Randomized Search and Emergent Systems

Chapter 4

Escaping Local Maxima

We have seen that most real world problems create search spaces that are too huge to explore fully. The most studied problems with large solution spaces are *SAT* and *TSP*.A *SAT* problem on a hundred variables has 2^{100} candidate assignments. If we had a machine inspecting a thousand of these per second, and if the machine was started at the time of the Big Bang, fourteen billion years ago when the (current) universe came into being, it would have finished less than one percent of its task by now. Clearly, inspecting all the candidates is not a viable option. Similar search spaces crop up with most problems we formulate. With optimization problems like the *TSP,* the difficulty is compounded by the fact that we would usually[1] not recognize that a solution is optimal, even if we were to find it. Observe that we also see the *SAT* problem as an "optimization" problem during search, in which we attempt to optimize the heuristic value of a candidate solution. Only, in this case, the optimal value is recognized easily; for example when the number of unsatisfied clauses is zero, and we can terminate the search. Complete search not being a viable option for large problems, search methods like *Hill Climbing* and *Tabu Search* work with bounded memories. While *Hill Climbing* is conceptually simple, it can get stuck on a local optimum. In *Tabu Search,* the attempt is to diversify search, as opposed to only following the steepest gradient, often moving to a node that is not the best successor, or even a better one. This allows it to move away from local optima, and the possibility of moving towards the global one is kept open.

The steepest gradient ascent attempts to *exploit* the gradient information (Michalewicz and Fogel, 2004). To this, the *Tabu Search* adds an *explorative* component by trying to push the search into newer areas. It does this in a deterministic way, keeping a *Tabu* list of recent moves to be avoided. In this chapter, we look at randomized approaches to promoting exploration. First, we look at a way to randomize the *Hill Climbing* algorithm. Then, we look at other approaches motivated by the way random moves made in nature can lead to build up and preservation of good solutions.

4.1 Iterated Hill Climbing

Working in the solution space, our search algorithms perturb current

candidate solutions in search of better ones. The notion of a start node and a goal node in the state space is replaced by optimizing the value of the evaluation or heuristic function. The start node in the search tree has no particular significance when we are searching in the solution space. It is simply a candidate solution we begin with. For the *Hill Climbing* algorithm, this is the starting point of the steepest gradient ascent (or descent, if the problem is of minimization). Once the starting point is decided, the algorithm moves up (or down) along the steepest gradient, and terminates when the gradient becomes zero. The end point of the search is determined by the starting point, and the nature of the surface defined by the evaluation function. If the surface is monotonic then the end point is the global optimum; otherwise the search ends up at an optimum that is in some sense closest to the starting point, but may only be a local optimum. *Iterated Hill Climbing* exploits this property by doing a series of searches from a set of randomly selected different starting points. The hope is that the best value found by at least *one* of the searches will be the global optimum. The algorithm can be written as shown in Figure 4.1.

```
IteratedHillClimbing(n)
1  node ← random candidate solution
2  bestNode ← node
3  for i ← 1 to n
4    do  node ← random candidate solution
5        newNode ← Head(Sort_h(MoveGen(node)))
6        while h(newNode) > h(node)     /* ">" for maximization */
7              do
8                   node ← newNode
9                   newNode ← Head(Sort_h(MoveGen(node)))
10             if h(newNode) > h(bestNode)
11                  then bestNode ← newNode
12 return bestNode
```

FIGURE 4.1 Algorithm *Iterated Hill Climbing (IHC)* for a maximization problem. It does a number of *Hill Climbing* runs from random starting points.

The *Iterated Hill Climbing* approach will work well if the surface defined by the evaluation function is smooth at the local level, with perhaps a small number of local optima globally. Such a surface is illustrated in Figure 4.2.

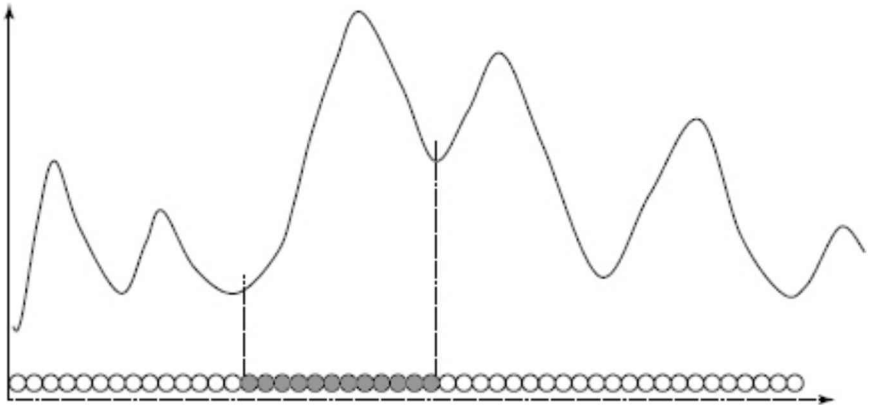

FIGURE 4.2 A smooth surface with a small number of local optima is well suited for *Iterated Hill Climbing*. A random starting point in any iteration from any of the shaded nodes would lead to the global maximum.

The *IHC* algorithm has the same space requirements as *Hill Climbing*. Both need a constant amount of space to run. The difference is that for different runs on the same problem, the *Hill Climbing* algorithm will always return the same result; while the *IHC* algorithm may return different results. This is because its performance is determined by the random choice of the random starting points. Figure 4.2 shows a set of shaded nodes from where the steepest gradient leads to the global maximum. Let us call this set the *footprint* of the *HC* algorithm. If in one of the outer loop iterations of *IHC*, a point in *this* region is chosen as the starting point then the algorithm will find the optimal solution. The likelihood of *IHC* finding the global optimum depends upon the size of the footprint. The larger the footprint is, the greater is the chance of starting in it, and finding the optimum. If the footprint covers the entire search space then *HC* itself will work. As the footprint gets smaller, one would need a larger number of iterations in the outer loop to have a good chance of finding the optimum. If the footprint is very small then the number of iterations in the outer loops may become prohibitively large, and one may have to look for an alternative approach. Such an evaluation function surface is depicted below in Figure 4.5, and the next section describes an approach that might work better there.

4.2 Simulated Annealing

The algorithms we have seen so far have all depicted deterministic search moves. The *IHC* algorithm above introduces a randomized aspect by choosing a series of random starting points followed by deterministic moves in the search space. We now look at the possibility of making random moves. The effect of this would be that even from the same starting point, the search may proceed differently for different runs of the algorithm. A completely random procedure is the *Random Walk*, in which the search process makes random moves in the search space in complete disregard of the gradient. The algorithm is shown below in Figure 4.3.

```
RandomWalk()
1  node ← random candidate solution or start
2  bestNode ← node
3  for i ← 1 to n
4       do node ← RandomChoose(MoveGen(node))
5            if h(node) > h(bestNode)
6                 then bestNode ← node
7  return bestNode
```

FIGURE 4.3 *RandomWalk* explores the search space in a random fashion. Function *RandomChoose* randomly picks one of the successors of the current node. The above algorithm has *n* random moves.

Given a surface defined by the evaluation function, *Random Walk* and *Hill Climbing* are two extremes of local search. *Random Walk* relies on *exploration* of the search space. Its performance is dependent upon time. The longer you explore the space, the more the chances of finding the global optimum. Because it is totally oblivious of the gradient, it never gets stuck on a local optimum. *Hill Climbing*, on the other hand, relies on the *exploitation* of the gradient. Its performance depends upon the nature of the surface. It terminates on reaching an optimum. If the surface has a monotonic gradient towards the global optimum, *HC* will quickly reach that. Otherwise, it will reach the nearest local optimum.

Simulated Annealing (*SA*) is an algorithm that combines the two tendencies, *explorative* and *exploitative*, of the two search methods. The basic idea is that the algorithm makes a probabilistic move in a random direction. The probability of making the move is proportional to the *gain* of value made by the move. Traditionally, this gain is associated with energy[2] and we use the term ΔE to represent the change in the evaluation value. The larger the gain, the larger is the probability of making the move. The algorithm will make a move even for negative gain moves with nonzero probability, though this probability decreases as the move becomes worse. The point is that the algorithm will make moves *against* the gradient too, but will have a higher probability of making better moves. Consider a maximization problem from which three states *A, B* and *C* are shown in Figure 4.4. Both *A* and *B* are maxima with *B* having a higher evaluation value. If the algorithm has to move from local maximum *A* to *C*, it will have a negative gain of ΔE_{AC}. Likewise, if it has to move from *B* to *C*, it will have to go through a negative gain of ΔE_{BC}. Since this is larger than ΔE_{AC}, it is likely that the algorithm moves from *A* to *C* more often than from *B* to *C*. Again, since the positive gain from *C* to *B* is higher; the algorithm is more likely to move from *C* to *B* than to *A*. That is, it is more likely that the algorithm will move from *A* to *B* than in the other direction. Then, given a series of local maxima of increasing magnitude, the search is likely to move towards the better ones over a period of time.

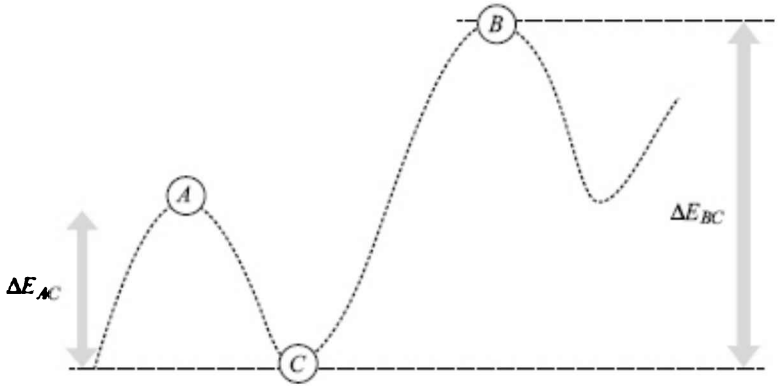

FIGURE 4.4 To move from *A* to *C*, the algorithm has to incur a smaller loss than to move from *B* to *C*. Simulated Annealing is more likely to move from *A* to *C* to *B* than vice versa.

One hopes, and this has been empirically supported by numerous applications, that by and large, the search will have a *tendency* to move to

better solutions. The search may not perform very well on surfaces like in Figure 4.2 where *IHC* worked well, but for jagged surfaces with many maxima, like the one shown in Figure 4.5 below, it will probably do well.

A randomized algorithm that has a simple and constant bias towards better values would be called *Stochastic Hill Climbing*. *Simulated Annealing* adds another dimension to this. It starts of being closer to the *Random Walk*, but gradually becomes more and more controlled by the gradient and becomes more and more like *Hill Climbing*. This changing behaviour is controlled by a parameter *T* called temperature. We associate high temperatures with random behaviour, much like the movement of molecules in a physical material. In *Simulated Annealing,* we start with a high value of *T* and gradually decrease it to make the search more and more deterministic. The probability of making a move at any point of time is given by a *sigmoid* function of the gain ΔE and temperature *T* as given below,

$$P(C, N) \leftarrow 1/(1 + e^{-\Delta E/T}) \tag{4.1}$$

where *P* is the probability of making a move from the current node *C* to the new node *N*, ΔE = (*eval(N) -eval(C)*) is the gain in value and *T* is an externally controlled parameter called temperature. Note that the above formula is for maximization of the evaluation function value. For minimization, the negative sign in the denominator will be removed. The probability as a function of the two values is illustrated in Figure 4.6. The *Y* axis shows the probability value, and the *X* axis varies ΔE. The different plots are for different values of *T*.

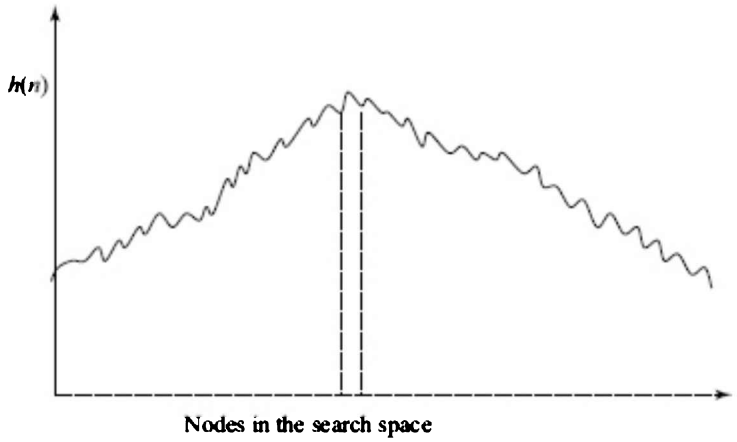

FIGURE 4.5 A jagged surface with many local optima, but a broader trend towards the optimum is well suited for *Simulated Annealing*. Observe that the footprint of *HC* is very small.

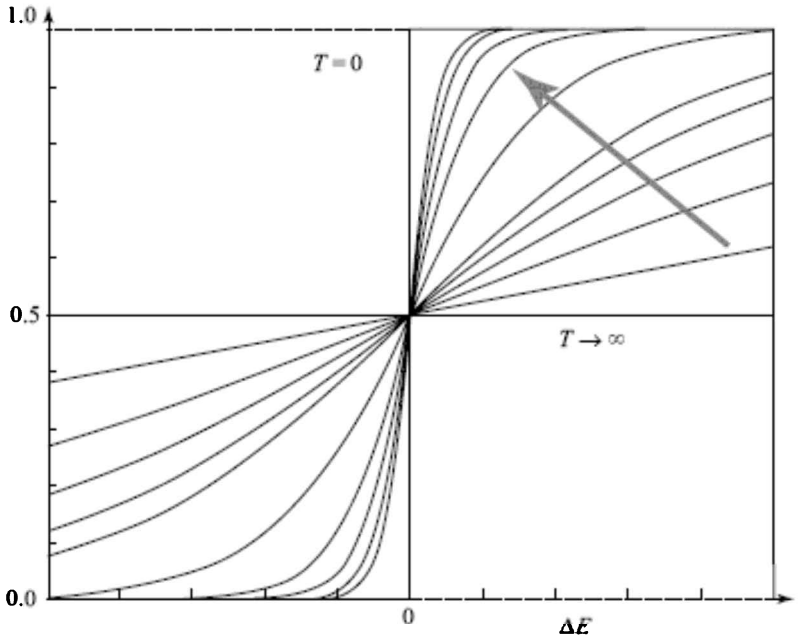

FIGURE 4.6 The probability curves (Sigmoid function) for different values of *T*. The probability of making a move increases as Δ*E* increases. For very large *T*, the algorithm behaves like *Random Walk*. As *T* tends to zero, the behaviour approaches the *Hill Climbing* algorithm.

When the temperature tends to infinity, the probability is uniformly 0.5, irrespective of the Δ*E* value. This means that there is equal probability of staying in the current node, and shifting to the neighbour. This is like the *Random Walk* algorithm. For finite values of *T,* one can see that when Δ*E* is positive, the probability is greater than half; and when it is negative, the probability is less than half. Thus, a move to a better node is more likely to be made, than a move to a worse node, though both have nonzero probability. For any given temperature, the greater the Δ*E*, the more the probability of the move being made. When Δ*E* is negative then the probability becomes lower as the loss in value becomes greater. Observe that for all temperatures, the probability is half when Δ*E* = 0. This means that when the neighbour evaluates to the same value as the current node, it does not matter and one may or may not move to it with equal probability. Finally, one can observe that as temperature *T* becomes lower, the Sigmoid probability function tends to become more and more like a step function. When *T* = 0, it is a step function and the decision to move to the neighbour becomes a deterministic one. The search moves to the neighbour (with probability = 1) if the neighbour is better (Δ*E* > 0), else it does not move to it (moves with probability = 0). This is like the *Hill Climbing* algorithm.

The *Simulated Annealing* algorithm does a large number of probabilistic moves with the temperature parameter being gradually reduced from a large initial value to a lower one. The manner in which the temperature is brought down is known as the *cooling schedule*. The intuition is that starting off with random moves that allow the search to explore the search space, the temperature is lowered gradually to make the search more exploitative of the

gradient information. As time goes by, the probability of the search being in the vicinity of the global optimum becomes higher, and at some point the gradient dominates the search and leads it to the optimum. The general outline of the *SA* algorithm is given below in Figure 4.7. There are many variations that can be done here, by choosing different cooling schedules. Another variation is one in which probabilistic moves are made only for negative gain moves.

```
SimulatedAnnealing()
1  node ← random candidate solution or start
2  bestNode ← node
3  T ← some large value
4  for time ← 1 to numberOfEpochs
5     do while some termination criteria  /* M cycles in a simple case */
6          do
7               neighbour ← RandomNeighbour(node)
8               ΔE ← Eval(neighbour) - Eval(node)
9               if Random(0, 1) < 1 / (1+e^(-ΔE/T))
10                        then node ← neighbour
11                   if Eval(node) > Eval(bestNode)
12                        then bestNode ← node
13                   T ← CoolingFunction(T, time)
14 return bestNode
```

FIGURE 4.7 Simulated Annealing makes probabilistic moves in the search space. We use the function *Eval* instead of *h* in the style used in optimization. Function *CoolingFunction* lowers the temperature after each epoch in which some probabilistic moves are made. Function *RandomNeighbour* randomly generates one successor of the current node, and *Random*(0,1) generates a random number in the range 0 to 1 with uniform probability.

One way that the *SA* algorithm is different from the many algorithms we have seen is that it *does not* do local optimization. It does not look around the neighbourhood of a node for the best neighbour. Instead, it generates one successor or neighbour randomly, and then decides probabilistically whether to move to it or not. Thus, it can easily be used in problems where there are a large number of neighbours for a given node. In domains where one is dealing with real-valued variables, there could potentially be an infinite number of neighbours, like on a real hill; but this would not deter one from writing a search algorithm.

4.3 Genetic Algorithms

The natural world around us is a manifestation of life. In the world, and including this world itself, "*things that persist, persist and things that don't, don't*" (Grand, 2001). We can say that life is made up of forms that persist. We can also say that the goal of life is *persistence*, that is, to exist, or to live. Whichever way we look at it, life forms or living creatures are manifestations of different strategies to persist. Each *species* of life forms represents a strategy in which the *individuals* strive to persist. All living creatures have finite lifespans, but life itself continues as individuals beget offspring, which carry forward the strategy to live.

Persistence of living creatures requires energy, and different mechanisms have evolved to consume resources to generate the energy that sustains life.

Plants feed on the energy in sunlight and nutrients in the soil; rabbits, cows and goats eat the plants; lions, foxes and tigers eat the rabbits, cows and the goats. Living creatures die naturally too, and are consumed by bacteria, which in turn produce the nutrients for plants. Life exists in an ecosystem, in which a multitude of individuals from different species exist in a dynamic equilibrium (see Figure 4.8).

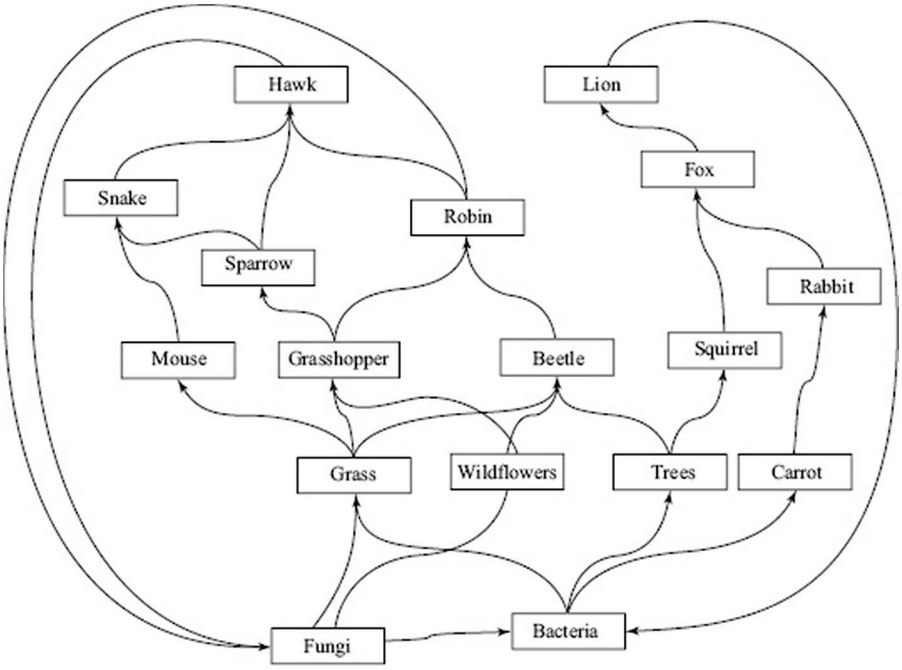

FIGURE 4.8 A small fragment of the vast ecosystem. The natural world contains millions of species interacting with each other. Arrows depict a positive influence of the population of one species on another.

Individuals may be born or may die, but life goes on. Even the extinction of a species or two does not stop this juggernaut. Once in a while though, some catastrophic event does occur that wipes out a part of the ecosystem and radically shifts the equilibrium, for example the one that happened about 65 million years ago wiped out the dinosaurs on Earth and created the opportunity for mammals to come to the fore.

Each species has its own strategy to live and consume resources, and when it can do that successfully, it survives. A species survives if the individuals of that species survive, at least long enough to create the next generation. Darwin called it the '*survival of the fittest*' (Darwin, 1859). A good strategy, a good design of a living creature, *a candidate solution for the problem of life*, is the one that survives. Life cannot explode limitlessly because the amount of matter is limited, and life uses matter to build creatures that live. And these creatures have to survive on the limited amount of energy that is available. So there is competition for survival, and life forms evolve by a process of natural selection. The ones that survive are the "fit" ones or good ones. The bad designs die out. In that sense, we can think of

nature as searching through different designs of living creatures.

While there is competition *between* the species, there is also competition *within* the species, perhaps more so. It is *this* competition that has led to the continual improvement of the species as a whole. The American paleontologist Edward Drinker Cope hypothesized that the body size of the members of a species grows with evolution (Cope's rule)[3]. This continues till a point when the members become so large, like the dinosaurs, that they become susceptible to rapid changes in environment, and become prone to extinction. It has been suggested that it is this competition that has resulted in the development of human brains and intelligence.

The species evolves and the ecosystem selects the individuals that survive longer. The species are interdependent upon each other. It is the individuals that participate in this interaction between the species. Each individual member strives for survival and propagation, and competes with other individuals for the means to do so. The fastest foxes catch the rabbits; the slower ones have to starve. The faster rabbits escape and live to eat and procreate. The early bird gets the worm. Life forms embody strategies that are more complex than just speed or timing. Foxes and rabbits may be clever too. Fruits and flowers rely on taste and looks for their seeds to be dispersed. Many predators and prey adopt disguises to hide from the other. In general, the strategies are complex and multi-faceted. Some simple strategies like that of a cockroach have survived for long periods, while other more complex ones like that of the cheetah seem endangered in this age of human dominance.

The offspring of a single individual would be like the individual in its strategy for survival. Nature has struck upon a novel way to produce improved designs. Procreation in nature is mostly through bisexual reproduction. Every species has male and female members, who mate and produce offspring. Selection happens because individuals that are alive and can find mates will get to reproduce, and others will not. When mating and reproduction does happen, the offspring *inherits features from both parents*. This process has the elements of a *random move* built in. Some children will inherit good features from both parents, and will be better than both, and will probably have better chances of survival and reproduction. In this way, "fitter" members of a species will survive and the population will have more and more fitter members.

The design of an individual is expressed in the genetic make-up of the individual, known as its *genotype*. A *chromosome* is a large macromolecule into which *DNA* is normally packaged in a cell and which contains many genes. The genes carry instructions encoded in a physical form in the *DNA* of an individual. Deoxyribonucleic acid (*DNA*) is a nucleic acid that contains the genetic instructions for the biological development of a cellular form of life or a virus[4]. *DNA* consists of a pair of molecules, organized as strands running start-to-end and joined by hydrogen bonds along their lengths. Each strand is a chain of chemical "building blocks", called nucleotides, of which there are four types: adenine (A), cytosine (C), guanine (G) and thymine (T). *DNA* is thought to have originated approximately 3.5 to 4.6 billion years ago, and is responsible for the genetic propagation of most inherited traits. In humans, these traits range from hair colour to disease susceptibility. The genetic

information encoded by an organism's *DNA* is called its *genome*. During cell division, *DNA* is replicated, and during reproduction is transmitted to offspring. The offspring's genome is a combination of the genomes of its parents. The *phenotype* of an organism, on the other hand, represents its actual physical properties, such as height, weight, hair colour, and so on. It is the physical entity or the phenotype that goes out and competes in the world. So while the genotype represents the inherited design, the phenotype is the individual that is the result of that design.

A newborn individual is like a computer program let loose in a system that can interpret it. Simple life forms, like the paramecium, earthworm, jellyfish or a cockroach, embody simple programs or strategies. More complex forms, like a typical mammal, embody more complex strategies. "Intelligent" life forms deploy very flexible strategies[5]. And in each species, the competition within has led to a continuous improvement in the strategy, through a process of natural selection.

Genetic Algorithms (*GA*) are a class of algorithms that try and build solutions by introducing evolution and selection of the best in a population of candidate solutions. The first thing one must do is to encode the problem by devising a schema for candidate solutions. One can think of this as an artificial chromosome, or the *DNA* sequence that is the blueprint of the solution. The chromosome is made up of the different components that make up the solution. Starting with a population of strings, *GAs* involve churning of the genes in search of better combinations.

There are three basic operations in *Genetic Algorithms* (Goldberg, 1989):

1. Selection

The selection operator allows the fitter candidates to be chosen for reproduction, and thus propagation of the genes (components). While in nature, the phenotype is out in the world competing for survival, such an approach is not suitable because the purpose of writing the *GA* is to produce a good design (solution). It would be too time consuming to build the phenotypes and test them in the real world, though such an approach has been used where simulation is possible (Sims, 94). In practice, *GAs* employ a user specified function to decide which designs/solutions are good or not. This function is called the *fitness function*, and gives a fitness value to each candidate. In optimization, we called this function the *evaluation function*. The selection operator takes a population of candidates and reproduces (clones) them in proportion to their fitness values. Fitter candidates will have more copies made, and the worst ones may not get replicated.

2. Recombination

The recombination operator takes the output of the selection operator, and randomly mates pairs[6] of candidates in the population, producing offspring that inherit components (genes) from both parents. The new population thus contains a totally different set of solutions. In order to preserve the best candidates, sometime *elitism* is followed, that allows the best few candidates to have cloned copies in the new generation.

3. Mutation

Once in a while in the real world, a mistake happens and a child gets a mutated copy of a gene from the parent. Most of the time this happens, it is disastrous for the individual, but on a rare occasion the mutated gene is beneficial, giving rise to a more successful individual. *GAs* incorporate mutations to allow for the possibility of making good random moves. As a result, even when the population does not contain a good gene, there is a chance that it may arise out of a random mutation.

A typical framework of a *Genetic Algorithm* is given below. The most commonly used operator to recombine the genes from the two parents is known as the *crossover* operator. A *single point crossover* simply cuts the two parents at a randomly chosen point and recombines them to form two new solution strings. For example, if the solution has eight components then given the two parents:

$$P_1 = X_1X_2X_3X_4X_5X_6X_7X_8$$
$$P_2 = Y_1Y_2Y_3Y_4Y_5Y_6Y_7Y_8$$

If we choose a crossover point between components 5 and 6, we get the two children as follows,

$$C_1 = Y_1Y_2Y_3Y_4Y_5X_6X_7X_8$$
$$C_2 = X_1X_2X_3X_4X_5Y_6Y_7Y_8$$

A two point crossover would be equivalent to doing the above operation twice. One can define many crossover operators which will take components from the parents and mix them up. Care has to be taken that the resulting strings are valid candidates. This is easy when dealing with problems like *SAT,* where the i^{th} bit is a value of the i^{th} variable, but for most problems, the crossover point should be at a component level. Some problems like the *TSP* will not work with the crossover depicted above. We will look at *TSP* separately since it is a problem of considerable interest.

The algorithm *GA* is described at a high level in Figure 4.9.

In the algorithm shown in Figure 4.9, if we make the population size 1, and work only with the mutation operator then we would have the *Random Walk* or *Simulated Annealing* algorithms. The novel feature in *GA* is that it combines two processes working on a population of candidates. One recombines the components, and the other selects the best candidates. It is this dual action that makes *GAs* different. Without recombination, the algorithm would reduce to a parallel *Random Walk* or *Simulated Annealing*, where the mutation operator would be used to perturb solutions. *Perturbation* is a local change in the solution, and affects a local move in the neighbourhood graph. The crossover operator, on the other hand, results in a (big) jump in the neighbourhood graph. We can think of the crossover as a move from two parents to the two children. The two children may not be anywhere in the vicinity of the two parents, unless the two parents are similar to each other. At the same time, the *Selection* operator chooses the best (fittest) members and makes more copies of them. As the above two processes work in tandem, the population becomes fitter and fitter. This leads to more and more members accumulating near the peaks in the search space

(see Figure 4.10 below). If the number of peaks is small, this will in turn lead to a population that has more members that are similar to each other because they are all congregating near the peaks where the fitness values are high. Since more and more similar parents will be chosen to mate, the children will be closer to both of them. If there is only one peak, eventually the population will stabilise to most members becoming similar to each other and the genetic variation will diminish[7], leading to fewer changes in future generations (Figure 4.11). This convergence behaviour is somewhat similar to the one in *SA*, which too starts off jumping about the search space.

```
GeneticAlgorithm()
  1 Initialize an initial population of candidate solutions p[1..n]
  2 repeat
  3     Calculate the fitness value of each member in p[1..n]
  4     selected[1..n] ← the new population obtained by picking n members
  5                    from p[1..n] with probability proportional to fitness
  6     Partition selected[1..n] into two halves, and randomly mate and
  7                           crossover members to generate offspring[1..n]
  8     With a low probability mutate some members of offspring[1..n]
  9     Replace k weakest members of p[1..n] with the k strongest members of
 10                                                          offspring[1..n]
 11     until some termination criteria
 12     return the best member of p[1..n]
```

FIGURE 4.9 The *GA* algorithm works by reproducing a population in proportion to fitness, recombines the genes by crossover, and randomly mutates some members in each cycle. Parameter *k* decides how many of the parent population is to be retained.

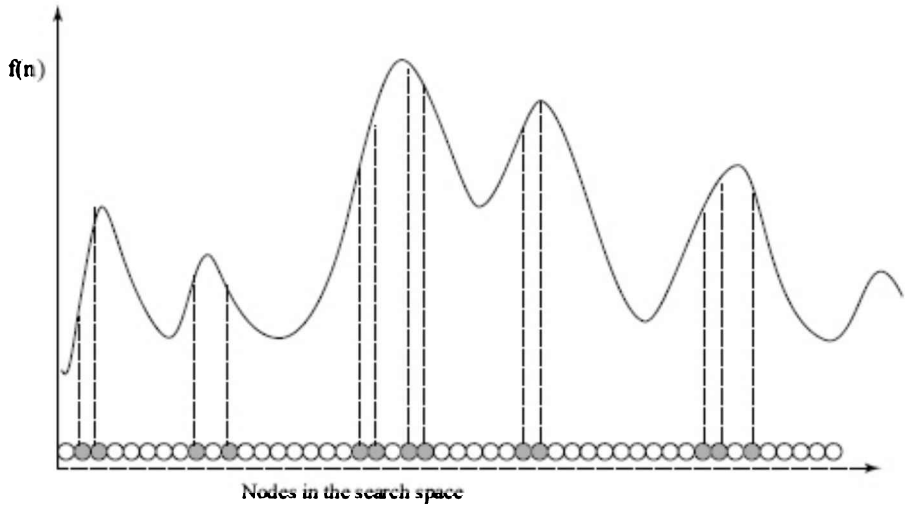

FIGURE 4.10 The Initial population may be randomly distributed, but as Genetic Algorithm is run, the population has more members around the peaks.

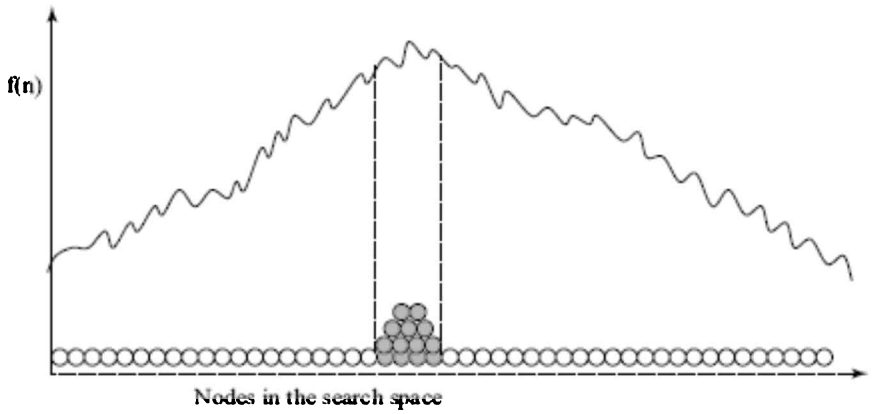

FIGURE 4.11 If there is one major peak, then most of the population is expected to converge towards that peak, leading to high similarity in the genetic make-up.

The genetic algorithms described here are instances of randomized processes at work. In particular, the mating and crossover is done by random selection from fit members. In the natural world, this process is far from random. Selection of mates is a conscious process in which attraction between opposite sexes plays a major role. One could say that both the concept of beauty and also its perception have *evolved* to help members choose mates that will result in better offspring. In a way then, that is a process of selection that operates at a different level. Another factor that affects mating is geographic location. Living creatures usually find mates from a nearby area. The structures of society and the often complex processes of matchmaking too, play a role. Many a time these are designed to serve a community or a clan, rather than the individual.

4.4 The Travelling Salesman Problem

The '*Travelling Salesman Problem*' (*TSP*) is one of the harder problems around, and considered by some to be the holy grail of computer science. The problem can informally be defined as follows. Given a graph in which the nodes are locations or cities, and edges are labelled with the cost of travelling between cities, to find a cycle containing each city exactly once, such that the total cost of the tour is as low as possible. Thus, the salesman must start from a city, visit each city exactly once, and return home incurring minimum cost. The cost may be distance, time, or money. Most studies of *TSP* assume a completely connected graph. While practical problems, like a country map, generate graphs that are not complete, one can make them complete by adding edges with a very high cost.

Given N cities, a tour may be represented by the order in which the cities are represented, ($City_1$, $City_2$,..., $City_N$) which is often abbreviated to (1, 2, ..., N). This is known as the *path representation*. One can observe that there are $N!$ permutations possible with N cities, each representing a tour. Every tour can be represented by N of these permutations, where each is a rotation of another. For example, in a nine city problem, (1, 2, 3, 4, 5, 6, 7, 8, 9) represents the same tour as (5, 6, 7, 8, 9, 1, 2, 3, 4). One must remember

that a tour is a complete cycle, so after the last city in the path representation, one has to go back to the first one. Further, one can observe that reversing the order does not change the cost of the tour because this simply means that the salesman goes in the opposite direction on the same tour. This assumes that the graph is not a directed graph. Thus, the number of distinct tours is $N!/(N \times 2) = (N - 1)!/2$. In practice, in an algorithm, one may not be able to prune the duplicate solutions always. In general, the number of candidate solutions grows as a factorial of the number of cities which, as we see below, is much worse than the exponential function, which is much larger than the exponential growth of *SAT* problems.

Let us look at the value 2^{100} which is the number of candidates for a 100-variable *SAT* problem. Two to the 100th power is 1, 267, 650, 600, 228, 229, 401, 496, 703, 205, 376. In the US number naming system, it is one nonillion, 267 octillion, 650 septillion, 600 sextillion, 228 quintillion, 229 quadrillion, 401 trillion, 496 billion, 703 million, 205 thousand, 376. This is about 10^{30}. If one were to take a sheet of paper 0.1 millimetre thick and double the thickness (by folding it) one hundred times, the resulting stack would be 13.4 billion light years tall. It would reach from Earth to beyond the most distant galaxy we can see with the most powerful telescopes—almost to the edge of the observable universe[8]. A hundred variable *SAT* is hard enough. But 100! is a much bigger number. The following output from a simple Lisp[9] program shows the number.

```
(factorial 100) → 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

This number is larger than 10^{157} and clearly it is impossible to inspect all possible tours in a hundred-city problem. Thus, the general case of *TSP* is a prime candidate for applying stochastic (randomized) local search methods. While inspecting all candidates is not going to be an approach, (using techniques of dynamic programming, one can solve the problem exactly in time $O(2^n)$) the *TSP* problem has been shown to be NP-hard (Gary and Johnson, 1979). Exact solutions are hard to find for a given large problem, and thus it makes it difficult to evaluate one's algorithm. A library of *TSP* problems *TSPLIB*[10] (Reinelt, 2004) with exact solutions is available on the Web. An exact solution for 15, 112 German cities from *TSPLIB* was found in 2001 using the cutting plane method proposed by George Dantzig, Ray Fulkerson and Selmer Johnson in 1954, based on linear programming (Dantzig, 1954). The computations were performed on a network of 110 processors. The total computation time was equivalent to 22.6 years on a single 500 MHz Alpha processor. In May 2004, the travelling salesman problem of visiting all 24,978 cities in Sweden was solved: a tour of approximately 72,500 kilometres was found and it was proven that no shorter tour exists. In March 2005, the travelling salesman problem of visiting all 33,810 points in a circuit board was solved using Concorde (Cook, 2006). A tour of length 66,048,945 units was found and it was proven that no shorter tour exists. The computation took approximately 15.7 CPU years[11] .

Stochastic Local Search (SLS) methods (Hoos and Stutzle, 2005), on the other hand, can find very good solutions quite quickly. For example, for randomly generated problems of 25 million cities, a solution quality within 0.3

percent of the estimated optimal solution was found in 8 CPU days on an IBM RS6000 machine (Applegate, 2003). More results on performance can be obtained from the Website for the DIMACS implementation challenge on *TSP* (Johnson et al., 2003) and (Cook, 2006). Some more interesting *TSP* problems available at (Cook, 2006) are: The *World TSP* -A 1,904,711-city TSP consisting of all locations in the world that are registered as populated cities or towns, as well as several research bases in Antarctica; *National TSP* Collection—a set of 27 problems, ranging in size from 28 cities in Western Sahara to 71,009 cities in China. Thirteen of these instances remained unsolved, providing a challenge for new *TSP* codes, and; *VLSI TSP* Collection—a set of 102 problems based on VLSI data sets from the University of Bonn. The problems range in size from 131 cities up to 744,710 cities.

TSP problems arise in many applications (Johnson, 1990), for example circuit drilling boards (Litke, 1984), where the drill has to travel over all the hole locations, X-ray crystallography (Bland and Shallcross, 1989), genome sequencing (Agarwala, 2000) and VLSI fabrications (Korte, 1990). These can give rise to problems with thousands of cities, with the last one reporting 1.2 million cities. Many of these problems are what are known as *Euclidean TSP*s, in which the distance between two nodes (cities) is the Euclidean distance. One can devise approximation algorithms that work in polynomial time. Arora (1998) reports that in general, for any $c > 0$, there is a polynomial time algorithm that finds a tour of length at most $(1 + 1/c)$ times the optimal for geometric instances of *TSP*, which is a more general case of an Euclidean *TSP*. Special cases of *TSP*s can be solved easily. For example, if all the cities are known to lie on the perimeter of a convex polygon, a simple greedy algorithm, *TSP-Nearest-Neighbour* shown below works.

4.4.1 Constructive Methods

Several constructive methods construct tours from scratch. We look at a few of them starting with the most intuitive nearest neighbour construction (Figure 4.12).

```
TSPNearestNeighbour()
  1  currentCity  ← some startCity
  2  tour  ← List(currentCity)
  3  while an unvisited city exists
  4       do  neighbour  ← NearestAllowedNeighbour(currentCity)
  5           tour ←  Append(tour, List(neighbour))
  6           currentCity ←  neighbour
  7  return tour
```

FIGURE 4.12 The *Nearest Neighbour Tour* construction algorithm moves to the nearest neighbour (that has not been visited) at each stage. Function *NearestAllowedNeighbour* picks the nearest neighbour from the unvisited cities. The last segment back to the *startCity* is implicit.

The complexity of the algorithm is $O(bn)$, where n is the number of cities and b is the maximum degree of nodes (which is $n - 1$ for fully connected graph). We can observe that for most greedy algorithms for *TSP*, the complexity is $O(n^2)$. For *TSP* problems that satisfy the triangle inequality[12], it

has been shown that the tour found by the nearest neighbour algorithm is not more than $(\lceil \log_2(n) \rceil + 1)/2$ times the optimal tour cost (Rosenkrantz et al., 1977). In practice, the algorithm yields fairly good tours where there may be a few long edges that are added in the final stages. The reader can verify this by trying out the method on the problem shown in Figure 4.13. Observe that if the two extreme cities were not there, the cities would have satisfied the condition of being on a convex polygon, and the algorithm would have found the optimal solution.

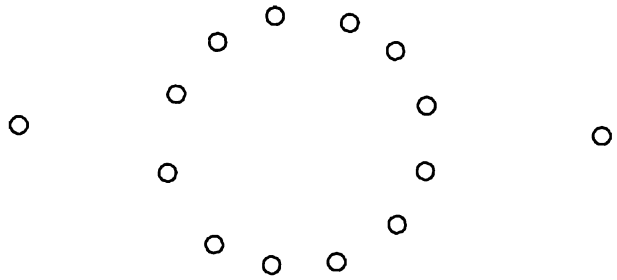

FIGURE 4.13 A near polygon arrangement of cities. The distances are the Euclidean distances in the figure.

We can improve upon the above algorithm slightly by extending the partial tour at the either end, choosing the one that has the nearest neighbour. The above algorithms work by adding nodes (cities) to the partial tours. The new node to be added is the one that is nearest to either end of the partial tour and extends the partial tour. Another heuristic method also adds nodes to the partial path, but selects the node to be added as the one that is nearest to *any node* in the partial tour. The new node is inserted into the partial tour by connecting it to the nearest node and splicing the new node. This is known as the *nearest insertion* heuristic, and it has been shown that the resulting tour costs are at most twice the optimal tour costs (Rosenkrantz et al., 1977). The reader is encouraged to try the heuristic on the problem in Figure 4.13.

We now look at constructive methods that add edges instead of nodes. The idea is that one adds edges making up the tour in some order, eventually stringing together the entire tour. In an algorithm known as the *Greedy Heuristic,* one starts off by sorting the edges in order of their costs (or weights or distances). The algorithm begins with the graph G containing all the nodes and no edges. One then keeps adding the cheapest edges to a partial tour being constructed in the graph G, such that no node ever has a degree more than two, and no cycles except the final tour exists.

Another method known as the *Savings Heuristic* starts off by constructing $(n - 1)$ tours of length two, all originating from a base node n_b. At the start, the i^{th} tour looks like (n_b, n_i, n_b). The algorithm then performs $(n - 2)$ merges, at each stage removing two edges from two cycles to $n_b$and adding an edge to connect the two hanging nodes. Of the four combinations of edges that can be removed and replaced, the algorithm chooses the one that gives maximum *savings* in combined tour cost. Figure 4.14 below illustrates the algorithm.

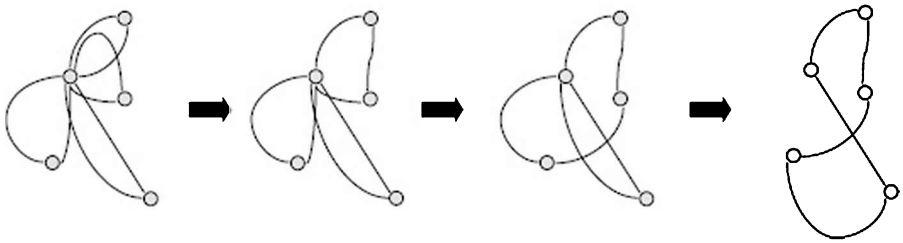

FIGURE 4.14 The *Savings Heuristic* starts with (n – 1) tours of length 2 and performs (n – 2) merge operations to construct the tour.

It has been shown (Ong and Moore, 1984) that the *Greedy Heuristic* produces tours at most (1 + log(n))/2 longer than the optimal tour, and the *Savings Heuristic* at most twice of that. In practice, however, the *Savings Heuristic* has empirically produced better tours. The following tours from the DIMACS Webpage (Figure 4.15) illustrate typical performances of these heuristic algorithms.

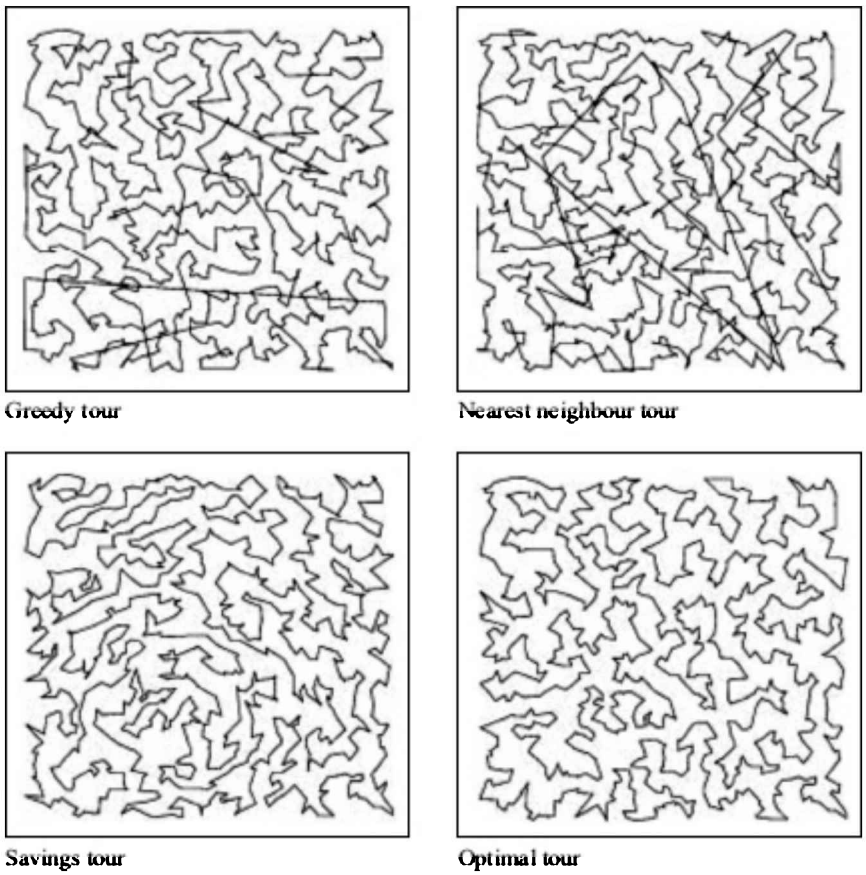

FIGURE 4.15 Tours found by some heuristic constructive algorithms. Figure taken from *http://www.research.att.com/~dsj/chtsp/*

4.4.2 Perturbation Methods for the TSP

Search methods that operate in the solution space can be easily applied to *TSP* problems. They work on complete tours, and the neighbours of a search node are tours obtained by perturbing the given tour in some way. In the path based representation of a tour, a given tour can be modified by exchanging any two cities in the tour, as shown in Figure 4.16. For example, cities 4 and 5 can be exchanged in (2, 3, 4, 1, 7, 6, 5, 8, 9) to give (2, 3, 5, 1, 7, 6, 4, 8, 9). We can call this the 2-city-exchange move. For a tour of N cities, this generates ${}^{N}C_{2}$ neighbours for any given tour. We can also exchange more than two cities to design a neighbourhood function k-city-exchange. Notice that if we remove 3 cities from the tour, we can put them back in (3!–1) = 5 other different ways, thus giving us ${}^{N}C_{3}$* (3!-1) neighbours for any given tour. In general, as k increases, the neighbourhood function becomes denser.

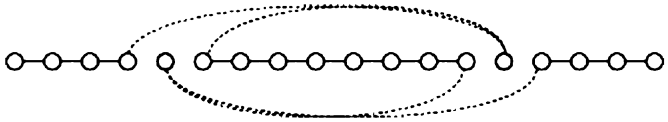

FIGURE 4.16 In the 2-city-exchange, the two shaded cities exchange their position in the path representation. The new tour has the dashed edges instead of the four broken ones in the linear order.

Another way of defining neighbourhood functions is to think of exchanging edges instead of cities in a tour. In a 2-edge exchange move, we would remove two edges and cross connect them to give a different tour, as shown in Figure 4.17. For example, in the tour (2, 3, 4, 1, 7, 6, 5, 8, 9), we could remove edges 4-1 and 8-9 and replace them with 4-8 and 9-1 to give a tour (2, 3, 4, 8, 5, 6, 7, 1, 9). The reader is encouraged to verify that this is the only way the two edges can be replaced to give a valid tour. It can also be observed that the 2-edge-exchange can be implemented by reversing a subsequence, in this case 1-7-6-5-8 in the path representation. We can do a 3-edge exchange by taking out and replacing three edges from the tour, and this can be done in four different ways (see Figure 4.18).

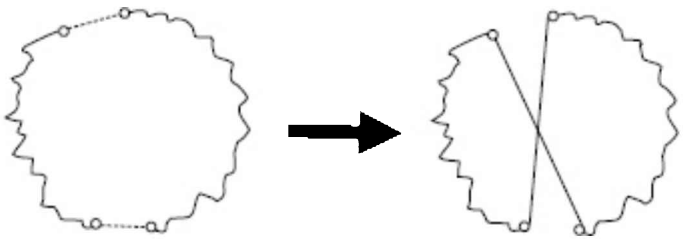

FIGURE 4.17 In the 2-edge-exchange, the tour is altered as shown.

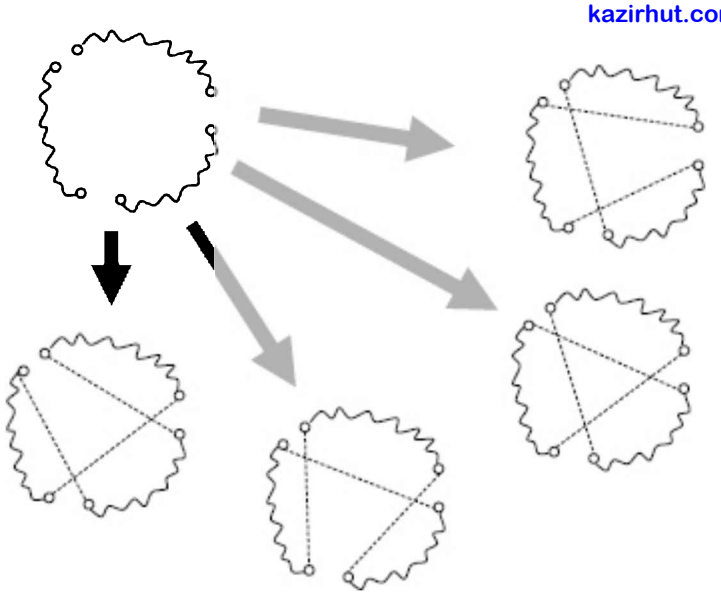

FIGURE 4.18 In a 3-edge-exchange, three edges are removed and replaced by other edges.

A closer look at the 2-city-exchange shows that it is one particular case of 4-edge-exchange in which two sets of two consecutive edges are removed from a tour, and each freed node is connected to the neighbours of the other freed node. Experiments have revealed that edge exchanges give better results than city exchanges. They are also intuitively more appealing. For example, one can visualise the 2-edge-exchange undoing a crossed path in a tour (if the move shown in the figure were reversed). It also allows for using a heuristic like selecting longer edges to be removed from a given tour. In general, using a higher *k*-exchange operator gives a denser neighbourhood. In the case where $k = n$, the entire search space comes in the neighbourhood, as perturbations are no longer local. The graph is fully connected as all permutations become neighbours. We have seen that a deterministic search method like *Variable Neighbourhood Descent* (Chapter 3) can exploit increasingly denser neighbourhood functions. The *TSP* problem, like the *SAT* problem, allows for a number of neighbourhood functions of increasing density, and one can implement algorithms that work within the capacity of a given set of resources.

4.4.3 GAs for TSP

Instead of perturbing a given solution, *GAs* generate new candidates by inheritance and recombination of solution components from two parents selected, based on their fitness. The idea is to serendipitously combine good components that occur in the two parents. In the *TSP* problem, the components are the individual segments in a tour, and also subtours made up a sequence of segments. For *GAs* to work well, one must be able to devise crossover operators that allow for such recombination. This is a somewhat difficult task, and different crossovers, and even alternate representations,

have been tried. We look at a few of them here.

Path Representation

In path representation, the simple one point or multipoint crossovers defined in this chapter earlier do not work because the resulting sequences are not likely to be valid tours. For example, given two parent tours

P_1 = (2, 4, 7, 1, 5, 9, 8, 3.6), and
P_2 = (4, 5, 7, 2, 6, 8, 1, 9, 3)

if we do a crossover after four segments, we get the two offspring:

C_1 = (2, 4, 7, 1, 6, 8, 1, 9, 3), and
C_2 = (4, 5, 7, 2, 5, 9, 8, 3.6)

Neither of the two offspring is a valid tour because cities repeat in them. We need crossover operators that will retain the *n* cities and only introduce a different order. Some interesting crossovers that have been tried out (Michalewicz and Fogel, 2004) are as follows.

The *Partially Mapped Crossover* (PMX) builds a child as follows. It chooses a random subsequence from one parent, and fills in the remaining tour by maintaining the order and position of cities as in the other parent. For the above example, choosing the subsequences from fourth to seventh city gives the children.

C_1 = (x, x, x, 1, 5, 9, 8, x, x), and
C_2 = (x, x, x, 2, 6, 8, 1, x, x)

This defines a series of mappings (1↔2), (5↔6), (9↔8) and (8↔1). Next, the cities that can be copied from the other parent are brought in to give

C_1 = (4, x, 7, 1, 5, 9, 8, x, 3), and
C_2 = (x, 4, 7, 2, 6, 8, 1, 3, x)

Finally, the four x's in the above children are replaced by 6 (from the mapping (5↔6) because 5 should have been there), 1 (from the mappings (9↔8) and (8↔1), 9 (from (1↔2), (8↔1) and (8↔9) – it should have been 2, but since 2 is already there 1, but since 1 is also there 8, and since 8 is already there 9), and 5, to give the two offspring

C_1 = (4, 6, 7, 1, 5, 9, 8, 6, 3), and
C_2 = (9, 4, 7, 2, 6, 8, 1, 3, 5)

In *Order Crossover* (OX), we copy a substring from one parent as in PMX, but fill in the remaining nodes in the order they occur in the other parent. Starting with

C_1 = (x, x, x, 1, 5, 9, 8, x, x), and
C_2 = (x, x, x, 2, 6, 8, 1, x, x)

the remaining cities for $C_1$are arranged in order as in P_2= (4, 5, 7, 2, 6, 8, 1, 9, 3) giving (4, 7, 2, 6, 3).

Likewise for the other child, the remaining cities are arranged in the order of P_1giving the two offspring

C_1 = (1, 5, 9, 8, 4, 7, 2, 6, 3) and
C_2 = (2, 6, 8, 1, 4, 7, 5, 9, 3)

In *Cycle Crossover* (CX), the attempt is made to inherit the position of each city in the offspring from one of the two parents as far as possible. We start constructing the two offspring with the first city as

C_1 = (2, x, x, x, x, x, x, x, x), and
C_2 = (4, x, x, x, x, x, x, x, x)

Since CX tries to inherit the position of each city from one of the parents, it can be seen that in $C_1$the position of city 4 can only be inherited from P_1, because the first slot where 4 occurs in $P_2$is already used up in C_1. Likewise, the position of city 2 in C_2can only be inherited from P_2, giving

C_1 = (2, 4, x, x, x, x, x, x, x), and
C_2 = (4, x, x, 2, x, x, x, x, x)

Now since 4 is in second place in C_1, the position of the other city in the second place, that is city 5, can only be inherited from one parent, which is C_1. In this manner, we propagate the constraints till we cannot fill up a city without creating a cycle in the partial tour. At this stage, we have

C_1 = (2, 4, x, 1, 5, 9, 8, 3.6), and
C_2 = (4, 5, x, 2, 6, 8, 1, 9, 3)

The remaining cities, in this case there is only one, is filled up from the other parent. Let us look at another example where a smaller subtour is selected. Given two parents

P_3 = (1, 2, 3, 4, 5, 6, 7, 8, 9), and
P_4 = (3, 4, 6, 5, 2, 7, 9, 8, 1)

the reader should verify that the subtours chosen by from one parent are

C_3 = (1, x, 3, x, x, 6, 7, x, 9) from P_1, and
C_4 = (3, x, 6, x, x, 7, 9, x, 1) from P_2

In the positions marked x, the cities from the two parents are exchanged, giving

C_3 = (1, 4, 3, 5, 2, 6, 7, 8, 9), and
C_4 = (3, 2, 6, 4, 5, 7, 9, 8, 1)

We can see the cycle crossover as a carefully chosen simple multipoint crossover, in which selected parts of the two “chromosomes” are exchanged.

Ordinal Representation

Interestingly, there does exist a tour representation where the simple crossover can be used producing valid tours. This is known as the ordinal

representation. We begin by arranging the cities to create a reference order R of the cities. Since we are interested in the index of cities in this reference, we name the cities themselves by letters to avoid confusion.

R = (A, B, C, D, E, F, G, H, I)

Then a tour, say C-D-F-E-B-G-I-H-A, is represented as a list L of references which is constructed as follows. The first entry in L is the reference index from R for the first city in the tour, in this case, city C in position 3 in R. We also modify the reference index R by deleting the city C from it.

L = (3)
R = (A, B, D, E, F, G, H, I)

The next city in the tour D is now in the third position in the modified reference R. We have,

L = (3, 3)
R = (A, B, E, F, G, H, I)

The next city F is in the fourth position in the updated R.

L = (3, 3, 4)
R = (A, B, E, G, H, I)

Continuing in this manner, we get a tour representation for the given tour as,

L = (3, 3, 4, 3, 2, 2, 3, 2, 1)

The reader is encouraged to verify that with the ordinal representation, one can use an arbitrary simple crossover operator that will yield valid tours as offspring. The advantage is that while the coding of tours may take some computational effort, once we have a population of tours available, the *GA* implementation becomes faster because we can use, say a single point crossover.

Adjacency Representation

Another representation of tours that has been experimented with is the adjacency representation. This is an indexed representation in which $city_i$ is in position j in the list if in the tour the salesman goes from $city_j$ to $city_i$. For example, the representation (4, 5, 7, 2, 6, 8, 1, 9, 3) can be interpreted as follow. Starting with 1, the next city in the tour is 4, because 4 is in the first place in the representation. From 4, the next hop is to city 2, because 2 is in the fourth position. The complete tour can in this manner be constructed as 1-4-2-5-6-8-9-3-7-1.

One problem with the above representation is that not every permutation of cities represents a valid tour. For example, no permutation can start with 1, because that would mean going from 1 to 1. Furthermore, any permutation that contains say (3, x, 5, x, 1, x, x, x, x), is not a tour because it contains a cycle 1-3-5-1. Also, the single point crossover will not work. The appeal of the representation lies in the fact that it represents explicitly where to go from any

given city. Thus, we can say in the tour (4, 5, 7, 2, 6, 8, 1, 9, 3) that after city 3 one goes to city 7, because 7 is in the third location in the representation. Given two parents, in the adjacency representation, the two options in the two parent tours are available at the same location, and could thus be inherited from either parent. One could then construct a child tour by choosing the next location to go to, using some heuristic approach.

In the *Heuristic Crossover* (HX), a child tour is constructed by choosing a random city as a starting point. The next city is chosen from the two options in the two parents, by choosing the one that is linked by a shorter edge. One has to keep a lookout for the formation of a cycle in the tour, and if that is happening at some stage, a random city is chosen that does not introduce a cycle. Observe that if one of the parents has a sequence of cities connected by short edges, they are likely to be carried forward to the offspring being constructed.

One can simplify the above crossover by choosing the successor cities from the two parents alternately. This is known as the *Alternating Edges Crossover* (AEX). A variation of this is to select a sequence of edges from one parent before choosing some from the other. This is known as the *Subtour Selection Crossover* (SSX). One can observe that this is similar to the PMX crossover with path representation. A check for cycle formation has still to be kept in all these approaches though.

In summary, the *TSP* problem is one of those problems in which defining crossover functions is not a straightforward process. The intuition behind *GAs* is that the offspring have a chance of inheriting and combining good components from the parents. In *TSP,* the good components are tour segments. But the shortest tour segments may not add up to a valid tour. If this happened, a greedy tour construction would have worked.

Box 4.1: A Note on Innovation and Creativity

Over a period of 3.5 billion years, nature has produced a vast plethora of designs for life through a process of recombination and selection. Starting presumably with a few simple forms, life has diversified into a number of species, with a large number of individuals within the species. Life forms display a bewildering diversity, occupying all nooks and corners of the earth. The number of living creatures living can safely be said to be unknown. Even the number of species is not known, and is variously said to be anywhere between 3 to 100 million. The world around us is a result of billions of generations of recombination and selection over a population of billions.

In contrast, human endeavour is relatively short termed. Our problem solving efforts are focused on speed and minimal use of resources. There has been considerable evidence that the completely unconstrained methods employed by nature are too slow to solve our problems (Grand, 1998; Goldberg, 2002). If we are to devise *GAs* that work, we have to put in more structure to guide the search. (Goldberg, 2002) defines *competent GAs* as follows.

"*The primary objective is to design what we call competent genetic algorithms. A GA is called competent if it can solve hard problems*

quickly, accurately, and reliably. Each of these can be quantified further, but qualitatively, hard problems are those that have large subsolutions that must be discovered whole, badly scaled subsolutions, many different local optima, a high degree of subsolution interaction, or a lot of external noise or stochasticity. In short, we are interested in designing effective solvers for the class of nearly decomposable problems (Simon, 1969). Speed, accuracy, and reliability requires that we get to near-global or high-quality solutions in times that grow as a polynomial function of the number of decision variables with high probability."

The competent *GA* is centred around decomposing problems into subproblems and finding subsolutions of those subproblems and combing them to "build" the solution. The subsolutions are called *Building Blocks* (BB). This is clearly different from the *GAs* described in this chapter because it looks for ways to *find and preserve BBs*. Unconstrained *GAs* will take too long[13] a time to do this. Clearly, the design of *GAs* that will find solutions in reasonable time is still in the realm of being an art form.

Having expressed the above *caveat,* we would like to observe that the dual process of *recombination and selection* is the only known approach to creativity. Moreover, reports on the study of human creativity have repeatedly thrown up the notion that innovation and creativity arise when humans combine and recombine ideas, often subconsciously; and have the ability to latch onto a good idea when it does emerge. We look at some of the evidence reported below.

The French mathematician Hadamard (1954) has attributed innovation and discovery to recombination in a pool of ideas "*We shall see a little later that the possibility of imputing discovery to pure chance is already excluded.…Indeed, it is obvious that invention or discovery, be it in mathematics or anywhere else, takes place by combining ideas*". He also quotes the French poet Paul Valéry who argued, "*It takes two to invent anything. The one makes up combinations: the other chooses … what is important to him in the mass of the things which the former has imparted to him*." This statement describes the two processes of recombination and selection succinctly.

Recent work employing techniques from psychology and brain imaging has shown that a key ingredient to creativity is the ability to handle many diverse kinds of ideas together, allowing them to cross fertilize each other. Our normal thinking processes are tuned to focusing on relevant matters and ignoring information that may be irrelevant to the task at hand. This process of ignoring irrelevant inputs is called *latent inhibition*. With high levels of latent inhibition, a person can pursue a task with a single mindedness that is a highly valued trait. The work done in Harvard University by a team led by Shelley Carson, a Harvard psychologist, reports a study that reveals that people with low levels of latent inhibition are the ones who can combine ideas creatively (Carson, 2003). Interestingly, low levels of latent inhibition are also related to psychotic conditions. "*Scientists have wondered for a long time why madness and creativity seem linked, particularly in artists, musicians, and writers. … Our research results indicate that low levels of latent*

inhibition and exceptional flexibility in thought predispose people to mental illness under some conditions, and to creative accomplishments under others." (Carson, 2003, Cromie, 2003).

Unlike the "normal" thought processes that are focused, people with low levels of latent inhibition have their minds flooded with many different ideas. If the person is able to handle this meaningfully, she can be creative; otherwise there is a danger of being labelled as 'mad'. Low latent inhibition, it seems, increases the available "*mental elements*"—thoughts, memories, and the like, or what Carson calls *"bits and pieces in the cognitive workspace*"—that supply the raw material for originality and novelty. Although too much material entering the "*cognitive working area*" might disorient psychotics, Carson wondered whether "*highly creative people could use those many bits and pieces in the cognitive workspace and combine them in novel, original ways*." (Lambert, 2004). "*Perhaps they do not dismiss as easily as the rest of us "irrelevant" ideas that pop into their heads, but instead entertain them long enough for one of them to connect with another thought that is kicking around—giving birth to a novel, creative idea. … Getting swamped by new information that you have difficulty handling, may predispose you to a mental disorder*," Carson says. "*But if you have high intelligence and a good working memory, you are more likely to be able to combine bits of new information in creative ways."* The "*high intelligence*" part of Carson's statement is key (Begley, 2005). High intelligence, she adds, "*should help you to better process the increasing information that goes along with low latent inhibition. To be creative, you can be bright and crazy, but not stupid*." Some minimal level of intelligence is, therefore, required for creativity. The reason is that in order to generate novel combinations, it helps to have a wealth of mental elements to work with. Without a *sufficient supply* of elements that can be combined in an original way, creativity is impossible (Begley, 2005).

This view is also expressed by Dean Simonton, a psychologist from the University of California, Davis. He says (Simonton, 2004) that creativity is analogous to variation and selection in Darwinian evolution. "*The creator must generate many different novelties from which are selected those that satisfy some intellectual or aesthetic criteria. … Underlying creativity, therefore, must be some process that generates these variations, made up of novel combinations of cognitive bits and pieces, as well as some way to choose among them. Creative people in diverse fields have said that this is exactly what it feels like they did. Chemist Linus Pauling described the need to "have lots of ideas and throw away the bad ones.… You aren't going to have good ideas unless you have lots of ideas and some sort of principle of selection*." Mathematician Henri Poincaré recalled the feeling that accompanied a creative breakthrough: "*Ideas rose in crowds; I felt them collide until pairs interlocked, so to speak, making a stable combination. By the next morning, I had established the existence of a class of (previously unknown mathematical) functions*." Einstein described how "*combinatory play seems to be the essential feature*" in creativity (Begley, 2005).

"*One eternal question is the relationship between madness and*

creativity. To be sure, most people who are mentally ill are not especially creative. But history is full of creative geniuses who were insane, including Vincent van Gogh and Robert Schumann; those who committed suicide, such as Ernest Hemingway and Virginia Woolf; or who were paranoid, such as Sir Isaac Newton. In the largest study ever conducted of the connection between creativity and madness, Arnold Ludwig analysed the biographies of about one thousand eminent men and women (Ludwig, 1995). He found that mental illness occurred more frequently in this group than it did in the general population. Specifically, 60 percent of the composers had psychological problems, as did 73 percent of the visual artists, 74 percent of the playwrights, 77 percent of the novelists and short-story writers, and 87 percent of the poets. But only about 20 percent of scientists, politicians, architects, and business people had even mild mental illness." (Begley, 2005).

It is the pathways and flexible connectivity in the human brain that determines what thought patterns can come together to "mate" and produce novel ideas.

Can we implement systems that are creative? The above discussion indicates that creativity involves the ability to make connections between seemingly unrelated ideas and selecting the useful combinations. Our exploration of *GAs* has just begun. The chromosomes we talk about are all of strings of the same length and a fixed schema. Though there has been some research with variable length strings and working with tree structures (see (Koza, 1992)), we are still far away from having integrated representations for all the problems our agents solve. As a result, each system that we develop for a specific task has its own representation. We will see in Chapter 15 that repositories (populations) of solutions can also be accumulated and used in the form of experience by memory-based agents. Even here, the schema of the solution is fixed and each system is designed for a specific task. Building integrated knowledge representation systems that an agent can use for different tasks is definitely a challenge for artificial intelligence research. Until we can do so, there is little chance of being able to make "long-distance" cross connections that seem to be the source of novel solutions. In some sense, our systems are tuned to work within a species, making changes within a rigid framework. To mimic, natural evolution would also result in accepting the slow process of change that nature has worked with. To emulate human creativity, we must be able to represent diverse ideas in a common pool and develop an ability to combine them in different ways and recognizing the good ideas when they do come along.

Box 4.2: Swarms in the Semiosphere

In genetic algorithms, we talk about systems (species, designs and solutions) that are made up of parts; and of processes that recombine the parts to produce new and novel systems. A question one might ask is what these processes are and how these parts come together. The

idea of emergent systems is that these parts come together by themselves in an environment where "*things that persist, persist and things that don't, don't*" (Grand, 2001). The idea behind *emergent systems* is that small simple parts *can* come together to form systems that *can* display complex behaviour.[14] And that simple parts come together and cooperate; not as a conscious process, but because their simple behaviours mesh together easily. Well known examples of complex systems emerging out of combinations of simple ones are ant colonies and human brains.

Jesper Hoffmeyer says[15], *"The emerging discipline of biosemiotics looks at how complex behaviour emerges when simple systems interact with each other through signs. Sign processes (or semiosis) are processes whereby something comes to signify something else to somebody (and 'somebody' here may be taken in the broadest sense possible, as any system possessing an evolved capacity for becoming alerted by a sign). The study of living systems from a semiotic (sign theoretic) perspective is called biosemiotics. According to biosemiotics, most processes in animate nature at whatever level, from the single cell to the ecosystem, should be analysed and conceptualized as sign processes. Biosemiotics is concerned with the sign aspects of the processes of life. In the biosemiotic conception, the life sphere is permeated by sign processes (semiosis) and signification. Whatever an organism senses also means something to it, food, escape, sexual reproduction, etc.; and all organisms are born into a semiosphere. The study of signs is known as semiotics, and the notion of a semiosphere (Lotman, 1990) refers to a world of meaning and communication: sounds, odours, movements, colours, electric fields, waves of any kind, chemical signals, touch, etc. The semiosphere poses constraints or boundary conditions to the Umwelts of populations since these are forced to occupy specific semiotic niches, i.e. they will have to master a set of signs of visual, acoustic, olfactory, tactile and chemical origin in order to survive in the semiosphere."*

The behaviour of termites and ants has long been understood as being coordinated through a system of signs. Higher level creatures like mammals routinely employ signs for communication. Many animals are known to mark their territory through various means. Humans, of course, have taken communication through symbols to a totally new level with the invention of language. Even outside of language, we have a rich and diverse system of communicating through signs, and by doing so, we often (consciously) organize ourselves into teams and mobs where the activity of one becomes a sign for others to interpret. One has only to watch a high level team sporting event, like soccer, to see how the players "read" the actions of their teammates as well as opponents. We have no difficulty in accepting the fact that mammals and humans communicate through signs. Biosemiotics however, shows that communication through signs may result in simpler entities coming together, and forming swarms that can be seen as a more complex system. Consequently, it emerges that not only are our social structures immersed in a semiosphere, but also that we ourselves *are* kinds of

semiospheres, in which the cells making up our different organs are bound together.

Hoffmeyer (1994) introduces a notion of *semetic interaction* (from Greek: *semeion* = sign, *etos* = habit) interaction between simple elements as a general phenomenon in the life sphere. Semetic interaction refers to the tendency of living systems to make signs based on any persistent regularity: wherever there has developed a *habit,* there will also exist an organism for whom this habit has become a sign. He illustrates this with the behaviour of termites in the following paragraph —"*When termites initiated nest constructing, the following sequence of events was observed by Grassé: First, hundreds of termites move around at random, while they exhibit a peculiar habit of dropping small pellets of masticated earth in places which are elevated a little bit from the ground. In spite of the disorganized character of this activity, it results in the formation of small heaps of salivated earth pellets. Second, these heaps of earth pellets are interpreted by the termites as a sign to release a new habit. Every time a termite meets a heap, it energetically starts building earth pellets on top of it. The effect of this activity will soon be the formation of a vertical column. The activity stops when the column has reached a certain species-specific height. Third, if the column has no immediate neighbours, the termites completely stop bothering about it. But if in an adequate distance there are one or more other columns, a third habit is released. The termites climb the columns and start building in a sloping direction towards the neighbouring column. In this way, the columns become connected with arches. The amazing fact is that through a seemingly haphazard sequence of events, a nest is actually produced which cannot but elicit the feeling in the observer, that there must have been some kind of intelligence behind it*."

The last statement resonates with Richard Dawkins who says that nature is "*The Blind Watchmaker*" who has fashioned the world. Looking at life from the perspective of a semiosphere, one can even think of the living forms achieving persistence by passing information about themselves in their genes. The notion is best described in another quote from the author (Hoffmeyer, 1994a)—"*To grasp the fundamentally semiotic character of animate nature, let us begin by considering the key process in life's peculiar way of persistence, heredity. Heredity is a phenomenon which is now rather well understood. And yet its real significance is rarely properly explained. The significance is this: Since living systems are mortal, their survival has to be secured through semiotic rather than physical means. Heredity is semiotic survival, i.e. survival through a message contained in the genome of a tiny template cell, the fertilized egg in sexually reproducing species. … In addition to this vertical semiotic system, i.e. genetic communication down through the generations, all organisms also partake in a horizontal semiotic system, i.e. communication throughout the ecological space. Every organism is born into a world of significance. Whatever an organism senses also mean something to it, food, escape, sexual reproduction, etc. This is one of the major insights brought to light through the pioneering work of Jakob von Uexküll: "Every action, therefore, that*

consists of perception and operation imprints its meaning on the meaningless object and thereby makes it into a subject-related meaning-carrier in the respective Umwelt (subjective universe)" (Uexküll, 1982).

One can then imagine all activity in terms of hierarchically composed semiotic systems, in which smaller parts come together (by themselves, through a long process of experimentation of recombination and selection) to form bigger and bigger "entities" that are more complex than the parts that make them up. The swarms like those of bees and ants may be made up of components that are similar, as are human organizations.

It must be observed though that even when *similar* members organize themselves into larger entities, they take up different roles. A typical corporate house, for example, will have CEOs, directors, managers, engineers, technicians, drivers, typists, cleaners, accountants and even lawyers. So they are similar as human beings, but functionally very different. In the same manner, we can think of our own body as made up of functionally different parts, which have some basic level of similarity, in that they all contain the same genetic code. One must keep in mind though that this analogy is a very loose one since bodies and industrial houses differ in many other ways. The point is that maybe one can loosen the strong sense of focused "self" that we all have and try and visualise the creation of life forms as something that arises out of a bottom up process as described by Hoffemeyer (Hoffmeyer, 1994) —"*The general principle which has made this bottom up or swarm conception of the body-mind biologically possible is the introduction of semiosis as the basic principle of life. By delegating semiotic competence to decentralised units, and ultimately to single cells, it becomes possible to ascribe intelligent behaviour to distributed systems. Stupid molecules become powerful tools as soon as they acquire semiotic quality, i.e. as soon as they are interpreted according to cellular habits. The transformation of molecules to signs opens for an unending semiogenic evolution based on semetic interaction patterns between entities at all levels. And through this evolution, the semiotic aspects of material processes gradually increase their autonomy, thereby creating an ever more sophisticated semiosphere. A semiosphere which finally had the power to create semiotic systems, such as thoughts and language, which are only in the slightest way dependent on the material world from which they were ultimately derived*."

4.5 Neural Networks

The discipline of biology has revealed that all life forms we see around us are *colonies of cells* that exist in a symbiotic equilibrium.

Life exists in many forms varying from the simple organisms made up from a few cells to very complex creatures hosting hundreds of *types* of cells. From the simple amoeba to the human being, there is an increase in complexity of behaviour accompanied with the faculty of awareness. Somewhere along this order of increasing complexity, the *notion of a self* emerges and creatures like