**CHAPTER TWO**

**LITERATURE REVIEW**

Timetabling is the procedure of assigning classes to time-slots and classrooms, subjected to various constraints. The common timetabling problem is described by Burke, Kingston, and de Werra (2004) as follows: A timetabling scheduling problem consists of four parameters:

* A finite set of times T.
* A finite set of resources R.
* A finite set of meetings M.
* A finite set of constraints C.

The task is to allocate T and R to M so as to satisfy C as much as possible [10]. For example in Examination timetabling scheduling problem, we have to assign various exams to different time periods in a way that satisfies a set of constraints and preferences. Many researchers worked on various timetabling problems like sports timetable scheduling, Railway timetable scheduling, and Teacher assignment problem and of course educational timetable scheduling. Timetabling generation is very useful in all domains also e.g. hospitals, transportation, sports etc.

School timetabling problem involves the scheduling for all the classes of a high school, also it is different from other scheduling problems, as it is weekly scheduling, normally most of the students will have completely full timetable with no free time between periods which means the time slots will be adjacent. It also avoids any teacher to meet more than one class at the same time. Here in the case of exam timetable, the exam room cannot  
be used by different subjects or exams at the same time slot. There is difference between school timetabling problem and university course timetabling problems because high schools students need to attend nearly every hour of the day and their teachers have teaching load usually higher than university teachers.[34][64]D.Abramson(1991), D.Abramson et al (1999) proposed simulated annealing[66][20]M.

School Timetabling is one of the tedious and frustrating computationally difficult problems in scheduling. It is known to be a non-polynomial complete problem i.e. there is no known efficient way to locate a solution. Also, the most striking characteristic of NP-complete problems is that, no best solution to them is known. In time tabling, the aim is to find suitable time slots for a number of class that require limited resources respecting some constraint. Depending on the nature of the problem, the constraints in the problem can vary and there may be many different objectives. For example, in some cases the objective may be to minimize the length of the total time period over which the class are to be scheduled, in other cases the objective may be to find a feasible solution subject to a fixed total time period and several other constraints. Yet in others the objective can be to find a solution in which least number of constraints are violated.

Timetabling problems can arise in many different settings, but generally it refers to the timetabling at educational institutions. The significance of this problem is mainly due to the difficulty of constructing a feasible timetable that satisfies the preferences of the administration, the instructors, and the students. In certain cases, it may be extremely difficult even to find a single feasible solution.

It is not possible to formulate a general model that is applicable for all cases, since every educational institution has its own special constraints and objectives. For example, for a secondary school, there should not be any gap between the class meetings, on the other hand this is allowed, and in some cases encouraged, in a university.

Timetabling problems can be grouped into two types: Examination and Course Scheduling problems. Examination scheduling problems deal with assigning the examinations over an examination period subject to several constraints. The objective can be assignment of the examinations to a minimum number of periods without any conflict. In other problems, the number of periods are fixed and the objective is to optimize a preference function that is based on the preferences of the administration, the instructors, and the students. Some of these preferences may be:

* Maximizing the time interval between the two consecutive examinations of a student.
* Scheduling the examinations with large number of students in earlier time slots, to allow more time for grading.

In course scheduling, the time period is fixed and, in general, it is one week. The objective is to find a course schedule that is feasible with respect to a number of constraints. Course scheduling and examination scheduling problems have both similar and differing characteristics. For example, in both problems a student cannot take more than one examination or attend more than one class meeting at a time. On the other hand, in examination scheduling problems there may not have a fixed time period, however all course scheduling problems are for fixed time periods

**2.1 CULTURAL ALGORITHM**

The Cultural Algorithm is inspired by the principle of cultural evolution. Culture includes the habits, knowledge, beliefs, customs, and morals of a member of society. Culture does not exist independent of the environment, and can interact with the environment via positive or negative feedback cycles.

Cultural Algorithms have been developed so as to model the evolution of the cultural component of an evolutionary computational system over time as it accumulates experience (Reynolds & Chung, 1996). As a result, Cultural Algorithms can provide an explicit mechanism for global knowledge and a useful framework within which to model self-adaptation in an evolutionary system.

Cultural Algorithms are based on knowledge of an evolutionary system that implements a dual mechanism of inheritance. This mechanism allows the Cultural Algorithms to explore as much microevolution as macroevolution. Microevolution is the evolution that happens in the population level. Macroevolution occurs on the culture itself, i.e. the belief space evolution.

The belief space is the place where the information on the solution of the problem is refined and stored. It is acquired through the population space over the evolutionary process. The belief space has the goal to guide individuals in search of better regions. In the Cultural Algorithm evolution occurs more quickly than in population without the mechanism of macroevolution. The characteristics and behaviors of individuals are represented in the Population Space and as mentioned earlier the population space can support any population-based computational model such as Genetic Algorithms among others (Jin & Reynolds, 1999). The communications protocols dictate the rules about individuals that can contribute to knowledge in the Belief Space (function of acceptance) and how the Belief Space will influence new individuals (Function of Influence).

The two most used ways to represent knowledge in the belief space are (Reynolds & Peng, 2004) Situational Knowledge and Normative Knowledge. Situational Knowledge represents the best individuals found at a certain time of evolution and it contains a number of individuals considered as a set of exemplars to the rest of the population.

The number of exemplars may vary according to the implementation, but it is usually small. For example, the structure used to represent this type of knowledge is shown in Figure 3. Each individual is stored within its parameters and fitness value (Iacoban et al. 2003). The Situational Knowledge is updated when the best individual of the population is found. This occurs when its fitness value exceeds the fitness value of the worst individual stored.

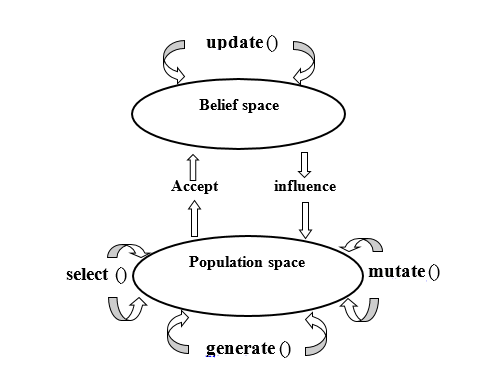
Normative Knowledge represents a set of intervals that characterize the range of values given by the features that make the best solutions (Iacoban et al., 2003). The adjustment of the range of Normative Knowledge varies according to the best individual.

That is, if the individual was accepted by the acceptance function and its range is less than the range stored in the belief space, the range is adjusted, and vice versa. The resolution of problems produces experiences from individuals in the population space, which are selected to contribute to the acceptance by the belief space, where the knowledge is generalized and stored. In the initial population, the individuals are evaluated by the fitness function. Then, the information on the performance of the function is used as a basis for the production of generalizations for next generations. The experiences of the individuals selected will be used to make the necessary adjustments on the knowledge of the current belief space.

**2.1.1 CULTURAL ALGORITHM FRAMEWORK**

The Cultural algorithm comprises of a knowledge component called belief space and the population space. The belief space has four subdivisions which are situational knowledge, normative knowledge, topographical knowledge and history knowledge. The following functions are performed considering the framework of the algorithm:

* **Update function:** a process that affects the computational efficiency of the algorithm. The belief space is updated after each solution is selected and accepted. To update the belief space every solution generated is compared to the ones already existing in the belief space to see if the newly generated is better than the ones in the belief space using their fitness function, then the newly generated solution replaces the solution that is less fit. The belief space is said to be updated when this process occurs.
* **Influence function:** this function decides which knowledge source influence individuals. The knowledge represented in the belief space can be used to influence the creation of offspring through an influence function. This knowledge controls the population component.
* **Accept function:** a selected group of individuals is obtained with the accept function which is applied to the entire population. The main idea behind this approach is to preserve beliefs of the individuals that are socially accepted and discard unacceptable beliefs. The cultural algorithm is applied for optimization, the acceptable beliefs can be seen as constraints that directs the population at the evolution level.
* **Selection function:** it is performed considering the main population size. Each individual is confronted against other individuals which are randomly chosen from the main population. The selection operator is modified by the influence function, this function acts in a way that the individuals resulting from the application of the operator tend to approach the desirable behavior while staying away from the undesirable behaviors.
* **Objective function:** it is a result of the performance function of an individual in the population space when selection is carried out. It is possible that the performance of an individual increases. Individuals are first evaluated using a performance function which represents the problem-solving experience of an individual.
* **Generate function:** its function is to generate new individuals. When individuals are randomly generated from the population space, they form the offspring of the population. For children generation, the variation of the differential evolution algorithm is influenced by the belief space. The main influence function is responsible for choosing the knowledge source to be applied to the variation operator of newly generated children.



**Figure 2.1: Cultural Algorithm Framework.**

**2.2 CONSTRAINTS**

Soft and hard constraints  are fundamental elements behind not only creating a great timetable, but also a working one. The make up of a timetable (any timetable) consists of a variety of constraints that are either hard or a soft, but what are they?, what is the difference? and how do I use them?

Let starts with hard constraints, these are the fundamental elements that you and the software use to create the timetable and must be abided by. For example, the number of hours in the teaching day is likely to be a hard constraint. If you teach Monday – Friday 09-17:00, then the timetable must ensure that all timetable activities are placed within this time frame. In this scenario a timetable booking placed on a Thursday at 18:00 or a Saturday at 12:00 will not be of any use, therefore the software and those who use it must abide by this constraint in order to create a working timetable i.e. a hard constraint.

Each hard constraint, is an element of the timetable that is equally weighted and must be abided by. Simply put – if they are broken, the timetable will not work. I have provided below the key hard constraints that typically make up a teaching timetable, although there can be more depending on how your institution works!

**Number of Teaching Spaces/Rooms** – You can only book as many rooms as you have available, if the number of timetable activities exceeds your teaching room availability then you have a problem!

**The Teaching Week** – As discussed, if you have a strict teaching week then there is no use timetabling outside of these times.

**Number of Weeks in the Semester** – As with the teaching week, there is no use timetabling activities outside of your institution's planned teaching weeks if all teaching activities must take place within them.

**Student Clash Checks** – These can be a mixture of both hard and soft constraints, however if a student must attend two lectures then the timetable should recognize that they cannot happen at the same time – i.e. a hard constraint.

**Staff Clash Checks** – As with students, staff can’t be in two places at the same time therefore again the timetable must recognize this.

**Teaching Room Capacity** – If a room has a 100 capacity then only 100 students can fit into this room, therefore this is typically a hard constraint. However, there can be some flexibility with this if you can accurately predict a certain percentage will not attend and therefore are willing to allow for example a 5% margin, as this would allow an extra 5 students into this room.

Each of these hard constraints are built around the same principles that govern most aspects of our life – space and time. You will be able to create a working timetable, providing you have resources available to accommodate the timetable activities whilst abiding by the hard constraints.

The soft constraints, are the other side of the coin and are made up of the elements which can turn your timetable from a working timetable into a great timetable. Of course, a “great” timetable is dependent on what you are looking to achieve and what resources you have available, but the principle of a soft constraint is that you are putting a constraint into the timetable that will help to improve your timetable that isn't a necessity.

For example, student feedback may indicate that the students want to have an hour lunch break within their timetable during each day of the teaching week. This request isn't a necessity, but it is a want and therefore you can test to see whether this is feasible by building it into your timetable as a soft constraint. The difference between this soft constraint and a hard constraint, is that the timetabling software (and you) must be able to recognize that if there is no other option (or there is a better option) this constraint can be ignored.

What do I mean by a better option? Well, one of the fundamental issues with soft constraints is that you are likely to have more than one. For example, students may want to have a lunch hour, but also a teaching day that does not exceed 6 hours, as well as have no more than 4 hours of lectures in a row, plus your staff also don’t want to teach more than three hours’ worth of activities in a row plus want to have a research day each week.

Each of these is a soft constraint, as if they are all ignored, and then you still have a timetable that works – just not one that makes very many people happy! However, you may well not be able to accommodate all of these requests without running out of space and time. Therefore, these soft constraints are typically put into timetabling software and ranked according to their importance.

By ranking the soft constraints you are telling the software how important each of these soft constraints are in comparison to each other. In doing so the timetabling software will try to abide by all the hard and soft constraints when creating a timetable, but if they cannot all be fulfilled the software will try to satisfy as many of the higher ranked soft constraints as possible whilst also abiding by the hard constraints – the hard constraint must also be abided by!

Using the same example, you could rank the soft constraints as shown below:

* **Student Lunch Hour** – 1 hour between 12-14:00 – Rank 7
* **Student Teaching Day** – Max 6 Hours – Rank 8
* **Student Consecutive Teaching Hours** – Max 4 Hours – Rank 4
* **Staff Consecutive Teaching Hours** – Max 3 hours – Rank 5
* **Staff Teaching Week** – Max 4 days – Rank 6

By ranking these soft constraints, you are telling the timetabling software what you perceive to be the order of importance (i.e. how they are ranked). By doing so, this helps to focus the software's auto scheduling function towards creating a working timetable that reflects the best option available. If not all the constraints can be met, those with lowest rank will be ignored until a suitable time/space can be timetabled.

The types of soft constraints that make up a timetable are completely dependent upon what you are trying to achieve and what resources you have available, therefore there are a huge number of permeations that you can build into a timetable that creates a great timetable for your institution. Some other examples of soft constraints include:

**Teaching Space Capacity** – Timetable activities into rooms that closely match their class size

**A Preferred Teaching Week** – Restrict the number of bookings at certain times or on certain days.

**Maximum Gap Between Lectures** – Limit the number of free hours students (and staff) have between lectures.

**Teaching Room Zoning** – Concentrate certain Departments bookings within specific areas of an estate. For example Department A would prefer to have their teaching in the 3 buildings the department is based in or close to.

**2.3 GENETIC ALGORITHM**

Genetic Algorithms are generally attributed to Holland and his students in the 1970s, although evolutionary computation dates back further (refer to Fogel (1998) for an extensive review of early approaches). Genetic Algorithms are stochastic meta-heuristic that mimic some features of natural evolution. Genetic Algorithms were not intended for function optimization, as discussed by De Jong (1993). However, slightly modified versions proved very successful. Many examples of successful implementations can be found in Bäck, Chaiyaratana and Zalzala (1997) and others.

In a nutshell, Genetic Algorithms mimic the evolutionary process and the idea of the survival of the fittest. Starting with a population of randomly created solutions, better ones are more likely to be chosen for recombination into new solutions, i.e. the fitter the solution the more likely it is to pass on its information to future generations of solutions.

In addition to recombining solutions, new solutions may be formed through mutating or randomly changing old solutions. Some of the best solutions of each generation are kept whilst the others are replaced by the newly formed solutions.

The process is repeated until stopping criteria are met. However, constrained optimization with Genetic Algorithm remains difficult. The root of the problem is that simply following the building block hypothesis, i.e. combining good building blocks or partial solutions to form good full solutions, is no longer enough, as this does not check for constraint consistency. To solve this dilemma, many ideas have been proposed. A good overview of these and most other techniques can be found in Michalewicz (1995).

Beasley and Chu (1996) bias the way the solution space is sampled towards feasible areas. We will use such a scheme in our decoder. Repairing infeasible solutions also has its drawbacks. Firstly, it is often as difficult to repair an invalid solution as it is to find a good feasible solution. Secondly, repeated repair might lead to a buildup of poor material within the population, as there is not enough incentive for development. Finally, repair routines are typically time consuming and it is arguable that this time is better spent on a more direct search of the solution space. The approach presented here is the combination of an indirect genetic algorithm with a separate heuristic decoder function.

The Genetic Algorithm tries to find the best possible ordering of the nurses, which is then fed into a greedy decoder that builds the actual solution. One way of looking at this decoder is as an extended fitness function calculation, i.e. the decoder determines the fitness of a solution once it has built a schedule from the permutation of nurses. One advantage of this approach is that all problem specific information is contained within the decoder, whilst the Genetic Algorithms can be kept canonical.

The only difference from a standard Genetic Algorithms is the need for permutation-based crossover and mutation operators (Goldberg, 1989). This should allow for easy adaptation to different problems.

**2.3.1 GENETIC OPERATORS**

To implement the GA process, many factors should be considered such as the representation scheme of chromosomes, the mating strategy, the size of population, and the design of the genetic operators such as selection, mutation and recombination (Ku & Lee, 2001).

1. **Selection:** is an operator that prevents low fitness individuals from reproduction and permits high fitness individuals to offspring more children to improve average fitness of population over generations.
2. **Crossover:** operator that will allow mating of randomly chromosomes, an operator that mixes the chromosomes of two individuals. Typically two children are generated by applying this operator, which are similar to the parents but not same. Crossover causes a structured, yet randomized exchange of genetic material between solutions, with the possibility that the “fittest” solutions generate “better” ones. A crossover operator should preserve as much as possible from the parents while creating an offspring.
3. **Mutation:** cloning a particular parent, having same attributes, it introduces totally new individuals to population. It helps extend the domain of search and will restrain the diversity of the population. Mutation involves the modification of each bit of an individual.

**2.4. STRENGTHS OF GENETIC ALGORITHMS**

The first and most important point is that genetic algorithms are intrinsically parallel. Most other algorithms are serial and can only explore the solution space to a problem in one direction at a time, and if the solution they discover turns out to be suboptimal, there is nothing to do but abandon all work previously completed and start over. However, since GAs has multiple offspring, they can explore the solution space in multiple directions at once. If one path turns out to be a dead end, they can easily eliminate it and continue work on more promising avenues, giving them a greater chance each run of finding the optimal solution (Adam, 2004; John, 1992).

However, the advantage of parallelism goes beyond this. Consider the following: All the 8-digit binary strings (strings of 0's and 1's) form a search space, which can be represented as \*\*\*\*\*\*\*\* (where the \* stands for "either 0 or 1"). The string 01101010 is one member of this space. However, it is also a member of the space 0\*\*\*\*\*\*\*, the space 01\*\*\*\*\*\*, the space 0\*\*\*\*\*\*0, the space 0\*1\*1\*1\*, the space 01\*01\*\*0, and so on. By evaluating the fitness of this one particular string, a genetic algorithm would be sampling each of these many spaces to which it belongs. Over many such evaluations, it would build up an increasingly accurate value for the average fitness of each of these spaces, each of which has many members. Therefore, a GA that explicitly evaluates a small number of individuals is implicitly evaluating a much larger group of individuals - just as a pollster who asks questions of a certain member of an ethnic, religious or social group hopes to learn something about the opinions of all members of that group, and therefore can reliably predict national opinion while sampling only a small percentage of the population. In the same way, the GA can "home in" on the space with the highest-fitness individuals and find the overall best one from that group. In the context of evolutionary algorithms, this is known as the Schema Theorem, and is the "central advantage" of a GA over other problem-solving methods (John, 1992; Mitchell, 1996; Goldberg, 1989).

Due to the parallelism that allows them to implicitly evaluate many schemas at once, genetic algorithms are particularly well-suited to solving problems where the space of all potential solutions is truly huge - too vast to search exhaustively in any reasonable amount of time. Most problems that fall into this category are known as "nonlinear". In a linear problem, the fitness of each component is independent, so any improvement to any one part will result in an improvement of the system as a whole. Needless to say, few real-world problems are like this. Nonlinearity is the norm, where changing one component may have ripple effects on the entire system, and where multiple changes that individually are detrimental may lead to much greater improvements in fitness when combined. Nonlinearity results in a combinatorial explosion: the space of 1,000-digit binary strings can be exhaustively searched by evaluating only 2,000 possibilities if the problem is linear, whereas if it is nonlinear, an exhaustive search requires evaluating 21000 possibilities - a number that would take over 300 digits to write out in full (Adam, 2004).

Fortunately, the implicit parallelism of a GA allows it to surmount even this enormous number of possibilities, successfully finding optimal or very good results in a short period of time after directly sampling only small regions of the vast fitness landscape (Forrest, 1993). For example, a genetic algorithm developed jointly by engineers from General Electric and Rensselaer Polytechnic Institute produced a high-performance jet engine turbine design that was three times better than a human-designed configuration and 50% better than a configuration designed by an expert system by successfully navigating a solution space containing more than 10387 possibilities. Conventional methods for designing such turbines are a central part of engineering projects that can take up to five years and cost over $2 billion; the genetic algorithm discovered this solution after two days on a typical engineering desktop workstation (John, 1992).

Another notable strength of genetic algorithms is that they perform well in problems for which the fitness landscape is complex - ones where the fitness function is discontinuous, noisy, changes over time, or has many local optima. Most practical problems have a vast solution space, impossible to search exhaustively; the challenge then becomes how to avoid the local optima - solutions that are better than all the others that are similar to them, but that are not as good as different ones elsewhere in the solution space. Many search algorithms can become trapped by local optima: if they reach the top of a hill on the fitness landscape, they will discover that no better solutions exist nearby and conclude that they have reached the best one, even though higher peaks exist elsewhere on the map (Adam, 2004).

Evolutionary algorithms, on the other hand, have proven to be effective at escaping local optima and discovering the global optimum in even a very rugged and complex fitness landscape. (It should be noted that, in reality, there is usually no way to tell whether a given solution to a problem is the one global optimum or just a very high local optimum. However, even if a GA does not always deliver a provably perfect solution to a problem, it can almost always deliver at least a very good solution.) All four of a GA's major components - parallelism, selection, mutation, and crossover - work together to accomplish this. In the beginning, the GA generates a diverse initial population, casting a "net" over the fitness landscape. (Koza et. al., 2003) compares this to an army of parachutists dropping onto the landscape of a problem's search space, with each one being given orders to find the highest peak.) Small mutations enable each individual to explore its immediate neighborhood, while selection focuses progress, guiding the algorithm's offspring uphill to more promising parts of the solution space (John, 1992).

However, crossover is the key element that distinguishes genetic algorithms from other methods such as hill-climbers and simulated annealing. Without crossover, each individual solution is on its own, exploring the search space in its immediate vicinity without reference to what other individuals may have discovered. However, with crossover in place, there is a transfer of information between successful candidates - individuals can benefit from what others have learned, and schemata can be mixed and combined, with the potential to produce an offspring that has the strengths of both its parents and the weaknesses of neither. This point is illustrated in Koza et. al. (1999), where the authors discuss a problem of synthesizing a low pass filter using genetic programming. In one generation, two parent circuits were selected to undergo crossover; one parent had good topology (components such as inductors and capacitors in the right places) but bad sizing (values of inductance and capacitance for its components that were far too low). The other parent had bad topology, but good sizing. The result of mating the two through crossover was an offspring with the good topology of one parent and the good sizing of the other, resulting in a substantial improvement in fitness over both its parents.

The problem of finding the global optimum in a space with many local optima is also known as the dilemma of exploration vs. exploitation, "a classic problem for all systems that can adapt and learn" (John, 1992). Once an algorithm (or a human designer) has found a problem-solving strategy that seems to work satisfactorily, should it concentrate on making the best use of that strategy, or should it search for others? Abandoning a proven strategy to look for new ones is almost guaranteed to involve losses and degradation of performance, at least in the short term. But if one sticks with a particular strategy to the exclusion of all others, one runs the risk of not discovering better strategies that exist but have not yet been found. Again, genetic algorithms have shown themselves to be very good at striking this balance and discovering good solutions with a reasonable amount of time and computational effort (Adam, 2004).

Another area in which genetic algorithms excel is their ability to manipulate many parameters simultaneously (Forrest, 1993). Many real-world problems cannot be stated in terms of a single value to be minimized or maximized, but must be expressed in terms of multiple objectives, usually with tradeoffs involved: one can only be improved at the expense of another. GAs are very good at solving such problems: in particular, their use of parallelism enables them to produce multiple equally good solutions to the same problem, possibly with one candidate solution optimizing one parameter and another candidate optimizing a different one (Haupt et.al., 1998), and a human overseer can then select one of these candidates to use. If a particular solution to a multi-objective problem optimizes one parameter to a degree such that that parameter cannot be further improved without causing a corresponding decrease in the quality of some other parameter, that solution is called Pareto optimal or non-dominated (Coello, 2000).

Finally, one of the qualities of genetic algorithms which might at first appear to be a liability turns out to be one of their strengths: namely, GAs know nothing about the problems they are deployed to solve. Instead of using previously known domain-specific information to guide each step and making changes with a specific eye towards improvement, as human designers do, they are "blind watchmakers" (Dawkins, 1996); they make random changes to their candidate solutions and then use the fitness function to determine whether those changes produce an improvement.

The virtue of this technique is that it allows genetic algorithms to start out with an open mind, so to speak. Since its decisions are based on randomness, all possible search pathways are theoretically open to a GA; by contrast, any problem-solving strategy that relies on prior knowledge must inevitably begin by ruling out many pathways a priori, therefore missing any novel solutions that may exist there (Koza et. al., 1999). Lacking preconceptions based on established beliefs of "how things should be done" or what "couldn't possibly work", GAs do not have this problem. Similarly, any technique that relies on prior knowledge will break down when such knowledge is not available, but again, GAs are not adversely affected by ignorance (Goldberg, 1989). Through their components of parallelism, crossover and mutation, they can range widely over the fitness landscape, exploring regions which intelligently produced algorithms might have overlooked, and potentially uncovering solutions of startling and unexpected creativity that might never have occurred to human designers. One vivid illustration of this is the rediscovery, by genetic programming, of the concept of negative feedback - a principle crucial to many important electronic components today, but one that, when it was first discovered, was denied a patent for nine years because the concept was so contrary to established beliefs (Koza et. al., 2003). Evolutionary algorithms, of course, are neither aware nor concerned whether a solution runs counter to established beliefs - only whether it works.

**2.5 HARD AND SOFT CONSTRAINTS**

A timetable is essentially a schedule which must suit a number of constraints. Constraints are almost universally employed by people dealing with timetabling problems. Constraints in turn, are almost universally broken into two categories soft and hard constraints (Burke and Ross, 1995).

Soft constraints are those whose violence should be minimized in order to produce the best timetable. Unlike hard constraints are not essential, but toll constraints satisfaction measure of how good a timetable is. Soft constraints vary greatly between institutions. Some common ones are:

1. Time assignment- A lecture may need to be scheduled in a specific period.
2. Time constraints between events- One lecture may need to be scheduled before/after another.
3. Spreading events over time- Students should not have lectures in consecutive periods or two lectures within periods of each other.
4. Resource assignment- A lecture must be scheduled into a specific room. Hard constraints are constraints, of which, in any working timetable, there will be no breaches. For example, a student cannot be in two places at once. Soft constraints are constraints which may be broken, but of which breaches must be minimized. For example, no student should have no contention. In addition to constraints, there are a number of exceptions which must be taken into consideration when constructing an automated timetabling system. Many constraints involved in lecture scheduling vary from university to university, department by department and faculty to faculty.

However, it is generally accepted that the following three constraints are fundamental to any timetabling problem:

1. No entry must be demanded to be more than one place at the same time. In lecture timetabling, this would mean that no student should be made to sit more than one lecture at any one time.
2. For each period in the timetable, the resources demand made by the events scheduled for that period must not exceed the resources available. In lecture timetabling, it is important not to schedule more than one lecture in a room than there are desks.
3. No student should have to sit for lectures in adjacent periods.

A room can be assigned more than one examination during the same session depending on the space available but not lectures. On the other hand, an examination can be split into more than one room depending on the size of the course and capacity of available space. Distance between lecture rooms is not important since there is a gap of at least one hour between sessions. Lecturers are specified by departments after the release of the timetable so that the planning is not part of this automatic process. The constraints implemented in this algorithm are as follows:

**2.5.1 HARD CONSTRAINTS**

1. No entity is demanded to be at more than one place at a time, i.e. no student can sit for more than one lecture at any one time.
2. For each period in the timetabling, the resources demanded made by the events scheduled for the period must not exceed the resources available, i.e. it is important not to schedule more students in a room than there are desks.
3. Some courses are scheduled in blocks of more than one hour, these restrictions must be respected.
4. No room can occupy more than one lecture at the same time.

**2.5.2 SOFT CONSTRAINTS**

1. No student should have to take two lectures in an adjacent period except it is necessary.
2. Lectures of students must be evenly spread.
3. As much as possible, minimize the use of early morning (7:00am) lunch hours (13-14) and late evening hours (18-20).
4. Specifically minimize the use of Friday 13-14 hour and 18-20 hour slots to allow for Muslim prayers and Adventists Sabbath day respectively.
5. As much as possible, evening lectures starting from 18 to 20 hours should be assigned to rooms with standby generators so as to minimize loss of lecture hours due to frequent power cuts.

**2.6 RELATED WORKS**

Several works have approached the timetabling problems. Oliveira (oliveira and reis, 2000) presents a language for representation of the timetabling problem, theUniLang, UniLang intends to be a standard suitable as input language for anytime ttabling system. It enables a clear and natural representation of data, constraints, quality measures and solutions for different timetrabling (as well as related) problems, such as school timetabling, university timetabling and examination scheduling.

Grobner(Grobner, 1997) presents an approach to generalize al the timetabling problems, describing the basic structure of this problem. Grobner proposes a generic language that can be used to describe timetabling problems and its constraints.

Chan(Chan, 1997) discusses the implementation of two genetic algorithm used to solve class-teacher timetabling problem for small schools.

Fang(Fang, 1994) in his doctoral thesis, investigate the use of genetic algorithm to solve a group of timetabling problems. He presents a framework for the utilization of genetic algorithms in solving of timetabling problems in the context of learning institutions. This framework has the following important points, which give you considerable flexibility, a declaration of the specific constraints of the problem and use of a function for evaluation of the solutions, advising the use of a genetic algorithm, since it is independent of the problem for its resolution.

Fernandez (Fernandez, 2002) classified the constraints of class-teacher timetabling problem in constraints strong and week. Violations to strong constraint (such as schedule a teacher in two classes at the same time) result in an invalid timetable. Violation to weak constraints result in a valid timetable, but affect the quality of the solution (for example, the preference of teachers for certain hours). The proposed algorithm, evolutionary, has been tested in a university comprising 109 teachers, 37 rooms, 1131 a time interval of one hour each and 472 classes. The algorithm proposed in resolving the scheduling without violating the strong constraint in 30% of executions.

Eley(Eley, 2006) in PATA’06 presents a solution to the exam timetable problem, formulating it asa problem of combinational optimization, using algorithms Ant, to solve. By the analysis of the results obtained by the various works published, we can say that the automatic generation of schedules is capable of achieving. Some works show that when compared with the schedule manuals in institutions of learning, the times obtained by algorithms for solving the class-teacher timetabling problem are of better quality since it uses some function of evaluation (Jose Joaquim Moreira, 2008).

In timetable scheduling problems, examination subjects must be slotted to certain times that satisfy several of constraints. They are NP-completeness problems, which usually lead to satisfactory but sub optional solutions. As PSO has many successful applications in continuous optimization problems, the main contribution of this paper is to utilize PSO to solve the discrete problem of timetable scheduling. Experimental results confirm that PSO can be utilized to solve discrete problem as well. (Singh et. Al .2005).

**2.7 SOFTWARE COMPLEXITY METRICS**

Software complexity is widely regarded as an important determinant of software maintenance costs (Boehm 1981). Software complexity is assumed to be a multidimensional construct (Wake and Henry 1988). The complexity of a program depends upon its magnitude, the complexity of its control structure, and the complexity of its data flows (Basili and Hutchens 1983). Other researchers add other factors to this list, such as the degree of modularity (Bowen 1978).

Munson and Khoshgoftaar (1989) conclude that four or five such complexity factors suffice to describe the multi-dimensional complexity of a program. On the other hand, correlations among the various kinds of metrics have generally proven to be extremely high (Li and Cheung 1987).

Complexity measures can be used to predict critical information about reliability and maintainability of software systems from automatic analysis of the source code. Complexity measures also provide continuous feedback during a software project to help control the development process. During testing and maintenance, they provide detailed information about software modules to help pinpoint areas of potential instability (Olabiyisi & Ganiyu, 2007). There are a number of ways to quantify complexity in a program. The best-known metrics, which provide such feature, are McCabe’s (1976) Cyclomatic number and Halstead’s (1977) volume. These metrics have been extensively validated and compared (Aggarwal, *et al*, 2002).

This section describes the performance metrics used in analyzing the performance of the developed automated nurse roster application using a genetic-based Cultural algorithm.

For the course of this project, we will be using three metrics to determine the performance of our cultural algorithm program. These metrics are:

1. Cyclomatic Complexity
2. Halstead Complexity
3. Maintainability Index

**2.7.1 CYCLOMATIC COMPLEXITY**

Cyclomatic complexity can be considered as a broad measure of soundness and confidence for a software program. Introduced by McCabe (1976) it measures the number of linearly independent paths through a program module. Cyclomatic Complexity is a measure of the logical complexity of an algorithm. It measures the amount of decision logic in a source code function.

Simply put the more decisions that have to be made in code, the more complex it is. It evaluates the risk associated with a program in terms of testability. Cyclomatic complexity is the most widely used member of a class of static software metrics. This measure provides a single ordinal number that can be compared to the complexity of other programs. It is often used in connection with other software metrics.

**Table 2.1 Cyclomatic Complexity and its risk evaluation**

|  |  |
| --- | --- |
| **CYCLOMATIC COPLEXITY** | **RISK EVALUATION** |
| 1-10 | A simple Program without much risk |
| 11-20 | More complex, moderate risk |
| 21-50 | The Program is complex, high risk |
| Greater than 50 | Untestable Program, very high risk |

It is calculated using the formula:

Where

n = Number of Nodes.

**2.7.2 HALSTEAD COMPLEXITY MEASURE**

Halstead theory of software metric is a technique used to measure complexity in a software program and the amount of difficulty involved in testing and debugging the software.

Halstead uses the number of distinct operators, distinct operands in a program to develop expressions for the overall program length, volume and the number of remaining defects in a program.

Halstead argues that algorithms have measurable characteristics analogous to physical laws. His model is based on 4 different parameters: The number of distinct operators (n1), the number of distinct operands (n2); the total number of occurrences of operators (N1), and the total number of occurrences of operands (N2).

**2.8.3 MAINTAINABILITY INDEX**

Maintainability index (M.I) calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means a better maintainability. It is calculated using the following formula:

**M.I** = 171 - 5.2 \* ln(V) – 0.23 \* (G) – 16.2 \* ln(LOC)

Where V = Halstead Volume is the size of implementation of an algorithm,

G = the cyclomatic complexity

LOC = the source line of codes.