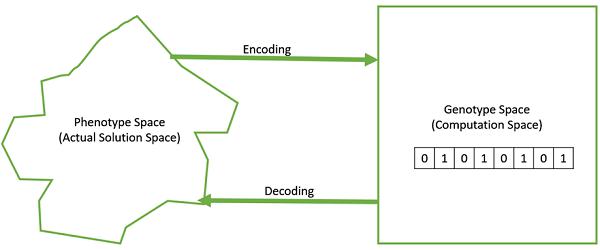
Before beginning a discussion on Genetic Algorithms, it is essential to be familiar with some basic terminology which will be used throughout this tutorial.

* **Population** − It is a subset of all the possible (encoded) solutions to the given problem. The population for a GA is analogous to the population for human beings except that instead of human beings, we have Candidate Solutions representing human beings.
* **Chromosomes** − A chromosome is one such solution to the given problem.
* **Gene** − A gene is one element position of a chromosome.
* **Allele** − It is the value a gene takes for a particular chromosome.

* **Genotype** − Genotype is thepopulation in the computation space. In the computationspace, the solutions are represented in a way which can be easily understood and manipulated using a computing system.
* **Phenotype** − Phenotype is the population in the actual real world solution space inwhich solutions are represented in a way they are represented in real world situations.
* **Decoding and Encoding** − For simple problems, the **phenotype and genotype** spacesare the same. However, in most of the cases, the phenotype and genotype spaces are different. Decoding is a process of transforming a solution from the genotype to the phenotype space, while encoding is a process of transforming from the phenotype to genotype space. Decoding should be fast as it is carried out repeatedly in a GA during the fitness value calculation.

For example, consider the 0/1 Knapsack Problem. The Phenotype space consists of solutions which just contain the item numbers of the items to be picked.

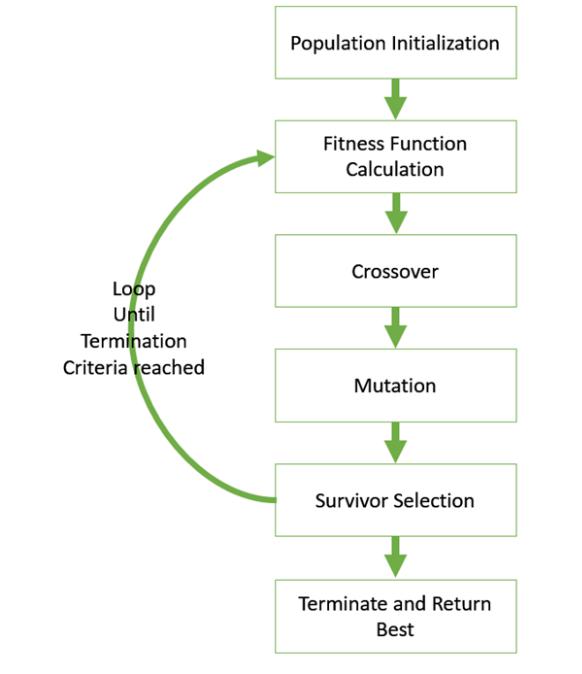
However, in the genotype space it can be represented as a binary string of length n (where n is the number of items). A **0 at position x** represents that **xth** item is picked while a 1 represents the reverse. This is a case where genotype and phenotype spaces are different.



* **Fitness Function** − A fitness function simply defined isa function which takes thesolution as input and produces the suitability of the solution as the output. In some cases, the fitness function and the objective function may be the same, while in others it might be different based on the problem.
* **Genetic Operators** − These alter the genetic composition of the offspring. Theseinclude crossover, mutation, selection, etc.

Basic Structure

* The basic structure of a GA is as follows −
* We start with an initial population (which may be generated at random or seeded by other heuristics), select parents from this population for mating. Apply crossover and mutation operators on the parents to generate new off-springs. And finally these off-springs replace the existing individuals in the population and the process repeats. In this way genetic algorithms actually try to mimic the human evolution to some extent.
* Each of the following steps are covered as a separate chapter later in this tutorial.



**Introduction to Crossover**

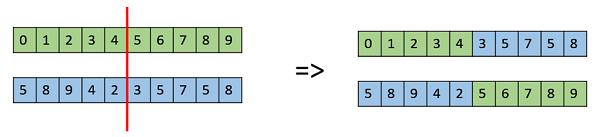
The crossover operator is analogous to reproduction and biological crossover. In this more than one parent is selected and one or more off-springs are produced using the genetic material of the parents. Crossover is usually applied in a GA with a high probability – ***pc***

**Crossover Operators**

In this section we will discuss some of the most popularly used crossover operators. It is to be noted that these crossover operators are very generic and the GA Designer might choose to implement a problem-specific crossover operator as well.

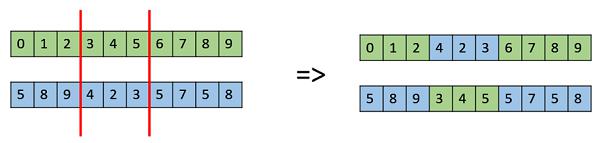
**One Point Crossover**

In this one-point crossover, a random crossover point is selected and the tails of its two parents are swapped to get new off-springs.



**Multi Point Crossover**

Multi point crossover is a generalization of the one-point crossover wherein alternating segments are swapped to get new off-springs.



**Uniform Crossover**

In a uniform crossover, we don’t divide the chromosome into segments, rather we treat each gene separately. In this, we essentially flip a coin for each chromosome to decide whether or not it’ll be included in the off-spring. We can also bias the coin to one parent, to have more genetic material in the child from that parent.

**Introduction to Mutation**

In simple terms, mutation may be defined as a small random tweak in the chromosome, to get a new solution. It is used to maintain and introduce diversity in the genetic population and is usually applied with a low probability – ***pm***. If the probability is very high, the GA gets reduced to a random search.

Mutation is the part of the GA which is related to the “exploration” of the search space. It has been observed that mutation is essential to the convergence of the GA while crossover is not.

Mutation Operators

In this section, we describe some of the most commonly used mutation operators. Like the crossover operators, this is not an exhaustive list and the GA designer might find a combination of these approaches or a problem-specific mutation operator more useful.

**Bit Flip Mutation**

In this bit flip mutation, we select one or more random bits and flip them. This is used for binary encoded GAs.



**Random Resetting**

Random Resetting is an extension of the bit flip for the integer representation. In this, a random value from the set of permissible values is assigned to a randomly chosen gene.

**Swap Mutation**

In swap mutation, we select two positions on the chromosome at random, and interchange the values. This is common in permutation based encodings.



**Scramble Mutation**

Scramble mutation is also popular with permutation representations. In this, from the entire chromosome, a subset of genes is chosen and their values are scrambled or shuffled randomly.



**Inversion Mutation**

In inversion mutation, we select a subset of genes like in scramble mutation, but instead of shuffling the subset, we merely invert the entire string in the subset.



**Online search agents and unknown environments**

So far we have concentrated on agents that use OFFLINE SEARCH offline search algorithms. They compute a complete solution before setting foot in the real world and then execute the solution. In contrast, an online search13 agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi-dynamic domains—domains where there is a penalty for sitting around and computing too long. Online search is also helpful in nondeterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that might happen but probably won’t. Of course, there is a trade-off: the more an agent plans ahead, the less often it will find itself up the creek without a paddle.

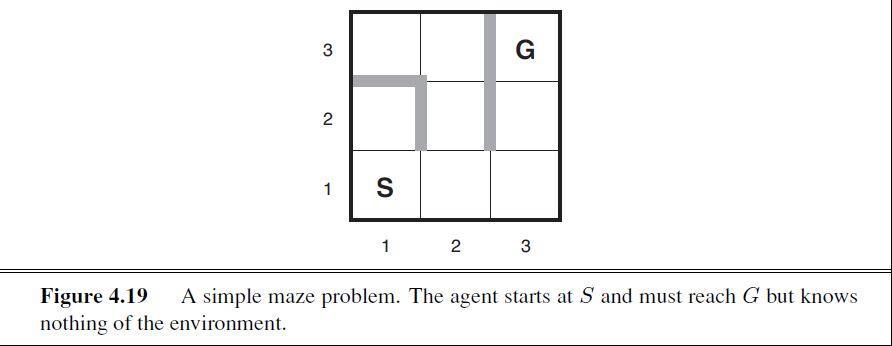
**Online search problems**

Online search is a necessary idea for unknown environments, where the agent does not know what states exist or what its actions do. In this state of ignorance, the agent faces an exploration problem and must use its actions as experiments in order to learn enough to make deliberation worthwhile.

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment but we stipulate that the agent knows only the following:

* ACTIONS(s), which returns a list of actions allowed in state s;
* The step-cost function c(s, a, s’)—note that this cannot be used until the agent knows that s’ is the outcome; and
* GOAL-TEST(s).

Note in particular that the agent cannot determine RESULT(s, a) except by actually being in s and doing a. For example, in the maze problem shown in below figure, the agent does not know that going Up from (1,1) leads to (1,2); nor, having done that, does it know that going Down will take it back to (1,1). This degree of ignorance can be reduced in some applications— for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.



**Online search agents**

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously. For example, offline algorithms such as A∗ can expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real actions. An online algorithm, on the other hand, can discover successors only for a node that it physically occupies. To avoid travelling all the way across the tree to expand the next node, it seems better to expand nodes in a local order. Depth-first search has exactly this property because (except when backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first search agent is shown in below figure. This agent stores its map in a table, RESULT[s, a], that records the state resulting from executing action *a* in state *s*. Whenever an action from the current state has not been explored, the agent tries that action. The difficulty comes when the agent has tried all the actions in a state.

