4-30-2010

A Hybrid Genetic Algorithm for the Student-Aware University Course Timetabling Problem

Ernesto Ferrer Queiros Nunez
Macalester College

A Hybrid Genetic Algorithm for the Student-Aware University Course Timetabling Problem

Submitted to the Department of Mathematics,
Statistics and Computer Science in partial
fulfillment of the requirements for the degree of
Bachelor of Arts

By
Ernesto Ferrer Queiros Nunes

Advisor:	Prof. Elizabeth Shoop, MSCS Department
Second Reader:	Prof. Susan Fox, MSCS Department
Third Reader:	Prof. Vittorio Addona, MSCS Department

Macalester College

April 30, 2010

Abstract

Traditionally, academic institutions schedule courses using constraints that ensure that instructors and courses do not overlap in available rooms and time periods; students' planning needs are rarely taken into account. This problem becomes particularly acute for students in liberal arts institutions, because they have multiple graduation requirements in addition to their chosen academic program. My research builds on the University Course Timetabling Problem (UCTP) to include students' scheduling needs. This approach to the UCTP problem uses a combination of a genetic algorithm and case-based reasoning.

To improve the performance of the genetic algorithm, I use a group-based genetic algorithm to place courses into distinct rooms and a self-fertilization crossover operator to avoid adding duplicate courses to the timetable during crossover. Case-based reasoning serves as a system to store and retrieve previous solutions. If a new problem is given, instead of using a genetic algorithm to produce timetables from scratch, the system first checks if the case-base has a previous timetable that solves the problem. I generate test data using knowledge of class scheduling at Macalester College. Although the student constraint is harder to satisfy than the instructor constraint, my results show that the genetic algorithm improves the fitness of the population for each generation, and it returns a feasible solution, even for the most constrained benchmarks.

Acknowledgment

First and foremost I would like to thank Professor Elizabeth Shoop for her tireless editing and advising efforts. I would like to thank Professor Susan Fox for the technical help with both the Genetic algorithm and Case-Based Reasoning systems. I would like to thank Professor Vittorio Addona for serving as one of the reviewers for this work, and Professors Shilad Sen, Daniel Kaplan and Andrew Beveridge for the continuous mathematical and other technical support.

Contents

List of Figures

1 Introduction

Imagine that you are given the task of producing the semester schedule for your academic department. You would need to schedule courses in such a way that there would be an instructor, room, time and students for each course to be offered. This task is called timetabling, also referred to as academic scheduling. A timetable is a table that shows the time, room and instructor for a set of courses that are to be scheduled. The task of producing timetables is for the most part a time-consuming and difficult task, because people building the timetable need to ensure that no scheduling conflicts will arise in any timetable.

The problem of scheduling courses is also called the University Course Timetabling Problem (UCTP). UCTP is concerned with the allocation of courses, instructors and students in a room and time slot, in such a way that it satisfies as nearly as possible the constraints imposed by a particular educational institution (Burke and Petrovic, 2002). For constrained instances of the problem, we can approximate the UCTP with the three graph coloring problem, which is a known NP-complete problem (Sipser, 1996). Consequently, the UCTP can be considered an NP-complete problem. Because of its characteristics and applicability, this problem has been extensively studied across the fields of operations research and artificial intelligence, researchers developed algorithms to solve timetabling problems for various educational institutions.

Researchers have used evolutionary techniques such as genetic and memetic algorithms, as well as other techniques such as case-based reasoning and clustering algorithms and a combination of these techniques to find optimized solutions for the UCTP. Most of the research efforts focus on how to efficiently schedule courses and instructors into rooms and time slots, with very little regard to students' scheduling needs. This problem becomes more acute for liberal arts academic institutions.

Students in liberal arts schools need to fulfill many prerequisites inside and outside their fields of study, and they need to plan their courses in order to complete these graduation requirements. This is particularly difficult for students who have more than one area of study. My research differs from the status quo in that it builds on the UCTP to include students' scheduling

needs as well as departments' administrative needs. In addition, it combines genetic algorithm and case-based reasoning techniques to estimate a solution for the UCTP.

In order to test the algorithm I generated benchmark data using knowledge of scheduling needs at Macalester College. I implemented and tested two versions of the algorithm: the non-elitist and the elitist version. Both versions return good results for less constrained benchmarks. The elitist version returns good quality solutions, even for highly constrained benchmarks. In order to produce data that is accurate to student needs, we at Macalester College have designed the Macalester Academic Planner web-application. This is a web application program that allows students to plan their classes ahead of time, giving the academic departments an idea of what classes, prior to registration, students want to take each semester. I use this data to place students in the courses to be scheduled into the timetable.

This paper is organized such that Section 2 gives a formal definition of the UCTP problem and introduces and discusses some of the techniques that previous studies have used to solve the UCTP. Section 3 describes more carefully what problem this work is solving, and introduces the research questions. Section 4 gives a high level account of the design choices for the system that I have implemented, and Section 5 presents the technical design and implementation choices I have made. Section 6 explains how I designed the experiments, and section 7 presents the results and discusses their implications. Finally, I conclude and present ideas for future research in section 8.

2 Background

In this section I will introduce some of the concepts that are relevant the solution models I have created using genetic algorithms and case-based reasoning . While I introduce the most relevant points about each concept, this section is in no way a thorough introduction to the University Timetabling problem, nor a rigorous technical discussion of genetic algorithms and case-based reasoning or of any other techniques I present here. For further details please read (Petrovic and Burke, 2004), (Wang, 2003), and (Watson and Marir, 1994) as they offer a more thorough coverage of the UCTP, genetic algorithms and case-based reasoning respectively.

2.1 The University Course Timetabling Problem - UCTP

Formally, the UCTP consists of a set of n events $E = \{e_1, e_2, e_3, ..., e_n\}$; in our case these events are courses to be scheduled in a set of j time slots $T = \{t_1, t_2, t_3, ..., t_j\}$ placed in a set of l rooms $\{r_1, r_2, r_3, ..., r_l\}$ and with a set of k students $S = \{s_1, s_2, s_3, ..., s_k\}$ and a set of j instructors $I = \{i_1, i_2, i_3, ..., i_j\}$ (Fen et al., 2009). Figure 1 shows an example of a timetable with ten courses, each with an instructor and an assigned time slot.

	Timeslot 1	Timeslot 2	Timeslot 3
Room 1	**Free time slot**	**Comp 123** S. Fox Student IDs = {ID 3, ID 13, ID 25, … }	**Math 137** K. Saxe Student IDs = {ID 5, ID 25, ID 43, … }
Room 2	**Comp 261** S. Fox Student IDs = {ID 1, ID 47, ID 53, … }	**Comp 124** S. Sen Student IDs = {ID 2, ID 37, ID 43, … }	**Free time slot**
Room 3	**Math 379** A. Beveridge Student IDs = {ID 2, ID 37, ID 53, … }	**Math 155** V. Addona Student IDs = {ID 4, ID 13, ID 25,… }	**Comp 225** E. Shoop Student IDs = {ID 34, ID 37, ID 43, … }

Figure 1: Example of a timetable

The UCTP imposes two types of constraints, hard constraints and soft con-

straints (Myszkowski and Norberciak, 2003). Hard constraints are those that a feasible solution should not violate, unless very exceptionally. For example, no instructor should be scheduled to teach two classes that are taught at the same time and no student should be scheduled to attend more than one class at the same time. Soft constraints are expected to be partially violated (McCollum et al., 2008). Examples of soft constraints are that no professor should be scheduled in more than two consecutive time slots. Another example is that no student should appear in all the time slots of a particular day.

The more interesting instances of the UCTP are those where the system is highly constrained. As Lewis and Paechter (2007) show, we can approximate the UCTP with the graph coloring problem. The graph coloring problem is an NP-Complete problem (Sipser, 1996); by inference, so is the UCTP. As such, no polynomial time algorithm finds an optimal solution to the problem. Thus we need to consider the problem as an optimization problem, in which we find the best solution from the set of all feasible solutions.

Because of the properties of the problem and its applicability in practical life, the UCTP has earned some popularity among the Artificial Intelligence and Operations Research community (Lewis and Paechter, 2007). Researchers have reported a number of solutions using a variety of techniques. Burke and Petrovic (2002) have divided these techniques into four groups: sequential methods, cluster methods, constraint based methods, and meta-heuristic methods. In this paper I present a meta-heuristic hybrid genetic system to find a feasible solution to the UCTP.

In sequential methods, timetables are usually represented as graphs, where courses are represented as vertices, and conflicts between vertices are represented by edges (Burke and Petrovic, 2002). The construction of a conflict free timetable can be considered a pure graph coloring problem. Each time slot represents a color in the graph. Adjacent vertices are given different colors in the graph coloring problem, similarly in UCTP, adjacent classes are given different time slots (Sipser, 1996). Figure 2 illustrates the concept of representing a timetable as a graph. In the Figure the arrows pointing in different directions serve to indicate an undirected graph. We can see that all the courses but Comp 123 and Comp 240 are connected by an edge, which is so because these two courses are offered at the same time and have the

same instructor. Referring back to the graph coloring problem, connecting the nodes representing Comp 123 and Comp 240 with an edge would be the same as connecting two nodes with the same color. Sequential methods have been widely employed in timetabling because they are easy to implement. However, they lack the power of modern intensive search methods, and as such they are not apt to solve highly constrained timetabling problems.

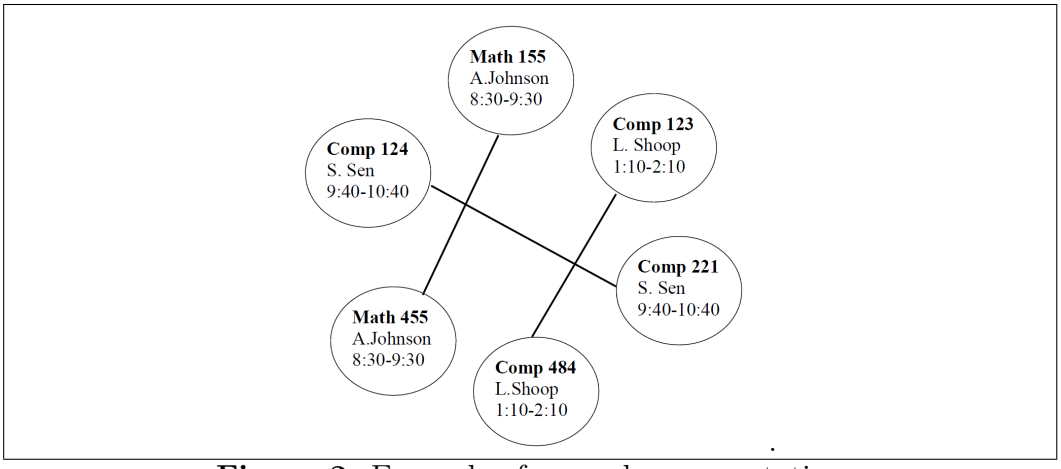


Figure 2: Example of a graph representation

Cluster algorithms assign classes into subsets called clusters, such that no class in a cluster is in conflict with each other. Cluster algorithms assign classes to rooms and time periods to satisfy hard constraints first and then re-assign them in order to satisfy the soft constraints. Each of the clusters will represent a possible timetable (Burke and Petrovic, 2002). To illustrate this concept, let us suppose we have the following courses to schedule: Comp 123, Comp 261, Comp 124, Comp 225, Math 155, Math 354,Math 355, Comp 494, Math 237, Comp 480 . Suppose that in our first attempt to schedule the courses we place Comp 123 and Comp 480, which have the same instructor, in the same *9:40-10:40AM* time slot. Given that an instructor can only be in one place at the time, placing these classes on the same time slot constitutes a collision. But clusters cannot contain conflicts, thus, Figure 3 shows us two clusters that are formed in order to avoid possible collision between the two classes. In cluster A we move the time of Comp 480 from *9:40-10:40AM* to *3:30-4:30PM*. In cluster B we move the time slot of Comp 123 to *3:30-4:30PM* from *9:40-10:40AM*. We end up producing two clusters representing

two different timetables. The main drawback of this method is that because the clusters are formed in a fixed manner, not many configurations are tried and therefore the quality of the timetables may not be the highest.

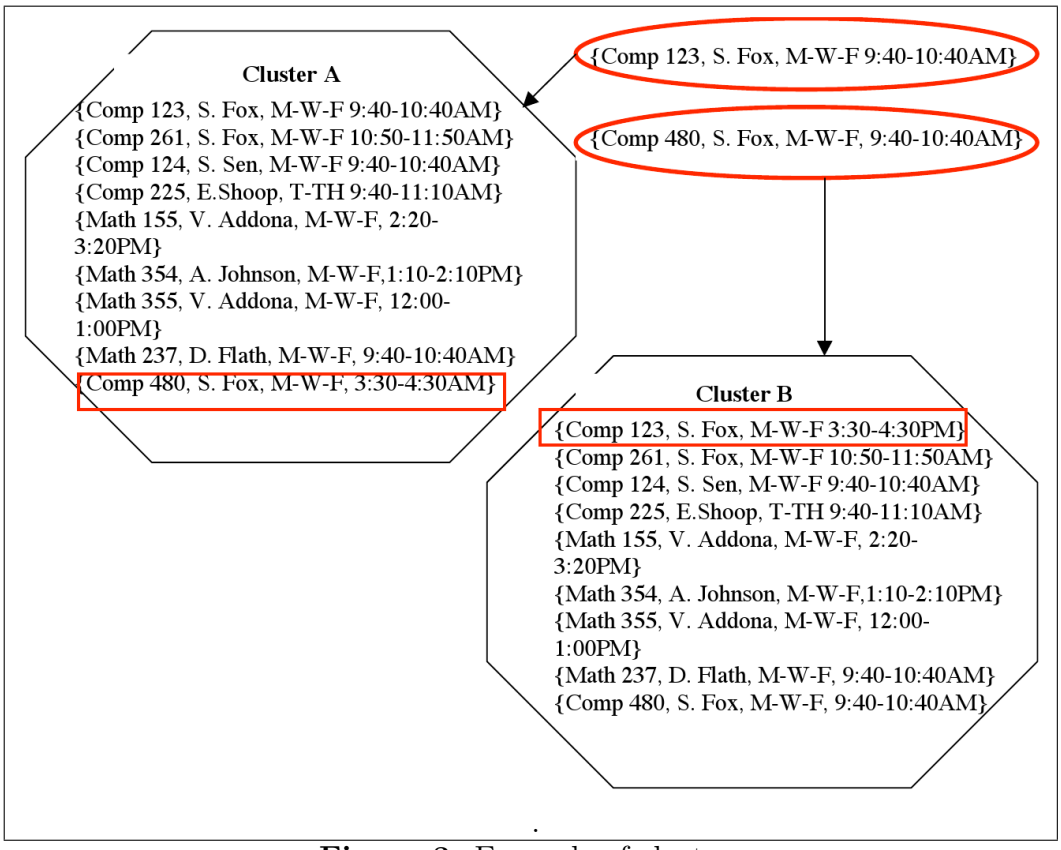


Figure 3: Example of clusters

Another approach to solving the UCTP is the constraint-based technique. In this technique the placement of courses and teachers to rooms and time slots is handled as a satisfiability problem. The method models the UCTP problem as a set of variables representing the courses and teachers to assign to rooms and time slots. The assignment is then modeled by a set of rules, if no rule is applicable to the current assignment the algorithm backtracks until a solution that satisfies all the constraints is found. the main drawbacks of this method is that it is difficult to express soft constraints as boolean expressions, and it is hard to improve the initial feasible solution, because the rules are predefined (Petrovic and Burke, 2004).

Meta-heuristic methods are more powerful than sequential, clustering and constraint-base techniques, in that they produce high quality solutions. These methods begin with one or more solutions and use search strategies to find local maxima(or minima depeding on the goal of the problem). These methods tend to eliminate or add new solutions to the set of current candidates with the goal of finding the optimum candidate(s) (Burke and Petrovic, 2002). Meta-heuristic methods have a fitness function to evaluate the pool of candidates, they have some human-defined time limit, and a function that allows the system to create new candidates. Meta-heuristic methods can produce high quality timetables, but they are computationally expensive (Petrovic and Burke, 2004). I am going to discuss meta-heuristic algorithms in more detail because my work falls into this class of problems.

2.2 Evolutionary Metaheuristic Approaches to Solving the UCTP

Over the past two decades, a number of researchers have focused on evolutionary algorithms to solve the university timetabling problem (Petrovic and Burke, 2004). Some of the commonly studied evolutionary meta-heuristic methods are genetic algorithms, memetic algorithms, ant-colony optimization algorithms and the hybridization of all of the above (Petrovic and Burke, 2004).

2.2.1 Genetic Algorithm

A genetic algorithm is an optimization technique based on the natural law of evolution. As Wang (2003) puts it, the "key idea of genetic algorithm is the survival of the fittest, as proposed by Darwin." Before I explain some of the genetic algorithm concepts, let me define a few important concepts that will appear in this section:

- A *solution candidate* or *chromosome* is a legal solution to the problem. In the case of the UCTP, a chromosome resembles the timetable in Figure 1.
- A *Population* is comprised of all of the chromosomes that the genetic

algorithm produces.

- The *Search space* is, theoretically, the set of all possible candidate solutions.
- The *Fitness* is the score or grade that is attributed to a chromosome. The fitness of a chromosome is inversely proportional to the amount of constraints the chromosome has violated.
- A *Gene* is the fundamental unit of the solution to a problem. In the case of the UCTP the gene is comprised of a course and its instructor, students and room.

This algorithm uses the biological principles of selection, crossover and mutation to perform a search in often complex and big search spaces and it attempts to find global solutions, while avoiding local optimal solutions (Petrovic and Burke, 2004). Genetic algorithms have a set of genetic operators: selection, crossover and mutation. Figure 1 shows the steps of a generic genetic algorithm:

Algorithm 1: A Generic Genetic Algorithm

```
Input: Genetic algorithm parameters
Output: Possible solutions to problem
begin
    initializePopulation(paramaters)
    evaluatePopulation(population)
    while currentGeneration ≤ maximumGeneration do
        select (listChromosomes)
        crossover (matingPool)
        mutate (matingPool)
        foreach chromossome in matinPool do
            if chromossome ≥ desiredChromosome then
                addToResultsList(chromosome);
            end
        end
    end
    return resultsList
end
```

A genetic algorithm is comprised of three phases: population initialization, evaluation and the evolutionary processes (Petrovic and Burke, 2004). In the population initialization phase a number of chromosomes are generated in order to encode solutions to the problem. There are two important factors when considering the creation of a population of chromosomes. The first factor is randomness. If the chromosomes are not created randomly enough, the process is biased towards certain types of chromosomes, which might lead to a local minima solution but not to a global minima as desired (Lucas, 2000). The second factor, a more important one, is the size of the population. The size of the population affects the convergence of the algorithm: larger population size leads to slower convergence speed. If the population is too small, then the search space will not be very useful (Lucas, 2000).

In the evaluation phase a fitness function is used to test each chromosome in the population to see if the chromosome is a suitable solution to the problem at hand. For a system with n constraints, a fitness function usually checks to see if a chromosome violates any of the n constraints imposed by the problem. The algorithm 2 illustrates the steps of an evaluation process:

Algorithm 2: A Genetic Evaluation Algorithm

```
Input: A chromosome
Output: List of chromosomes and their fitnesses
begin
    foreach contraint in listOfContraints do
        booleanViolates = checkIfViolatesContraint(chromosome)
        if booleanViolates then
            fitness=chromosome.getFitness() - penalty
            chromosome.setContraint(penalty)

        end
    end
    return chromosome
end
```

In the last phase, the algorithm depends on genetic operators to improve chromosomes for each generation. There are many genetic operators, but here I will discuss crossover and mutation, because these operators are the most used across the literature. Crossover is an operation used to generate a new

chromosome from two parents (Wang, 2003). The main aim of crossover is to mix the genes of two individuals in the hope that the result will be stronger than either parent. In order to perform crossover, we first evaluate the whole population and select chromosomes to become parents. During crossover, we take some of the genes of a chosen chromosome, parent 1, and we place it into another chromosome, parent 2, and vice-versa. Crossover provides a mechanism to mix and match through random processes. Algorithm 3 illustrated the crossover mechanism:

Algorithm 3: Genetic Crossover Algorithm

Input: The mating pool
Output: New population
begin
  while *matingPool* $\neq$ *empty* **do**
    pick two parents randomly pick a division point for each parent pick the data at the left of the division point from parent 1 pick the data at the right of the division point from parent 2 place both data pieces in the child chromosome add the child to the newPopulationList
  end
  return newPopulationList
end

Mutation takes a selected parent chromosome from the population and makes changes to the content of that chromosome. For most implementations of genetic algorithms, mutation happens with a very low probability. This means that only a few chromosomes in the population get changed in the mutation process. Mutation is used to increase diversity in the new population and, thus, avoid local minima (Wang, 2003).

Another very important mechanism in a genetic algorithm is selection, the process in which parents are chosen for reproduction. One of the simplest methods is the Roulette Wheel selection. This selection technique calculates the selection probability of a parent by dividing the fitness of the chromosome by the cumulative fitness of all chromosomes in the population. There are other selection methods such as random selection, tournament selection and best first selection; some are more or less elitist (favoring the fittest) in nature than others(Russell and Norvig, 2003). The selection process is important

Path	Cost
[0,1,2,3,4,5]	600
[0,2,4,1,3,5]	620
[0,2,1,4,3,5]	560
[0,3,1,2,4,5]	540

Table 1: Table of paths and their costs

because a poorly designed selection method will either make the algorithm find locally good solutions too early, or will make the algorithm return poor solutions.

Let me walk you through an example of a genetic algorithm, using the following problem: Imagine that a group of tourists want to visit six cities in the Midwest of the United States. They want to do so in a way to spend the least amount of gas possible. Thus, they would want the shortest path from the start city to the end city. Also, they want to go through each city only once. Let us use the numbers 1,2,3,4 and 5 to designate the cities on the way between the start city (city 0) and the end city(city 5). We use the fitness function to calculate the cost of each path from the start to the end city. Some example paths and costs are given in table 1:

Paths [2,1,4,3,5] and [3,1,2,4,5] are the ones with the lowest costs. Thus, the genetic algorithm chooses them as parents. Crossover will happen as illustrated in Figure 4. We pick a random point represented by the dashed line, and we take the items of parent 1 after the dashed line and place it in the child, and take the items of parent 2 before the dashed line and place it in the child, generating a new chromosome. The generated child has a duplicate city, meaning that it visits one less distinct city than the parent. In this example mutation is used to fix the child to make it conform to the "no duplicate city" constraint. If the algorithm is performing well the child might not attain a better score than its parents, but we are guaranteed that the overall fitness of the new generation should be greater. If we ran the algorithm for one generation, then the best feasible solution would be [3, 1, 2, 4, 5]. Although this example is trivial, If we increased the number of cities to 50 and the number of generations to 100, the problem would very hard for a human to solve, and then it would make more sense to apply a genetic algorithm to solve it.

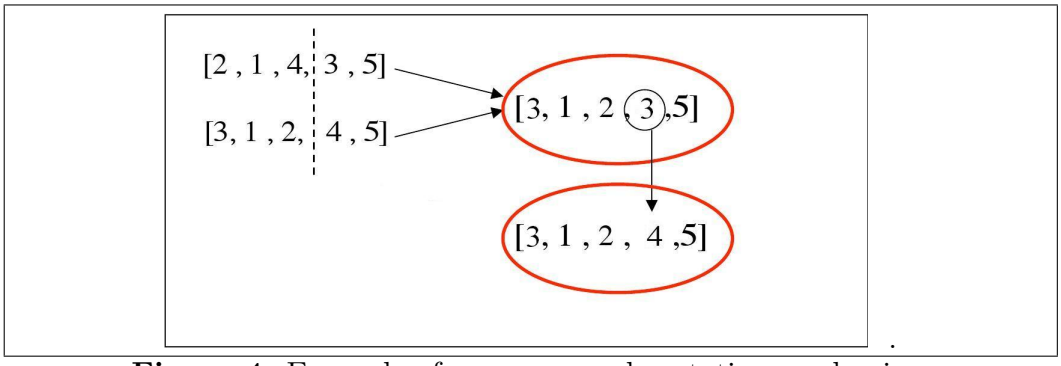


Figure 4: Example of crossover and mutation mechanism

2.2.2 Genetic Algorithm Applied to the UCTP

Many research efforts have used genetic algorithms to address the university time-tabling problem. Abramson and Abela (1992) proposed a parallel genetic algorithm to solve the university timetabling problem. This work is founded in the idea that biological mating is highly concurrent, in that chromosomes reproduce with little regard of the rest of the population. The algorithm uses a shared memory multiprocessing model to speed up both population creation and mating. A barrier level synchronization at the end of each generation is required in order to avoid chromosomes overriding each other.

Burke et al. (1994) developed a genetic algorithm for the university timetabling problem, which differs from Abramson and Abela (1992) in that its solution is sequential rather than parallel. Also, the proposed solution included a graphical user interface, which enabled users of the system to contribute towards the scheduling process by letting them change the parameters of algorithm to attain desired results. This work exemplifies earlier efforts to solve the university timetabling problem that were not fully automated.

Burke et al. (1995) proposed a hybrid genetic algorithm for highly constrained timetabling problems. This work differs from Burke et al. (1994) in that it applies two algorithms in order to solve the problem at hand. The first algorithm finds a non-conflicting set of courses, while the second algorithm assigns a room to each selected course event.

Ross et al. (1998) suggested that genetic algorithms should not be used as a method to find solutions, but rather as a way to search for the best algorithm to solve a given instance of the university timetabling problem, an approach that leans towards hyper-heuristic, where a set of algorithms is searched by a high level algorithm (Qu et al., 2006). The main idea behind such an approach is that the genetic algorithm has available a set of solutions and the methods that have generated these solutions (for example, genetic algorithm or tabu search). When a new problem arises, the genetic algorithm retrieves previous solutions that solved problems similar to the new problem. Then, the genetic algorithm evaluates the previous solutions to pick the fittest. In the last step, the genetic algorithm returns the technique that was used to solve the strongest solution.

Wilke et al. (2002) published a hybrid genetic algorithm for the university timetabling problem. The genetic algorithm included a number of hybrid operators. These operators consisted of adapted mutation and repair operators. If the solution attained by the genetic algorithm does not yield the desired fitness, then the system resorts to the hybrid operators. These operators change room assignments, and class sequences in order to find the correct time slot for a class, among other capabilities.

Lewis and Paechter (2005) proposed the use of grouping genetic algorithms in order to solve the university course timetabling problem. Their genetic algorithm differs from other genetic algorithms in that theirs treats the problem of solving hard constraints as a grouping problem. A grouping problem is one where the task is to partition a set of objects into a collection of mutually disjoint subsets. The union of all the subsets add up to the size of the set. Moreover, there is no intersection between any subset. Applying this concept to the UCTP, a feasible solution is one where all of the events (classes) have been partitioned into feasible time slots and none of the time slots are in conflict, and where all the events can be placed in their own suitable room.

In addition, Lewis and Paechter (2005) presented an elegant grouping genetic algorithm that represents the timetable as a matrix. The idea of the grouping algorithm is to create time slots for all possible classes, and if there are remaining un-scheduled classes, then we create more time slots in the matrix, and the penalty function is equivalent to the amount of newly allocated space

(Lewis and Paechter, 2007). My work uses the same representation for the UCTP as this work's. I also create time slots for all possible events, however I do not create more time slots in the matrix, because I use a fixed number of course events, consequently, I do not allow the size of the timetable to change during the execution of the algorithm.

Perzina (2006) presented a self-adaptive genetic algorithm that solves the UCTP. In addition, the paper also offers an algorithm to find feasible solutions for the student enrollment problem. The author defines a self-adaptive genetic algorithm as one in which the parameters of the algorithm are optimized during the same evolution cycle as the problem itself. The author implemented a parallel version of the self-adaptive algorithm that follows the master-worker model. The "master" genetic algorithm is responsible for running the genetic operators (crossover, mutation and others), while the "worker" genetic algorithms are responsible for measuring the fitness of the chromosomes that the master gives them. The enrollment problem that this paper presents is similar to my student constraint problem; however my method differs from this authors' in that the author solves the student enrollment problem separately from the UCTP, while I incorporate the student enrollment problem into my UCTP instance.

Wang et al. (2009) presented a genetic algorithm that is novel in that it introduces the concept of self-fertilization. Parent chromosomes do not exchange genetic materials, that is, timetables do not exchange time slots, instead, a parent self-reproduces to generate its children. The parent does so by applying a crossover operator that works the same way as a heavy mutation operator, it exchanges time slots within itself and not with any external entity. The advantage of using this method is that it keeps the integrity of the schedules, thus no further computing resources need to be spent in repairing the child schedules after recombination. I have implemented a similar crossover operator. It made sense to use this technique because of its improvement in time complexity, especially for highly constrained benchmarks.

2.2.3 Memetic Algorithms

Like genetic algorithms, memetic algorithms are population-based algorithms, which find feasible solutions for hard problems. Instead of following a bio-

logical route, memetic algorithms are founded in the "meme" concept, which signifies the basic unit of cultural transmission or imitation (Burke et al., 1996). Instead of genes, memetic algorithms operate with agents. Another significant difference from genetic algorithms is that memetic algorithms employ local improvement as one of their operators. Memetic algorithm operators are problem aware, and the solutions are not only globally optimized but also locally aware (Cotta and Fernandez, 2007). The algorithm follows every step of a genetic algorithm, but it adds local improvement to each generation of the population. Because of this, it reduces the search space in which to find the optimal solution (Burke et al., 1996). Though my project does not implement a memetic algorithm, the elitist genetic algorithm that I implemented is similar to memetic algorithms in that my algorithm gets the best solution per generation (which is a local solution), and it keeps that solution until It finds a better local solution. My algorithm repeats this procedure until It finds the best chromosome in the system.

Burke et al. (1996) and Cotta and Fernandez (2007) proposed a generic memetic algorithm as a solution to the UCTP. Burke et al. (1996) set up the theoretical approach to solving the UCTP using Memetic algorithms. The study produced a memetic algorithm that implemented hill climbing as selection method; the algorithm was applied to fixed size timetables. In their paper, Cotta and Fernandez (2007) offer guidelines on the best ways to design effective memetic algorithms. Rossi-Doria and Paechter (2004) proposed a more specific memetic algorithm. In this algorithm, solutions are represented as a matrix with r rows and t columns. The number of rows is dictated by the number of time-periods available, while the number of columns depends on the number of rooms available. Each cell of the table holds a list of courses that are scheduled in that time-period and room. This algorithm uses a stochastic local search algorithm that improves the feasibility of the timetable by reducing the number of time slots used.

2.2.4 Ant Colony Optimization (ACO)

Ant optimization is another form of evolutionary optimization technique. Dowsland and Thompson (2005) and Socha et al. (2003) present yet another evolutionary technique, the ant colony optimization technique. Ant algorithms are based on the observation of how real ant colonies find the shortest

path to their food. Both real and artificial ant colonies are composed of a population of chromosomes that work together to achieve a certain goal (Dorigo and Socha, Dorigo and Socha). In order to find the solution, the algorithm uses n cycles. During the cycles, m ants construct a feasible solution for each of the n cycles. Ants decide which solution to adopt based on the number of other ants that have chosen a given trail. Thus two heuristic methods emerge: constructive heuristic and pheromone trails heuristic.

The Constructive heuristic uses techniques such as randomized nearest neighbor. In this technique, the decision of which trail to follow favors the nearest neighbors, and the trail is picked at random among the nearest neighbors. The probability that an ant will follow a trail is determined by the amount of scent in the trail. In order to know the amount of scent, the scent is stored in a matrix t which is initialized to 0 for every trail. The values of the matrix are updated each time an ant chooses the trail. The most chosen trails are then favored.

One can use a graph structure to represent the problem instance. Each node in the graph is a possible solution. The graph is a complete graph as the artificial ants have access to every possible solution, thus there needs to be an edge from and to every node (Dorigo and Socha, Dorigo and Socha). We initialize the algorithm by randomly assigning an ant to a node or solution. Then the ant accesses the scent trail matrix to get the current scent of a given edge in the graph (the trail). The probability of an ant moving from one node in the graph to another is given in equation **??**. In equation **??**, P_{ik} is the probability of moving from current node i to next node k, τ_{ik} is the scent on the edge i, j, η_{ik} is the weight of the edge, α is a constant that controls for the influence of the scent and β is the constant that controls for the influence of the weight of the edge. F represents the set of feasible solutions.

$$P_{ik} = \begin{cases} \frac{\tau_{ik}^{\alpha} \eta_{ik}^{\beta}}{\sum_{ik \in F}} \\ 0 \qquad \text{else} \end{cases}$$

Let me walk you through an example of ant algorithm applied to a trivial instance of the UCTP. Consider the nodes in the graph in Figure 5 to be courses to be scheduled and the edges the time slots in which the connected

courses are placed. The weights of the edges represent the cost of placing the two particular classes (nodes) in the same timeslot. The cost is proportional to the number of constraints the combination of courses violate. Each ant starts in a randomly selected node. Let us track the movement of an ant randomly placed in node *A1*. At the beginning all the trails (paths between nodes) have a scent of zero. Using constructive heuristic, our ant randomly picks course *A2* and places both *A1* and *A2* in the same time slot, say in time slot 1, it then records the cost of scheduling these courses together (which is 2 from the graph). Before moving to the next node it calculates the scent and updates the scent matrix. It does the same until it has visited all the vertices in the graph. In the next iteration another ant will chose which classes to put in the same time slot depending on the scent of the trail. Any ant will know that courses *A1* and *A5* can be put in the same timslot, but *A2* and *A5* cannot.

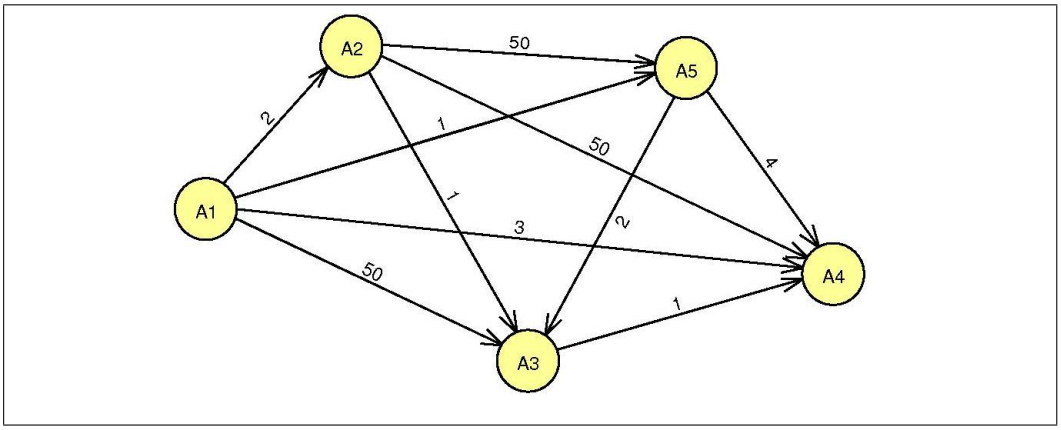


Figure 5: Graph of classes for the ant algorithm

Dowsland and Thompson (2005) built an ant colony heuristic that represents an instance of the UTCP problem as a graph coloring problem. The authors state that at each stage the option of coloring vertex i in color k is selected with the probability given in equation 4. For their algorithm they used a constructive heuristic. Socha et al. (2003) uses a similar approach, however their approach departs from that of Dowsland and Thompson (2005) in the way they build the graph that represents an instance of the UCTP problem.

In order to build the graph, Dowsland and Thompson (2005) create a set of virtual time slots with the same length as the number of existing events.

Note that in reality the number of time slots available is usually much smaller than the number of events. Each virtual time slot maps to one of the actual time slots. To find the best solution, ants walk through the virtual time slots and assign them an event and then use a general ant algorithm to build the optimal trail. The authors found that the success of the most effective meta-heuristic implementations depends on getting the right balance between searching for good solutions - and allowing for exploration of new areas of the search space. This is a balance that I too had to consider when I designed my genetic algorithm.

2.3 Local Search Approaches to Solving the University Timetabling Problem

2.3.1 Simulated Annealing

Simulated annealing (SA) is a Monte-Carlo technique used to find solutions to optimization problems. Its main aim is to find the global minimum or maximum of a function in a large search-space (search-space here means the same as in genetic algorithms), however this method is not guaranteed to find the optimum solution to a problem (Abramson, 1991). This system simulates the behavior of the cooling of particles: when particles have high temperature, they move around with random displacements, as the system cools, the particles move less randomly, and usually make smaller movements. Before continuing let me define a few terms:

- *State* or *configuration*, this is a possible solution to the problem, much like a chromosome for a genetic algorithm. For the UCTP a state is a course event in a time slot
- *Temperature (T)* is a parameter that is inspired from the physical temperature of an object. It represents a number and that number changes as the system evolves
- *Neighborhood* is the set of possible states to which a particular state can move.
- *Transition probability* is a probability with which we pick the next

neighbor.

- *Acceptance probability* is the probability that a change to a next state is accepted or not. That depends on whether the move from one state to the other lowers the system energy or not.
- *Cooling rate* is a constant that dictates the rate by which temperature should fall throughout the simulation.

A simulated annealing system relies on a number of parameters. The more important parameters are the search-space of the problem, the temperature of the system (often called T), the transition probabilities, and the acceptance probabilities (Fleischer, 1995). In order to specify the state space of a problem, we need to define the neighbors of each state, which is done by the user (Fleischer, 1995). In the case of the UCTP, the neighbors of each state are represented by the course events scheduled in a timetable.

On the other hand, the temperature of the system is a global time-varying parameter; the choice of how the temperature decreases depends on the user choices (Schaerf, 1999). It is important that the temperature is well chosen, as the evolution of a state depends on the temperature. The dependency is such that if the temperature is high, the next state is chosen almost at random, otherwise, the choice of the next neighbor is performed with a probability chosen to decrease error as T goes to zero (Fleischer, 1995).

The transition probability is the probability that the system will move to a next state S' given that the current state is S. This probability depends on the current temperature, on the order in which the neighbors are generated and on the acceptance probability. The acceptance probability function is defined as follows:

$$P = \begin{cases} 1 & \text{if } e' < e \\ exp^{(e-e')/T} & \text{if } e' >= e \end{cases} \quad (1)$$

The probability of acceptance is one if the energy of the next state (e') is less than the energy of the current state (e), or else the probability of acceptance is defined by the exponential difference in energy. A careful consideration of these parameters is essential for the efficiency of the system (Schaerf,

1999).

Let us apply this technique to the UCTP problem. In this case, a particle is replaced by course event that we want to schedule (courses, their instructors, and students). The system energy is replaced by the timetable fitness function. The initial temperature is computed from the difference in cost for moving the course event from one time slot to another, over the number of swaps performed during the motion. Algorithm 4 describing the general procedure for a simulated annealing system that solves the UCTP:

Algorithm 4: Simulated Annealing algorithm applied to the UCTP

```
Input: UCTP paratemeters
Output: Possible timetables
begin
    initiatePopulation()
    calculate the cost (cost = fitness-1) of a timetable
    calculate the initial temperature of the system
    foreach iteration i do
        randomly choose a from and to time slot
        calculate(cost of removing the from cost from its time slot)
        calculate(cost of inserting the from into the to time slot)
        calculate(cost of inserting the from course event into the to time
        slot)
        calculate(difference in costs between the two costs above)
        calculate the acceptance probability
        Either accept or reject the change depending on the probability

        if cost of timetable = 0 then
            stop
        end
        else if cost is not changing then
            stop
        end
    end
end
```

Abramson (1991) and Liu et al. (2009) used simulated annealing to solve the UCTP problem. Abramson (1991) used simulated annealing to devise

a general-purpose algorithm to solve an instance of the UCTP applied to an Australian high school. The proposed solution optimizes groups of objectives and solves the problem in multiple-stages. On the other hand, Liu et al. (2009) took on the more specific task of developing a new neighborhood structure for the simulated annealing. The neighborhood structure is obtained by performing a sequence of swaps between two time slots, instead of only one move in the standard neighborhood structure. Another technique that follows the local neighborhood search model is the tabu search algorithm. Next, I am going to briefly introduce this technique and the body of research that used it for the UCTP.

2.3.2 Tabu Search

Like simulated annealing, tabu search is a local optima mathematical optimization technique (Hertz et al., 1992). Unlike simulated annealing, tabu search does not have a temperature parameter. The performance of the tabu search algorithm is dependent largely on the chosen neighborhood, thus the subset neighborhood is the most important parameter. Neighborhood is defined as the set of all the neighbors generated by performing an atomic change within state s. Tabu search relies heavily on a memory system, called the tabu list, holding every visited solution, including those that are no longer to be considered during the search. The tabu list is important, because it enables the algorithm not to visit states already seen (Hertz et al., 1992). A reasonable way to implement the tabu list is to use a queue of fixed size k, when a new move is added to the tabu list, the oldest one is discarded.

Let us apply this technique to a general UCTP problem. Like simulated annealing, a tabu search starts with a random initial population. The population is generated by creating random timetables. This is done by randomly placing every course event into a time slot. The algorithm picks a timetable in the population and creates a new timetable by making an atomic change to the previous timetable. The atomic change happens when a course event is changed from its original time slot into a new time slot, just like what happened in simulated annealing. The algorithm evaluates the created solutions, and makes the one with the least cost the current solution. It adds all of the solutions it creates in each iteration of the algorithm into the tabu list (Gaspero and Schaerf, 2001).

Algorithm 5 shows a high level description of a tabu search procedure:

Algorithm 5: Tabu search algorithm applied to the UCTP

```
Input: UCTP parameters
Output: Possible timetables
begin
    foreach timetable in Population do
        choose(initial timetable)
        generate(new timetables by changing a course event in previous
        timetables)
        choose(best solution in the subset of new timetables)
        if best chromosome is satisfies the stopping criteria then
            stop
        end
        else
            place(the newly generated timetables into the tabu list)
        end
    end
end
```

Costa (1994) and Kendall and Hussin (2005) used tabu search as a technique to solve the UCTP problem. Costa (1994) developed a generic tabu search algorithm with various requirements (or constraints), while Kendall and Hussin (2005) developed a tabu search algorithm to solve the MARA University of Technology scheduling problem. According to Kendall and Hussin (2005) their work is novel because of the weekend constraints that were added due to the administrative needs of the university also because the data-set for this university has never been used before to model the university timetabling problem.

Looking across the meta-heuristic techniques I have introduced previously, one common theme is that most of these methods rely on a random generation of an initial population. Moreover, all of the techniques choose their chromosomes depending on a fitness or cost function. Also, all of the techniques create a new population that replaces the old one, and all of the techniques aim at creating fitter or cheaper new population. Most of the algorithms have parameters such as population size, and probability of choosing a state.

My genetic algorithm is not any structurally different from the other meta-heuristic techniques

The technique I will introduce next does not depend on exactly the same parameters and processes as meta-heuristic techniques. Instead, this method relies on past information to give answers to present problems. This technique is called case-based reasoning.

2.4 Case Based Reasoning - CBR

According to Qu (2002), Case-Based Reasoning is "a Knowledge-Based reasoning technique that solves problems by retrieving the most similar previous cases from a store called the case base and by reusing the knowledge and experiences from these cases." A case is a piece of knowledge that represents a solution within a context (Watson and Marir, 1994). For example, in the UCTP problem a case could be a timetable representing a feasible arrangement of classes. Before continuing this discussion, allow me to introduce few concepts that will appear later:

- A *Case or solution* a case represents a solution to a problem. It is similar to a chromosome for a genetic algorithm.
- The *Case-memory or case-base* is a storage unit that contains all the cases for the case-based reasoning system.
- An *Index* is a programmable object that contains information about parameters that are used to compare two cases.
- A *New problem* is the new problem that needs to be solved.
- The *Target case* is usually the new case, it is the case we are comparing to the ones in the case-base.

The CBR mechanism is comprised of the four "REs" (Watson and Marir, 1994): Retrieve, reuse, revise and retain. Figure 6 shows the case-base reasoning cycle (Qu, 2002). In the retrieve phase the system finds and returns the case in the system's memory that is the most similar to the target (or new) case (Watson and Marir, 1994). In the reuse phase we use the retrieved

case to solve the target case. In the revise phase we revise the solution to assure that it satisfies the constraints imposed by the target case. Lastly, in the retain phase the system adds the new case to the case-base (Qu, 2002).

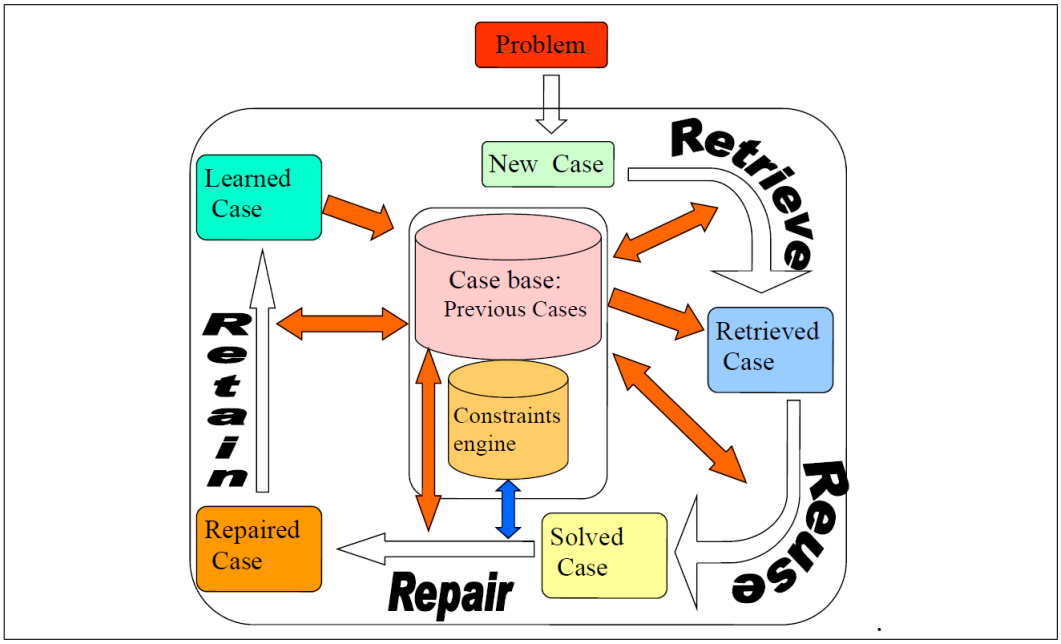


Figure 6: Case-Based Reasoning mechanism

In order to design and implement an efficient CBR system, one needs to consider five aspects of this system: Case representation, indexing, case-base maintenance and management, and case adaptation (Qu, 2002). In choosing the representation of the cases, we need to know what the relevant aspects of the underlying problem are, and what aspects differentiate one case from another (Burke et al., 2001).

The maintenance of the case-base relies heavily on the retain phase of the case-based reasoning cycle (Watson and Marir, 1994), since new cases are added to the case-base during this phase. The management of the system is dependent on the level of sophistication of the system. It can be done manually by having a human subject looking at the cases in the case-base and either add or delete cases, or it can done dynamically by writing a computer program that looks through the cases in the case-memory and decides to adds new cases or deletes old cases (Qu, 2002). This phase is

very important because it determines the quality of the case-memory, and therefore the quality of the case-based reasoning system.

Adaptation, as Watson and Marir (1994) put it, "looks for prominent differences between the retrieved case and the current case and then applies formulas or rules that take those differences into account when suggesting a new solution." Adaptation is the hardest phase of the case-based reasoning cycle (Qu, 2002). This is so because the system needs to understand how to change the contents of the current case to resemble the goal solution.

In addition, adaptation is difficult because the system might need to have the context of the contents of the current and goal cases before it can change them. For example, in the school timetabling example, in order to adapt a new timetable to a goal timetable we would need to know what a perfect goal timetable should look like, and this is a piece of information that we do not have readily available to us. Adaptation is the underlying mechanism behind the revise phase (Watson and Marir, 1994), because during revision the system changes the retrieved case to satisfy the constraints that the new case imposes (Burke et al., 2001).

In order to better illustrate this problem, let us apply it to a route finding problem similar to the one I have used during the discussion of genetic algorithms. Say we have 10 cities spread over the map as shown in Figure 7. Imagine that three different friends want to go to destinations 6, 9 and 10. Then, using some algorithm we calculate all the distances from the starting point to each target distance. We save the different routes from city 1 to city 6, from city 1 to city 9 and from city 1 to city 10 and their respective weights. So to increase the quality of the case-base, we should only add the most promising routes to each of the end destinations.

Say that the optimal paths to each destinations are [1,2,4,6] for the route to 6, [1,5,9] for the route to 9 and [1,3,8,10] for the route to 10. We add these routes to the case-base. If a friend three months later asks how do I get from city 1 to 10, the case-based reasoning system answers the question by retrieving the path [1,3,8,10]. If however the friend says that she wants to pass through cities 5 and 8 to get to 10. Then we need to revise [1,3,8,10] to include the new requirement, which is to pass through 5 and 8. The algorithm uses adaptation to learn about the new case. After adaptation the algorithm will return the route [1,5,8,10]. Then the case-based reasoning

system saves this new solution as a case of going from 1 to 10 through 5 and 8. Though this is a simplistic example, it depicts the core mechanisms behind case-based reasoning.

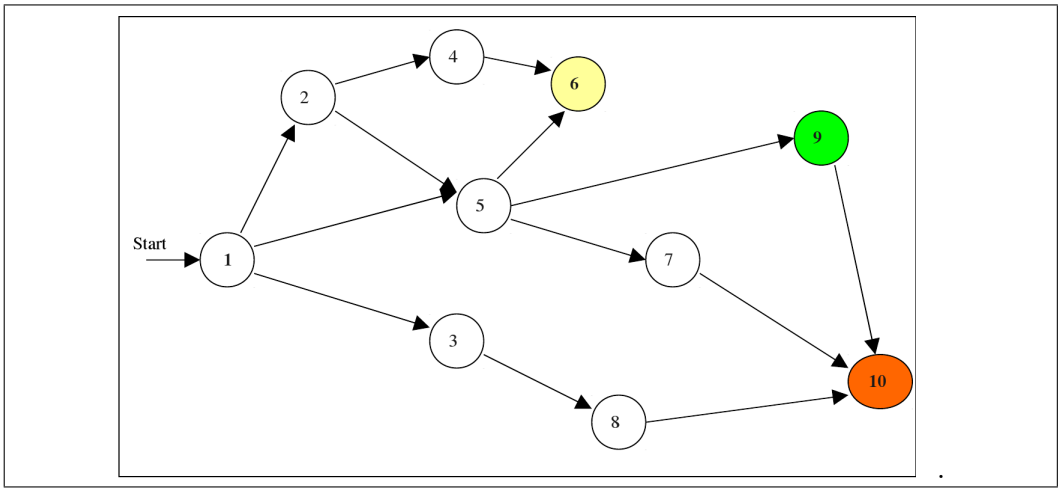


Figure 7: Graph representing the cities

There is a body of research investigating the use of case-based reasoning to solve the UCTP. Burke et al. (2000) and Burke et al. (2001) proposed a case-based reasoning approach to solving the UCTP. The authors used attribute graphs as form of representing cases. In attribute graphs, nodes represent events and edges represent the relationship between any two events. In the UCTP, nodes represent courses, edges represent hard constraints, or soft constraints. The only disadvantage of this method of representing cases is that matching it to other cases becomes a graph isomorphism problem, which is an NP-Complete problem (Burke et al., 2000).

Qu (2002) proposed another case-based reasoning system that departs from the one described previously. Instead of using the CBR system to retrieve solutions to cases, it rather looks to find the most appropriate heuristic to apply to a case in order to solve it. The system stores an attribute graph for the cases, and their respective heuristics. When a new case emerges, we take the new case and perform a nearest neighbor between the new and each of the existing cases. Once the closest case is found, the system retrieves the heuristic applied to solve the retrieved case, in turn, the heuristic is applied to the new case. This approach has advantages over the previous one in that

it allows for general timetabling problem solving, rather than just particular instances of the UCTP.

The techniques I have chosen to combine in my research are a genetic algorithm and case-based reasoning. The main motivation for my choice is that properly coded genetic algorithms can be very effective in solving the UCTP. In order to avoid the cost of producing timetables from scratch, I decided to use case-based reasoning as a system to save and retrieve previous solutions from gentic algorithms. Table 2 provides a summary of selected literature works and the methods they have used to solve the UCTP.

Study	GA	CBR	MA	PSO	ACO	SA	TS	COBR
(Abramson, 1991)						✓		
(Burke et al., 1994)	✓							
(Costa, 1994)							✓	
(Burke et al., 1995)	✓							
(Rich, 1996)	✓							
(Burke et al., 1996)			✓			✓		
(Goltz and Matzke, 1999)								✓
(Lucas, 2000)	✓							✓
(Burke et al., 2000)		✓						
(Burke et al., 2001)		✓						
(Wilke et al., 2002)	✓							
(Qu, 2002)		✓						
(Socha et al., 2003)					✓			
(Rossi-Doria and Paechter, 2004)			✓	✓				
(Lewis and Paechter, 2005)	✓							
(Kendall and Hussin, 2005)							✓	
(Dowsland and Thompson, 2005)					✓			
(Perzina, 2006)	✓							
(Cotta and Fernandez, 2007)			✓					
(Raghavjee and Pillay, 2008)	✓							
(Wang et al., 2009)	✓					✓		
(Fen et al., 2009)				✓				
(Liu et al., 2009)						✓		
(Nunes, 2010)	✓	✓						

Table 2: Studies of the UCTP and their methods. GA stands for genetic algorithms, CBR for case-based reasoning, PSO for particle swarm optimization, ACO for ant colony optimization, SA simulated annealing , TS Tabu Search and COBR for constraint-based reasoning

3 Problem Description

In this section I present the central problem that this research attempts to solve. I formulate the research questions that this paper answers, and I contextualize them to the timetabling needs of Macalester College, which is the institution of primary interest in this case. I then formulate some of the constraints that need to be put in place to ensure the usefulness and feasibility of the timetables.

3.1 UCTP Applied to Macalester College

3.1.1 Motivation

The UCTP problem arises in a number of diverse academic institutions with differing structure and timetabling needs (Burke and Petrovic, 2002). There is a line of research that aims at solving the UCTP problem in a generic manner, regardless of the unique needs of every institution (Petrovic and Burke, 2004)(Wang, 2003) (Lucas, 2000) (Qu et al., 2006) (McCollum et al., 2008). On the other hand,the case studies accomplished by Wang et al. (2009) and Wilke et al. (2002) are examples of research projects that have applied the UCTP problem to specific institutions.

My research focuses on Macalester College's timetabling needs. There are many motivations for my choice of this college. The first reasom, and the most obvious, is due to my matriculation at this school, and I feel that contributing towards timetabling automation might prove invaluable to the institution. But more importantly, Macalester's academic planning needs are very different from those of larger research universities, making it an interesting case study. One of the reasons for this difference is that Macalester is a liberal arts college, and as such students are required to take classes from a variety of disciplines outside of their area of study.

In addition, because of the small student to professor ratio, and the relatively small set of classes offered per department, scheduling classes should take into account students' needs to take the necessary classes to fulfill their academic program. This is an important difference between liberal arts and non-liberal

arts schools. A student registered to pursue a Bachelors of Science in a non-liberal arts school is primarily concerned with the course load for that major. However, a liberal arts student is required to take classes outside of her area of study to satisfy the multiple requirements that the school imposes. If the student decides to do multiple majors and minors, then the chance of scheduling a collision between two needed classes increases.

Another factor that makes the Macalester case-study compelling (and can be more widely applied to other liberal arts schools in general) is its many hybrid programs called interdepartmental majors. These programs require students to broaden their academic interests beyond one specific academic department, thus making it challenging to coordinate classes across disciplines. I call this conflict the student enrollment collision problem.

In my research, however, I do not use assigned classrooms as a variable that promotes conflict. Because of the limited number of classrooms at Macalester due to its small size, it isn't necessary to incorporate room choice as a constraint in scheduling conflicts. In a larger institution, room assignment may present a larger problem.

Applying the UCTP problem to Macalester College is a difficult task and probes important research questions. In the next subsection I will present some of these questions.

3.1.2 Research questions

At the end of each semester, students at Macalester College register for classes for the following semester. Each student has a unique personal timetable. Sometimes, students want to take two classes that are scheduled at the same time, and thus are forced to choose one class over the other. Often, this does not cause many problems, but in some circumstances this causes planning conflicts and consequently students must drop one their majors, minors or inter-departmental concentrations because of the inability to take a required class that coincides with another requirement.

To solve the UCTP given the contraints at Macalester, I formulated the following research questions:

- *Research question 1:* Is it possible to produce schedules that are feasible to both academic departments and student planning needs?
- *Research question 2:* Can such feasible schedules effectively accommodate instructors' scheduling preferences, while not increasing collision in students' academic schedules?
- *Research question 3:* How challenging is it to address student planning constraint compared to the instructor constraint?
- *Research question 4:* How many free time-periods must one put aside in order to assure high quality timetables?
- *Research question 5:* What is the numerical threshold for student enrollment overlap between two classes before it should be decided that those classes should not be offered during the same time slot?

In this work, I seek to answer the first four questions, while leaving the fifth question for future research. In order to address these questions, it is important to define the constraints that I need to place on the system such that it produces feasible schedules. Below are some of the hard constraints relevant to Macalester College:

1. No two classes scheduled during the same time-slot should share more than a given percentage of their pre-enrolled student set.
2. No two classes scheduled during the same time-slot should share the same instructor.

Below are some of the soft constraints that may be relevant to the chosen solution implementation:

1. Students who strongly favor one class instead of the other should get their top choice.
2. Computer Science 124 has to be taught in periods after 8:30 AM.

Though I present every constraint, I am more interested in the hard constraints and leave the inclusion of soft constraints for future work.

4 Design of the System

In this section, I introduce the algorithm I wrote to solve the UCTP problem. For the rest of this section, I analyze the genetic algorithm and the case-based reasoning systems separately, and then explore how to combine them together to produce the method I used to solve the problem at hand.

4.1 The Hybrid Algorithm

The algorithm I developed is hybrid because it combines genetic algorithms with case-based reasoning. Moreover, the genetic algorithm in itself is a hybrid algorithm because it combines group-based operators with self-fertilization operators during the crossover stage. I chose to combine genetic algorithms and case-based reasoning because scheduling time tables can be constructed based on previous schedules (Burke et al., 2001), and case-based reasoning is designed to solve current problems using past information. I used the genetic algorithm for two reasons: I needed a mechanism to populate the case-based reasoning system, and because if the case-based reasoning system fails to retrieve a good solution, then I would generate a feasible solution using the genetic algorithm.

In addition, for the genetic algorithm I chose to use the self-fertilization technique of Wang et al. (2009) as a crossover mechanism because that keeps the consistency of the solutions and improves the convergence of the algorithm. This occurs because it reduces duplicate classes in the schedule, and it does not completely change the position of the classes (as the genetic algorithm would have to do if it had implemented the traditional crossover operator). Consequently, using self-fertilization reduces the number of checks needed to move a class to a new time slot, thus, speeding up the system. Using self fertilization has another advantage: I do not need to mutate the solutions, as self-fertilization is much like a heavy mutation operator.

I also used concepts from group-based genetic algorithms - in the solution construction, both during population initialization and children generation. This approach does not allow more than one event to be scheduled to a cell of the matrix representing a given schedule – a matrix cell is either empty

or contain at most one event. Events are not allowed to be inserted into a position that causes conflict (Lewis and Paechter, 2005).

The activity diagram in Figure 8 illustrates the mechanism behind the hybrid system that I implemented.

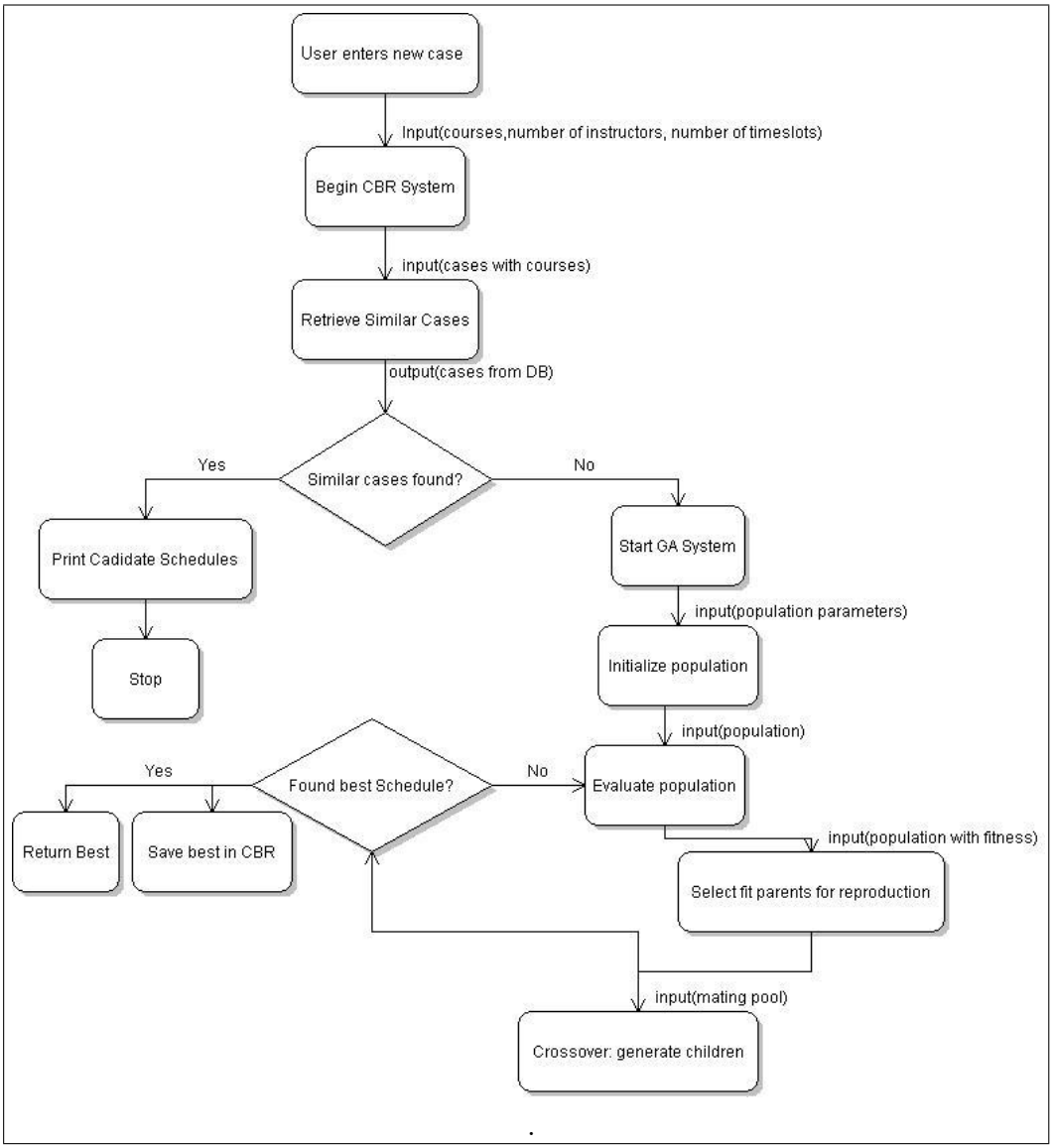


Figure 8: Hybrid system

4.2 The Genetic Algorithm

4.2.1 Solution Representation

The timetables are represented as a $m \times n$ matrix. where m is the number of available time slots and n represents the number of rooms available. Each cell in the matrix represents an event object. Events are comprised of the ID of the course, the ID of the instructor, the course and instructor names, the class size and the set of students pre-enrolled in that particular course. An event is a class without a time slot, and it represents the genes of the chromosome. As such, a cell in the chromosome matrix represents a class with a specified time slot and room assignment. Because I am not interested in the room assignment problem (as it is trivial to Macalester), the room's role in the matrix is to place a timetabling event to a virtual room and no actual room assignments are made. The matrix representation ensures that there will not be any room clashes (Raghavjee and Pillay, 2008). Figure 9 shows an illustration of a part of a possible schedule with two class events.

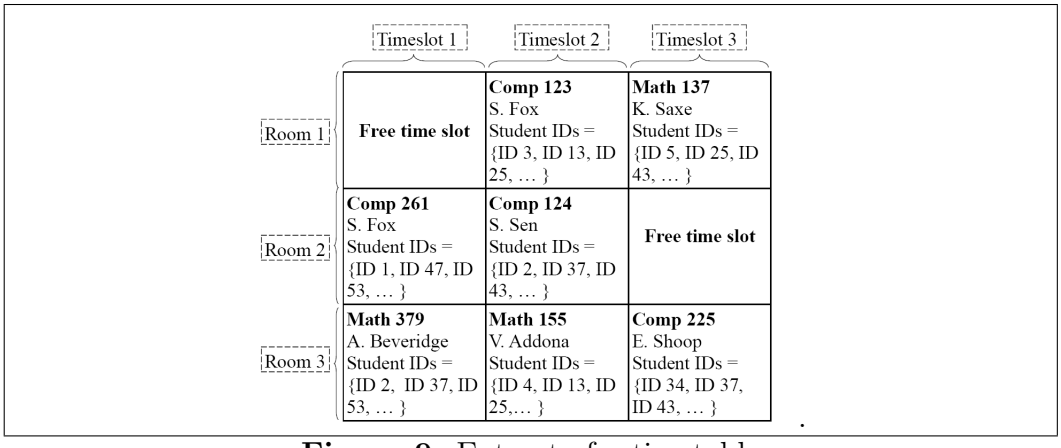


Figure 9: Extract of a timetable

4.2.2 Population Initialization

I began by collecting the courses to be scheduled, the number of available time slots, the number of available faculty, and the students who plan to take each of the offered courses. Before initializing the genetic algorithm, I

formed course events for each data piece using each of the courses. In the pre-processing phase, I produced as many of these events as the benchmark data allowed. Using the initial set of events, I then randomly picked a position in the schedule for each of the events. Note that if the position is taken, the algorithm looks for the closest free time slot for the event at hand. Algorithm 6 is a description of this process:

Algorithm 6: Genetic Algorithm Initiate Population Algorithm

```
Input: Set of events
Output: Initial population
begin
    foreach event in setOfEvents do
        random pickSlot1=randomNumber()
        random pickSlot2=randomNumber()
        checkIfAvailable(time slot[pickSlot1][pickSlot2])
        if checkIfAvailable == true then
            placeEvent(event)
        end
        else
            lookForNearestFreetime slot(time slot[i][j])
            if foundFreeTimeSlot == true then
                placeEvent(event)
            end
            else
                return Null
            end
        end
    end
end
```

Though I sought to avoid room and time slot conflicts during the initialization phase, the randomness of the process does not avoid the problem of student and instructor collision. Initializing the population at random was intentional so as to allow that infeasible timetables are improved upon during the evolution process Wilke et al. (2002).

4.2.3 Fitness Function

The fitness of the chromosome timetable is determined by the number of constraints the solution violates (Raghavjee and Pillay, 2008). In this case, I am interested in two particular constraints: the student and the instructor constraints, which are the hard constraints in this system. The fitness of any schedule is found by iterating through the classes in the timetable, and seeing if any of the course event placements violates the hard constraints. In this system, I chose to look for the global maximum, instead of the global minimum position and chose to attribute five points to a timetable that does not violate any constraint (two and a half points per constraint). Thus, the best solution will be one that at the end of the number of generations still has most of its points, preferably five or close to that. In order to assess the fitness of a timetable, I used a variation of the Jaccard similarity coefficient to calculate the penalty between two classes. The Jaccard similarity coefficient is used to measure the similarity and diversity of sample sets. The index is calculated by dividing the size of the intersection between the two sets by the size of the union (Goodall, 1966). Equation 2 shows the mathematical expression for the coefficient, where A is one set and B is the other set.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

This similarity index, however, has a shortfall in that it gives the same weight to both sets while calculating the index. To illustrate this point, let us build an example using the UCTP. Let us say that class A has 6 students and class B has 53 students, and imagine that the size of the intersection between the two classes amounts to 3 students. Then the Jaccard index will calculate the similarity to be 3/56, which is 0.054. This number would penalize these classes lightly because the intersection represents a small percentage of the total students for class B. Thus a more fair measure is to find the maximum of the ratio between the size of the intersection to the size of the chromosome classes. The equation 3 shows the mathematical representation of the modified penalty calculation, where SP is the penalty function between two courses(i and j) with a number students in common.

$$SP(i,j) = Max(\frac{|i \cap j|}{|i|}, \frac{|i \cap j|}{|j|}) \quad (3)$$

Below is the algorithm describing the evaluation process for a population:

Algorithm 7: Algorithm to measure fitness of candidates in the population

Input: The population
Output: The population with their fitnesses
fitness=5.0

```
foreach candidate in population do
    studentIntersectSize=0.0
    foreach row in candidate do
        foreach column in candidate-1 do
            class1Students = courseEvents[row][column].getStudents()
            class2Students = courseEvents[row][column+1].getStudents()
            instructor1=courseEvents[row][column].getInstructor()
            instructor2=courseEvents[row][column+1].getInstructor()
            foreach student1 in class1Students do
                foreach student2 in class2Students do
                    if student2 == student1 then
                        studentIntersectSize=studentIntersectSize+1
                    end
                end
            end
            if instructor1 == instructor2 then
                instructorPenalty = instructorPenalty + 1
            end
        end
    end
    fitness=fitness-max(studentIntersectSize/class1StudentSize,
    studentIntersectSize/class2StudentSize)

    fitness = fitness - (instructorPenalty/size(candidate.getTimetable()))
    candidate.SetFitness(fitness)
    populationFitness.add(candidate)
end
return populationFitness
```

The fitness function in mathematical form is shown in equation 4, in which n represents the number of courses in a timetable, SP is the student penalty, and IP is the instructor penalty(the number of courses that had the same instructor and were offered during the same time slot).

$$chromosomefitness = 5 - (\sum_{i,j} SP(i,j) + \sum_{i=1}^{n} \frac{IP(i)}{n}) \tag{4}$$

I measure the instructor constraint by counting the number of classes that are scheduled in the same time slot and have the same instructor. I find the number of classes in conflict, and divide by the total number of classes in that particular timetable. The instructor penalty is bound to be smaller than the student constraint, unless there are very few instructors teaching many classes. Five is the experimentally set maximum fitness value for a candidate solution. I decided to use five because it was the maximum fitness value that returned the least number of negative or zero fitnesses during my experiments. A maximum fitness value is important because the selection process needs to evaluate all the candidate solutions using the same scale, the maximum fitness sets the upper limit of the scale.

4.2.4 Selection

For the selection step of the genetic algorithm, I use the roulette wheel selection method. The roulette wheel is a fitness-proportionate selection method. As the name suggests, each candidate chromosome is given an area in the wheel according to its fitness. The mechanism is illustrated in Figure 10. Thus, chromosomes with higher fitness have a higher probability of being picked than chromosomes with lower fitness. The probability of an chromosome being selected to be among the parents that are allowed to reproduce (p_i) will take the form of the expression seen below:

$$p_i = \frac{f_i}{\sum_{i=1}^{m} f_i} \tag{5}$$

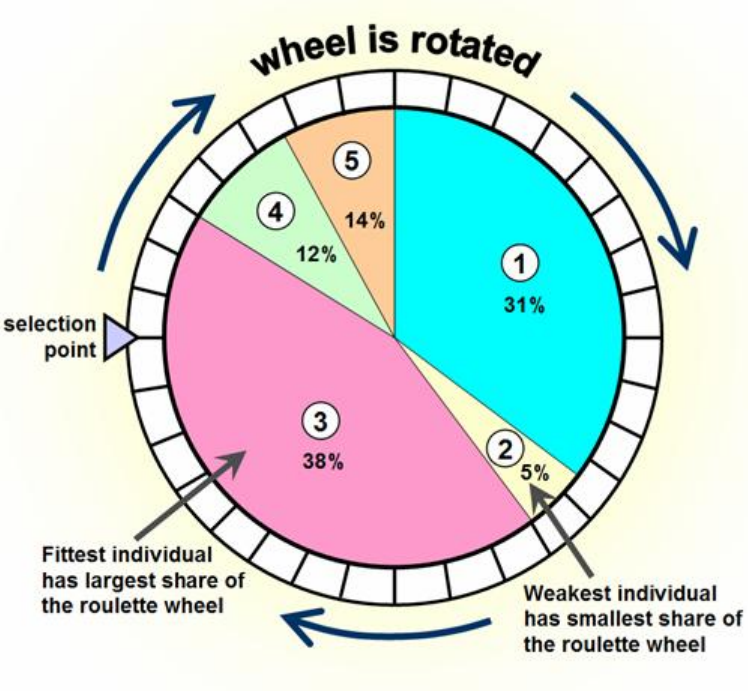


Figure 10: Roulette Wheel Mechanism. With permission of John Dalton (Dalton, 2010)

In the equation f_i is the fitness of a chromosome, and the f_m represents the fitness of the population added over each candidate's fitness.

One of the advantages of the roulette wheel selection is that it gives less fit chromosomes an opportunity to be selected unless their fitness is insignificant. To implement roulette wheel, I used an easy algorithm and chose a chromosome and its selection probability to check if its fitness is bigger the random number produced. If so, I placed it in the mating pool. If not, I added its fitness to the next candidate until the cumulative was a number that was greater than the random number, and then added the last parent (whose fitness made the cumulative bigger than the random number) to the mating pool. Algorithm 8 describes the algorithm I used for my selection mechanism:

Algorithm 8: Selection algorithm for the genetic algorithm

Input: Population size and Set of chromosomes and their fitness
Output: Mating pool

```
matingPool=new matingPool()
foreach chromosome in populationWithSelection do
    random randomNum=random() parent=null partialSum=0
    foreach chromosome in populationWithSelection do
        partialSum=partialSum+chromosome.getSelectionProbability()
        if checkIfAvailable == true then
            parent=chromosome matingPool.add(parent)
        end
    end
end
return matingPool
```

This implementation of the roulette wheel will benefit more fit chromosomes, but it is not hierarchical in nature. This is true because, depending on the value of the random number, some chromosomes with poorer fitness might be admitted into the mating pool. I purposefully decided to opt for this implementation because it widens the search space of the algorithm, and also helps measure how much improvement there is in terms of fitness per generation.

Selection is very important because it chooses the next chromosomes for procreation; however, it is the crossover mechanism that allows for procreation to take place. In the next subsection, I discuss the crossover mechanism I used in this genetic algorithm.

4.2.5 Crossover

I used the self-fertilization concept of Wang et al. (2009). In order to perform self-fertilization, the algorithm picks a time slot at random and extracts the class in it. It then iterates through the timetable matrix in order to place the class in an empty time slot. This process is different for a timetable with exactly the same number of time slots as the number of class events. In this case, because there will be no more available time slots, other than the one that the class has left opened, the best approach is to choose two

random classes out of two distinct time slots and swap their places. After swapping time slots takes place, the algorithm goes on to evaluate the child that was produced during crossover. There are two versions of the crossover mechanism. The first version accepts any child regardless of their fitness while the second version, the elitist approach, only places the child in the new population if the child has a better fitness than the parent.

Algorithm 9 describes the steps taken during crossover for the non-elitist version:

Algorithm 9: Non-elitist version of crossover

```
Input: Mating Pool
Output: New Population
listNewGeneration=new List()
foreach i in matingPool.size() do
    rand=random(matingPool.size) chosenParent=matingPool.get(rand)
    child=chosenParent.copy()
    foreach i in child.ChromosomeSet.size() do
        randIndex=random(parent.ChromosomeSet.size())
        chosenCourseEvent=parent.ChromosomeSet.get(randIndex)
        placeCourseinTimetable(chosenCourseEvent)
    end
    listNewGeneration.add(child)
end
return listNewGeneration
```

Algorithm 10 the steps taken during crossover for the elitist version:

Algorithm 10: Elitist version of crossover

```
Input: Mating Pool
Output: New Population
listNewGeneration=new List()
foreach i in matingPool.size() do
    rand=random(matingPool.size()) chosenParent=matingPool.get(rand)
    child=chosenParent.copy()
    foreach i in child.ChromosomeSet.size() do
        randIndex=random(parent.ChromosomeSet.size())
        chosenCourseEvent=parent.ChromosomeSet.get(randIndex)
        placeCourseinTimetable(chosenCourseEvent)
    end
    child.fitness=evaluate(timetable) if child.fitness ≥ parent.fitness then
        listNewGeneration.add(child)
    end
    else
        listNewGeneration.add(chosenParent)
    end
end
return listNewGeneration
```

The crossover operator is very important both for population diversity and fitness improvement. To ensure that I received the best results, I first used the non-elitist method and then switched to the elitist method. This caused an immediate improvement in the resulting timetables. However, the elitist method narrows down the search space, and though I might have received a very good solution, the solution could be a local maximum, instead of a global maximum.

During crossover, to improve the chances of convergence in reasonable time, I decided to allow any parent in the mating pool to create only one child in each generation. This is important because it guarantees that the population size will stay the same. This design choice is particularly important because the genetic algorithm is to be integrated with a web system, and web system can easily time out if a computation takes longer than the web framework time threshold.

Another design choice was to take the whole mating pool for the crossover

process, instead of the usual 80% that the literature advises. This change is related to the fact that parents are only allowed to generate one child. Thus in order to diversify the population and create a search space that is interesting, it is important that I include the whole mating pool in choosing parents for crossover. In addition, I chose not to use the mutation operator. Self-fertilization crossover is in its essence similar to a heavy mutation operator.

4.3 The Case Based Reasoning System

In this section, I describe the higher level design of the case-based reasoning system.

4.3.1 Case Representation

A case in the case-based reasoning system is represented by its ID. The list of course events (including courses, their time slots, and instructors), the number of slots available in the timetable, the number of rooms available in the timetable, the number of instructors available and weight or fitness of the case were all taken into account to distinguish one case from another.

4.3.2 Indexing

In order to measure the similarity between two cases, I chose not to design a whole new index, as an index is not organically different from a case. Thus, in this system the index is the same as the case itself. After comparing a new case to an existing case, I changed the similarity index of the old case to reflect the similarity between the new and old cases. Algorithm 11 describes the mechanism to compare two cases:

Algorithm 11: Algorithm to measure similarity between two cases

Input: New case and Old case
Output: Cases and their respective similarity indexes
begin
    oldCaseCourseEvents=oldCase.events()
    newCaseCourseEvents=newCase.events() score=0
    foreach *i in oldCaseCourseEvents.size()* **do**
        foreach *i in newCaseCourseEvents.size()* **do**
            if *oldCaseCourseEvents.contains(newCaseCourseEvents.get(i))* **then**
                score=+0.25
            end
        end
        if *oldCase.getNumberInstructors() == newCase.getNumberInstructors()* **then**
            score=+0.25
        end
        else if *oldCase.getNumberRooms() == newCase.getNumberRooms()* **then**
            score=+0.25
        end
        else if *oldCase.getNumberSlots() == newCase.getNumberSlots()* **then**
            score=+0.25
        end
        oldCase.setWeight(score) listRetrieved.add(oldCase)
    end
    return listRetrieved
end

4.3.3 Case Retrieval and Case Base Maintenance

The case-based reasoning memory is represented by a database table and the retrieval of cases is done through SQL queries. An example of such a query is:

```
select * from cbrcase
```

```
where  numslots= 12
AND numrooms= 8
AND numprofs= 9
```

Note that this query requires an exact match of the parameters between the new case and the retrieved case. This means that the algorithm finds two cases similar if they have exactly the same parameters. Moreover, the query is too broad, as it returns all the cases that match the criteria, more parameters are required to make this query more accurate. This makes the similarity measure too narrow, improvement is needed.An easy improvement over this query would be a more complex query that matches partially on different parameters.

Because of the data model discussed in the next section, this query returns the case and its attached course events when executed. This result is a slightly different form to represent a timetable than the genetic algorithm solution format. Algorithm 12 describes the retrieval process:

Algorithm 12: Algorithm to retrieve previous cases

Input: New case
Output: Retrieved solutions
begin
  List similarCases= new List List
  results=selectSimilarCases(newCase.numSlots,newCase.numRooms,
  newCase.numInstructors)
  foreach *oldCase in results* **do**
    computeSimilarity(newCase,oldCase) similarCases.add(oldCase))
  end
  return similarCases
end

I chose to do the case maintenance automatically by having the genetic algorithm populate the case-based reasoning memory. To do so, I ran some training trials, in which the genetic algorithm produces the best fitness chromosomes for every generation, and used these chromosomes as the old cases in our memory, which I compared against new cases.

4.3.4 Database model

In order to store the cases into the database, I devised a database model that is comprised of two entities: course event and the case-based reasoning memory. The former is used to stored the courses and their respective times and instructors, while that latter is used to store the numeric parameters for the case-based reasoning system. The relationship between a case and its respective course events is a one-to-many, because a case contains a timetable, which contains many course events. Figure 11 shows a database entity diagram that illustrates this relationship.

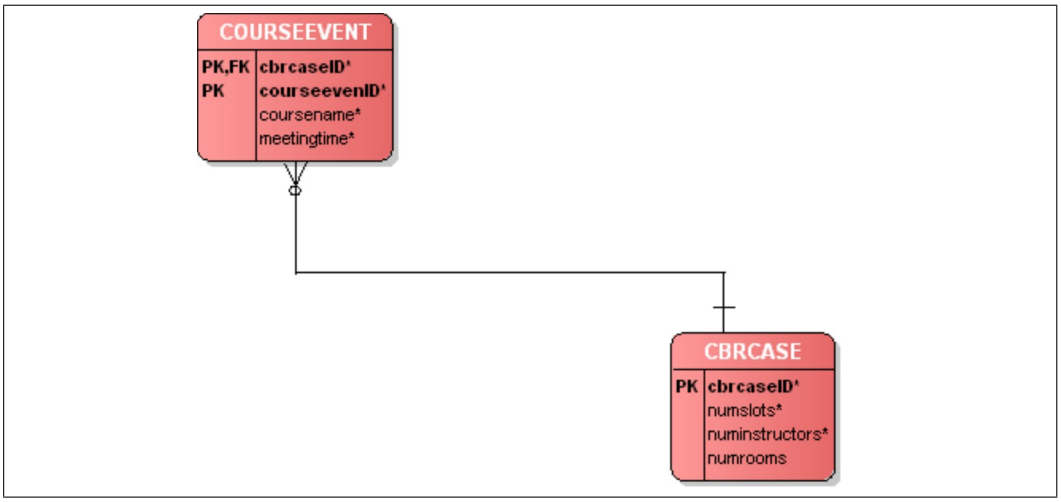


Figure 11: Entity relation diagram for case-based reasoning entities

4.4 Web Interface - MAP

The Macalester Academic Planner (MAP) is a web-application written in Groovy on Grails. This program creates a centralized system in which students can keep track of their academic plans, and departments can plan the courses to offer. The main advantage of this system is that under the layers of the web system, our algorithms extract both student and department course offering information to produce semester schedules that are compatible to both departments' timetabling needs and with students' plans. It is this automation that I hope will contribute to less laborious timetabling

activities.

MAP has three distinct functionalities: the student portal, the administrator portal, and the advisor (instructor) portal.

The student portal allows students to create, edit and share their major, minor and interdepartemental concentration plans. In the create interface, the student is exposed to every class of a particular major and all the respective course offerings during the students' four year matriculation. Thus, in order for a student to create a plan, the student will only need to click on the radio buttons to choose their respective classes and save that information.

After creating a plan a student has the option to either edit the plan or add advisors to her list of advisors. After adding the advisor(s), then the student can choose to share the plan with one or more advisor so the advisor can provide feedback. Figure 12 shows the actions that a student interacting with MAP can perform. The solid lines indicate activities that are in sequence while the dotted lines indicate alternative actions that the user could have taken from a given step.

The administrator can create new academic programs, such as majors, minors and interdepartmental concentrations. The administrator can also enter information about the semester and year in which courses will be offered. Lastly, the administrator will also use the make-schedule functionality to input the genetic algorithm parameters, run the algorithm on the parameters, and then view the results of the algorithm. This is the interface that interacts with the hybrid genetic algorithm, both to provide the algorithm with data and to display the results of running the algorithm.

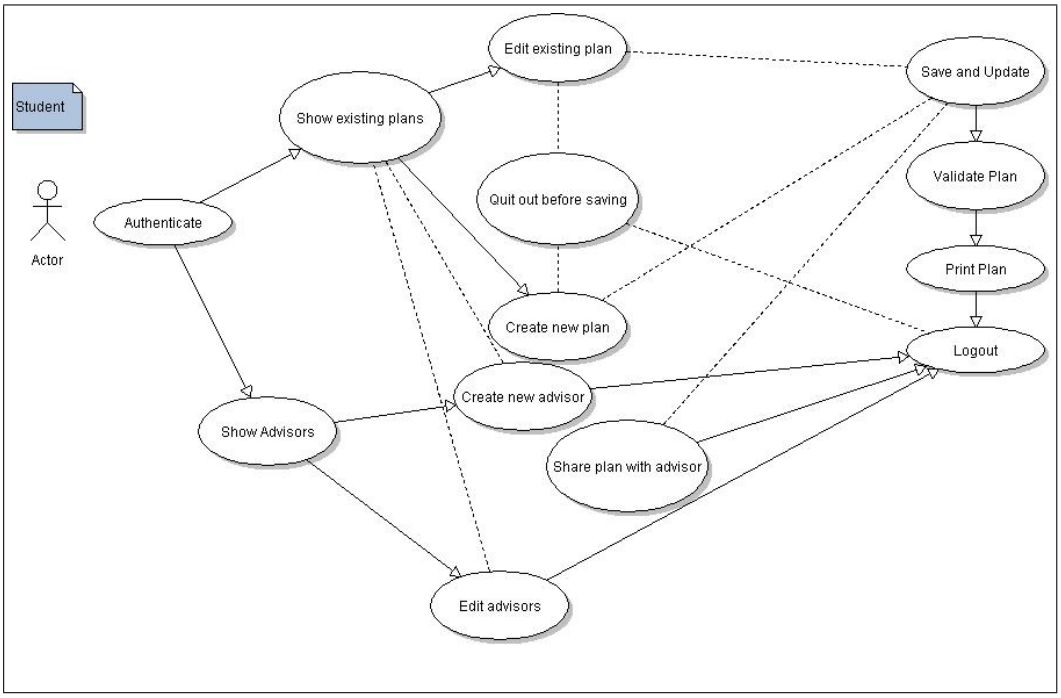


Figure 12: Activity diagram for a student in MAP system

Next, I turn back to the algorithms and I present some of the programming and design choices I made while programming the genetic algorithm and the case-based algorithm.

5 Algorithm implementations

All the implementations were written in Java. I chose Java mostly because the hybrid algorithm is to be integrated with a web framework that sits on top of Java libraries. For the rest of this section I will discuss the design choices I have made in implementing the algorithms.

5.1 Genetic Algorithm Implementation

I have chosen to make a course event the basic unit of this system or the gene of the genetic algorithm. An event is represented by a Java class that serves solely as a data holder, and has no other functionalities. The instance variables for the course event class are the course ID (an integer)and name (a string), the instructor ID (an integer) and name (a string), the capacity of the class (an integer) and the set of students for the course event. The set of students is represented using a Java integer typed list, where the integers represent the student ID. Course event is similar to the traditional concept of an academic class.

Another important Java class is the ScheduleChromosome class. This is a class that holds all the data related to a timetable. It contains a 2D array that holds the actual timetable. The array is of type event. It also contains instance variables such as the fitness and the selection probability, which are double, and the number of rows and columns which are integers. In addition, it contains a list of events, which is used as a history data structure that keeps track of all the course events in the timetable.

The implementation of the genetic algorithm can be divided into two services: the algorithm helper services and the evaluation service. The algorithm helper functionalities are comprised of methods that implement the stages of the genetic algorithm: population initialization, selection and crossover. These functionalities are contained in the Java class GaUtil (for further description see appendix A).

To initialize the population I chose to take the list of course events and randomly assigned these events to the timetables for each chromosome. Following this process I created as many chromosomes as the population size parameter allowed.

For crossover I decided to implement two versions: the non-elitist and the elitist version. The elitist version is useful,because it always converges and returns the best possible timetable, even though it strongly biases the evolutionary process (it cuts out the least fit chromosomes in early generations).

The children created during crossover are deep copies of the parent Java

objects. I chose to make deep copies so that the changes in the child would not affect the parent. In doing so, I make sure that each Java object points to different memory locations in the Java heap.

The second group of services are represented by the GaConstraint Java class and these are the evaluation functions. The student and instructor constraint functions build the fitness function, with which we evaluate chromosomes in the population (see appendix A).

5.2 Case-based Reasoning Implementation

In implementing the case based reasoning system two important features were considered. First, the mode of storage of the cases, second, the retrieval process.

As previously discussed, cases in the case-based reasoning system are permanently stored to a database. In order to implemet this design choice I used Java JDBC library. This library allowed me to get a database connection, to execute queries and get the results of the database queries.

Each entity in the database corresponds to a java class, the course event table corresponds to the course event class. As I defined previously, the course event class is the Java class that encapsulates information about a course event, that is the course name and the time that course meets. The cbrcase table corresponds to the CBRCase Java object. The CBRCase class encapsulates all the pieces of information about a case. This includes the actual timetable for the case, which is comprised of a list of course events. It also contains other fields such as the number of rooms and number of instructors for that given timetable.

The implementation of the case-based system can be divided into two parts: The case-base initialization services and the case-based reasoning fuctionalities. I decided to use the genetic algorithm as a training tool for the case-based reasoning system. In order to attain variety in the cases in the case-memory I decided to use different benchmarks and ran these on the genetic algorithm. The CBRTraining class is responsible for calling the genetic algorithm, which in turn saves its feasible chromosomes into the case-base

(see appendix A).

There are many classes supporting the case-based reasoning mechanism, but the main ones are the CaseBase, CaseMemory and CBRCase classes. The CaseBase class offers services to save and retrieve cases, while the CBRCase class is a data-holder class with no additional functionalities. The CaseMemory class implements the Java JDBC services and persists cases to the database, and manages the database. It allows us to retrieve, save and delete case-based reasoning cases.

During the implementation of the algorithms active testing was crucial in fine-tuning parts of each algorithm. The goal of these implementations was to perform experiments that would either support or refute the use of these algorithms for the task of production of timetables for Macalester College. Next, I am going to discuss some of the experiments I set up to answer the research questions.

6 Experiments

In order to answer the research questions, I designed experiments that measure the performance of the case-based reasoning system and the genetic algorithm. The case-based reasoning experiments only test the retrieval process, further testing is needed in order to thoroughly test the system. For the genetic algorithm, I have prepared a number of experiments that reveal the different properties of this algorithm.

6.1 Test Data

I designed testing benchmarks using knowledge about scheduling at Macalester College. I selected twenty courses from the Mathematics, Statitics and Computer Science Department, as well as the teachers for these courses. I also assigned a number of students to each of these courses. I generated the student IDs randomly. These IDS are used to check whether a student is in more than one class or not. In order to assign students to the classes, I randomly selected student IDs and placed them in the course. Assigning students to

classes is not realistic because students are likely to pick classes with a probability that is proportional to the priority of the class. Below is a snippet of the Events.XML document, which is the document that holds course event data for the 20 courses I have chosen. The variables in this document define a course. A course is comprised of a suject course that has an ID and name, an instructor that also has an ID and name, the number of students that intend to take the course, and the number of times the course is offered each week. To view the whole document refer to appendix B.

```
----------
Events.XML
----------
<events>
   <event id="1">
      <courseID>1</courseID>
      <courseName>Comp 120</courseName>
      <instructorID>1</instructorID>
      <instructorName>Elizabeth Shoop</instructorName>
  <classSize>20</classSize>
  <frequency>3</frequency>
   </event>

   <event id="2">
      <courseID>2</courseID>
      <courseName>Comp 121</courseName>
      <instructorID>5</instructorID>
      <instructorName>Eric Theriault</instructorName>
  <classSize>36</classSize>
  <frequency>3</frequency>
   </event>
   </events>
```

Here is the simple algorithm that assigns students to course events:

Algorithm 13: Algorithm to assign students to courses

```
Input: listOfCourses
Output: listOfCoursesAndStudents
begin
    foreach course in listOfCourses do
        classSize = course.classSize foreach i in classSize do
            Random rand = new Random() studentID =
            rand.nextInt(classSize) course.listOfStudents.add(studentID)
        end
        listOfCoursesAndStudents.add(course)
    end
    return listOfCoursesAndStudents
end
```

In addition to the course event XML document, I designed the XML document that holds the parameters for the genetic algorithm. I called this document the GAbenchmark document. This document is comprised of four numbers that define the behavior of the genetic algorithm. These numbers are the population size, the number of generations, number of available course times and the number of available rooms. Below is a sample GAbenchmark document:

```
----------------
GA_benchmark.XML
----------------
<parameters>
<parameter id="1">
<parameterName>POP_SIZE</parameterName>
<value>100</value>
</parameter>
<parameter id="2">
<parameterName>NUM_GENERATIONS</parameterName>
<value>100</value>
</parameter>
<parameter id="3">
<parameterName>NUM_time slotS</parameterName>
<value>6</value>
</parameter>
```

```
<parameter id="4">
<parameterName>NUM_ROOMS</parameterName>
<value>6</value>
</parameter>
</parameters>
```

Next I am going to show how I used these files to perform the experiments.

6.2 Experiments on the GA System

In order to test the performance of the genetic algorithm, I designed a few instances of the UCTP problem with the aim of testing different aspects of the genetic algorithm. Here are the problem instances from which I designed the testing benchmarks:

1. An instance when the number of available time slots is much higher than the number of course events we need to schedule. This setup helps us see how hard the algorithm works for very easy problems.

2. An instance when the number of time slots is slightly higher than the number of classes to schedule. In this experiment I chose to allocate five more time slots than the number of classes. This choice seemed realistic enough to mimic a real-life scheduling situation. This tests how much improvement the algorithm offers when the problem is more constrained. However, in this case the problem is not very hard either.

3. An instance when the number of time slots available is exactly equal to the number of classes to schedule. This is the most constrained case, as the algorithm has no extra room in which to place conflicted class arrangements. This tests not only fitness improvement, but also the algorithm solution quality for extremely hard problems.

4. An instance when the population grows. This tests how population of solutions growth affects the time and the quality of the solution that the algorithm returns.

5. An instance when we leave the algorithm running longer. In this case

I test whether a greater number of generations result in better quality solutions without increasing computation time.

Using the XML files I described in the previous subsections, I have prepared six different benchmarks using six different GABenchmark documents and only one Events.XML file. I chose to use only one Events.XML file, because I want to use the same twenty courses across all the experiments. This is a reasonable choice because the twenty classes I picked are reflective of a variety of courses that Macalester College offers (in terms of size).

- *Benchmark 1:* the population size is set to 100 chromosomes this is the number of chromosomes that the genetic algorithm has to randomly produce in the initialization phase. The number of generations is set to 100. This means that the algorithm will run at least 100 times, it may run up to 400 iterations if the fitness of the best individual for each generation is lower than my threshold of 4.70. There are 20 courses to schedule, each with 1 instructor, two courses may share the same instructor. The number of available rooms was set to 6 and there are 6 possible time slots in which we could place any course event. Putting together the number of rooms and the number of available times, there are 36 different combinations of rooms and times that could be generated, thus 36 time slots would be available for the 20 courses we want to schedule.

- *Benchmark 2:* I only changed the number of rooms available to 6 and the number of times available to 5. This means that the number of available time slots were reduced from 36 to 30 in total. This is a harder problem than the one in benchmark 1, because there are less free time slots in which the algorithm could place conflicted classes.

- *Benchmark 3:* I further reduced the number of available times to 4 and the keep the number of available rooms as 6, this produces the a total of 24 available time slots for 20 classes. This is a harder case the previous one, however, it still gives the algorithm some freedom to move solutions to free spaces.

- *Benchmark 4:* I further reduced the number of of available rooms to 4 and the change the number of available times to 5, this produces the a total of 20 available time slots for 20 classes. This is a very constrained

situation as the algorithm has no free room in which to place courses that clash.

- *Benchmark 5:* I increased the population number parameter to 1000, while maintaining the number of generations to 100. The aim of this change is to evaluate timing and computational needs of the algorithm as the initial population increases.

  I divided this benchmark into two parts:

  - *Benchmark 5A:* is comprised of the instance of the UCTP in which there are 36 timeslots for 20 courses.
  - *Benchmark 5B:* is comprised of the instance of the UCTP in which there are 20 timeslots for 20 courses.

  I predicted that benchmark 5 would give us better insight on the performance of the algorithm , as it allows us to compare this performance with the performance of the algorithm when the population is smaller. My prediction was that benchmark 5B would produce both lower population fitness average, as well as the best solution found would have lower fitness than that of the instance in benchmark 5A.

- *Benchmark 6:* I set the population size to 100 and increase the generation number to a 1000, while keeping all the other parameters the same in bechmarks one though four. The aim of this change is to test how the quality of solutions change as the algorithm runs for longer time. Similar to benchmark 6, I decided to divide this benchmark into the same two parts: A and B, similar to what I used in benchmark 5.

I designed the experiment such that the algorithm ran on each benchmark. I measured the average fitness per generation, the best fitness per generation, and the number of new chromosomes the algorithm generates per generation. The average fitness per generation is measured by adding the fitness of all the chromosomes in a generation and dividing it by the number of chromosomes in the population. Below is the mathematical representation of this calculation. In the formula *fav(i)* is the average fitness function, i is the chromosome in the population and there are n of them in a generation.

$$fav(i) = \frac{\sum_{i=1}^{n} fitness(i)}{n} \quad (6)$$

I calculate best fitness per generation by finding the fitness of the best chromosome of a generation, and I do that for every generation. The distribution of fitnesses is measured by counting how many chromosomes in the population have that particular fitness value for every generation. The number of new chromosomes per generation is measured by counting the number of unique chromosomes (new Java objects of type chromosome) in each generation.

6.3 Experiments on the case-based reasoning system

In order to evaluate the case-based reasoning system I used the same benchmarks as the ones described in the previous subsection and ran the genetic algorithm in order to populate the database. I wrote an algorithm that randomly picked a combination of course events and genetic algorithm parameters to produce optimized schedules. Then I used these optimized schedules to populate the case-base. Below are the steps to populate the case-base memory.

1. Randomly choose an events.xml document - these documents are numbered from 0 to 4.
2. Randomly choose a GA benchmark XML document (I created one per benchmark)
3. Initiate the GA to create solutions
4. Save solution in the case
5. Repeat the steps above for the number of benchmark documents.

Once the case-base memory is populated, then the system queries the database for a new case. For experimental purposes I chose to use data from one of the XML benchmarks that I knew for sure that I had already inputted into the case-memory. When designing the method to populate the case-base I grappled with two questions: Will the UCTP solutions that the genetic al-

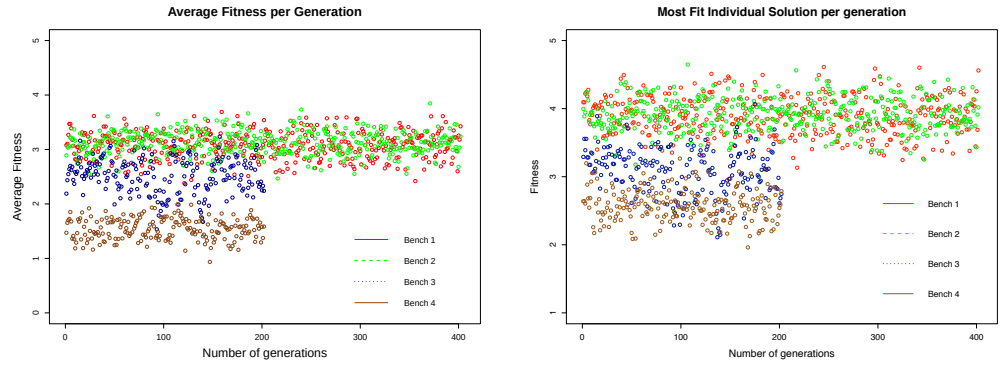
gorithm will provide to the case-based reasoning system be diverse enough? If not what other technique can I implement to increase the diversity of the cases? These questions were left as future research questions.

The experiments were performed in a Ubuntu Dell box with 8 GB of RAM and a processor with 3.5 MHz of speed. This computer was the only one that had enough memory to carry out the experiment for the sixth benchmark. In the next section I will show some of the obtained results and I will discuss the implications of the results.

7 Results and Discussion

I ran the non-elitist algorithm on benchmarks 1 through 4, and I graphed the fitness of the most fit individuals per generation as shown in Figure 13(a). The graph shows that the overall fitnesses of the best individuals changes in a random pattern from generation to generation. This pattern might e explained by that the algorithm does not save the best individuals seen so far, thus, it finds a best that is actually lower than the previously found. The flatness of the overall trend for of the benchmarks, show that the algorithm is not learning about the best possible individuals over time.

Furthermore, I have plotted the average fitnesses for each generations as shown in Figure Figure 13(a) . Similarly, the overall trend for the plotted points is flat. This means that the non-elitist version does not produce fitter populations over time. This result renders this version of the algorithm less suitable for my purposes.



(a) Average population fitness per generation across benchmarks 1,2,3 and 4

(b) Best population fitness per generation across benchmarks 1,2,3 and 4

Figure 13: Best and average fitness values for elitist genetic algorithm

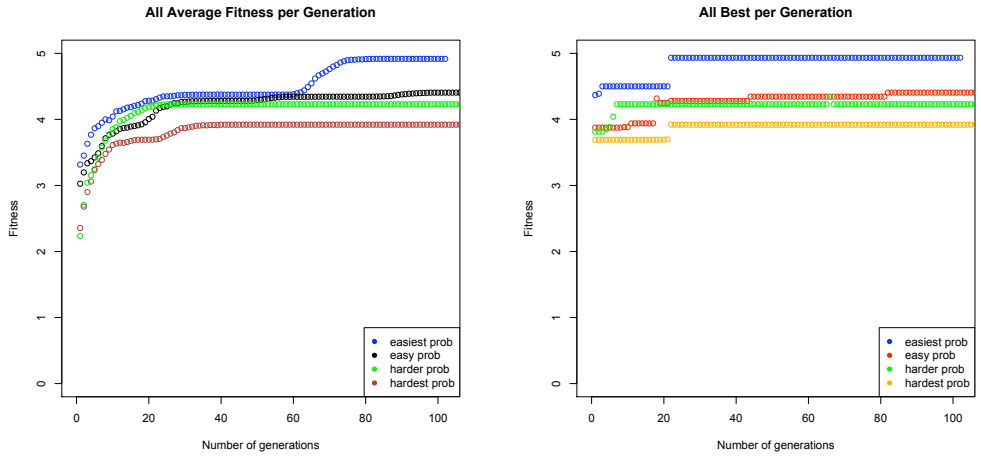
Across benchmarks 1 through 4, the elitist version of the algorithm successfully finds a solution that mostly satisfies the constraints. Though some of the solutions are only suboptimal, they still offer a timetable with low conflict. Figures 14(a) and 14(b) show the fitness improvements over the number of generations for the four benchmarks.

Figure 14(a) shows the improvements over time in the average fitness of the population. The exponential-like behavior of the curve tells us that as the number of generations grow, the algorithm improves the best for each generation. Because of the elitist nature of the algorithm, it passes the top candidates seen so far to the next generation, thus, if each generation does not produce a better best chromosome, then the best seen so far becomes the final solution.

For most of the benchmarks it takes the algorithm thirty generations to find the global best chromosome. However, in the case of the benchmarks 1 and 2, which are the problems with 36 and 30 time slots available, it takes the algorithm between 60-80 generations to converge into the solution. this so happens because the algorithm has more free time slots in which to place colliding courses. As such, there is a greater number of combinations that can lead to a greater number of more fit solutions. As result, the search space takes longer to shrink than in more constrained cases.

As predicted, the blue line shows us that the easiest problem finds solutions with fitnesses above any other benchmark. This so happens because the easiest benchmark (benchmark 1) has 16 free time slots, that is 80% more than the number of course events to schedule. This means that the algorithm has more free time slots in which to allocate conflicting courses. In addition, I expected this instance to be the one in which the machine would learn less, due to its easy nature. However, the graph shows us that the algorithm does learn, even in the easiest case. This might be so because the algorithm first produces low fitness chromosomes, thus as it explores the search-space it automatically learns about more fit chromosomes. The red line corresponding to benchmark 4, the hardest benchmark among benchmarks one through four, shows the lowest results. That was expected in that the algorithm has no rooms in which to re-allocate conflicting courses. As result, conflicts are very hard to avoid. Note that the benchmarks easy to hardest correspond respectively to benchmarks one through four.

Furthermore, the graph of the fitness of the best chromosomes per generation has a similar behavior to that of the average fitness. That makes sense because the fitness of the best chromosomes contribute with a larger investment to the overall fitness of the population in a given generation comparing to less fit chromosomes. One difference between the average and best fitness graphs is that the algorithm converges to the final solution earlier in the best fitness case than in the average case. One observation that can be made is that regardless of how constrained a benchmark is, the algorithm will always find the solution within the first twenty generations. My hypothesis is that as soon as the algorithm quickly finds a local best solution, and because the algorithm passes the best chromosomes from one generation to the next, the best chromosome eventually dominates the whole population.



(a) Average population fitness per generation across benchmarks 1,2,3 and 4

(b) Best population fitness per generation across benchmarks 1,2,3 and 4

Figure 14: Best and average fitness values for elitist genetic algorithm

In another experiment used benchmark 5. This experiment aimed at testing the performance of the algorithm when population grows. The algorithm yielded results comparable to those obtained for benchmarks one through four. The blue line in Figure 15 show that the easiest problem still yields solutions with higher fitness than the hardest problem. These results are different from those for benchmarks 1 through 4 in that the algorithm has a very slow learning process. We can see that from the flatter curves in Figure 15, while the graphs in Figures 14(a) and 14(b) show steeper curves.

Given that every parent is considered during the reproduction process, and because the overall population offers more candidate solutions than in previous benchmarks, I hypothesize that the flatness of the graph in Figure 15 has to do with the fact that the algorithm on average finds fitter best chromosomes than those in benchmark 1 through 4. This is so because there is a greater likelihood of getting a higher fitness best chromosome in the population of 1000 chromosomes, than in the population of 100 chromosomes.

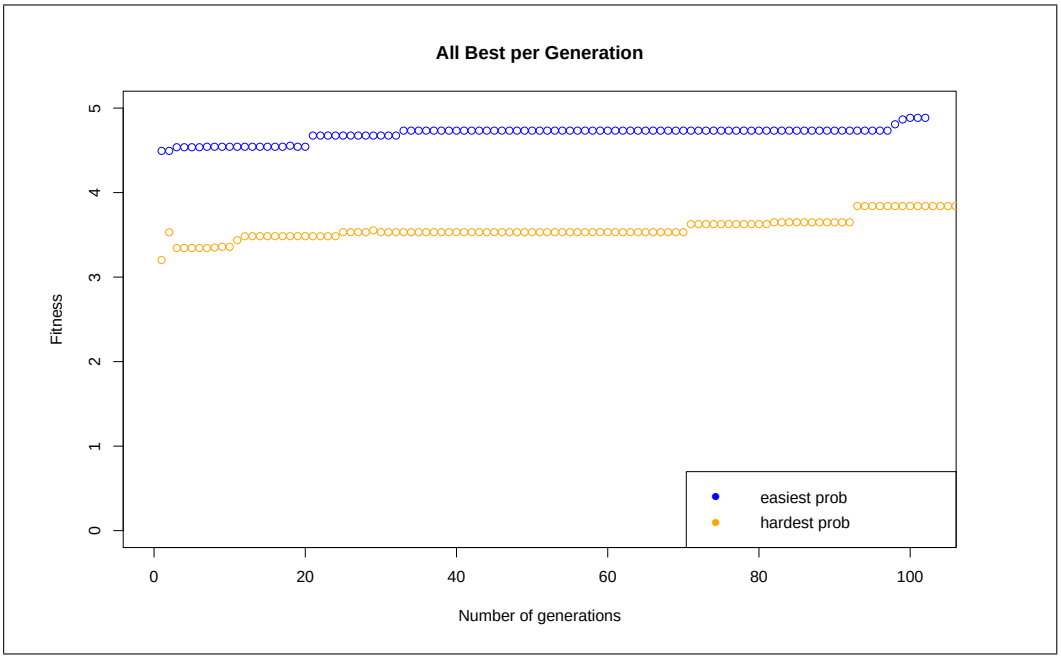


Figure 15: Best fitness per generation for population of size 1000

Next, I looked at how the algorithm responds when I increase the number of generations. In order to do that, I used benchmark 1. As we can see in Figure 16, similar to previous benchmarks, the algorithm has a gradual learning process. It starts with lower fitness best chromosomes, and as it explores the search space it finds the fitter best chromosomes. Note that for a 1000 generations the algorithm takes longer to find the final solution than when we run it on 100 generations. We can see from Figure 16 that it takes the algorithm about 450 generations to find the highest fitness chromosomes, while Figure 14(b) tells us that it takes the algorithm roughly 20 generations. This difference is crucial in terms of performance, as it would take longer time to find the solution using 1000 generations, than it would if we ran the algorithm on 100 generations, which might be undesirable, if we are to incorporate this system with a web system. The advantage of running in 1000 generations is that it better illustrates the gradual nature of the learning process of the algorithm, which can be seen from slow and steady rising curves in the figure. This gradual learning is harder to see when we run on 100 generations only.

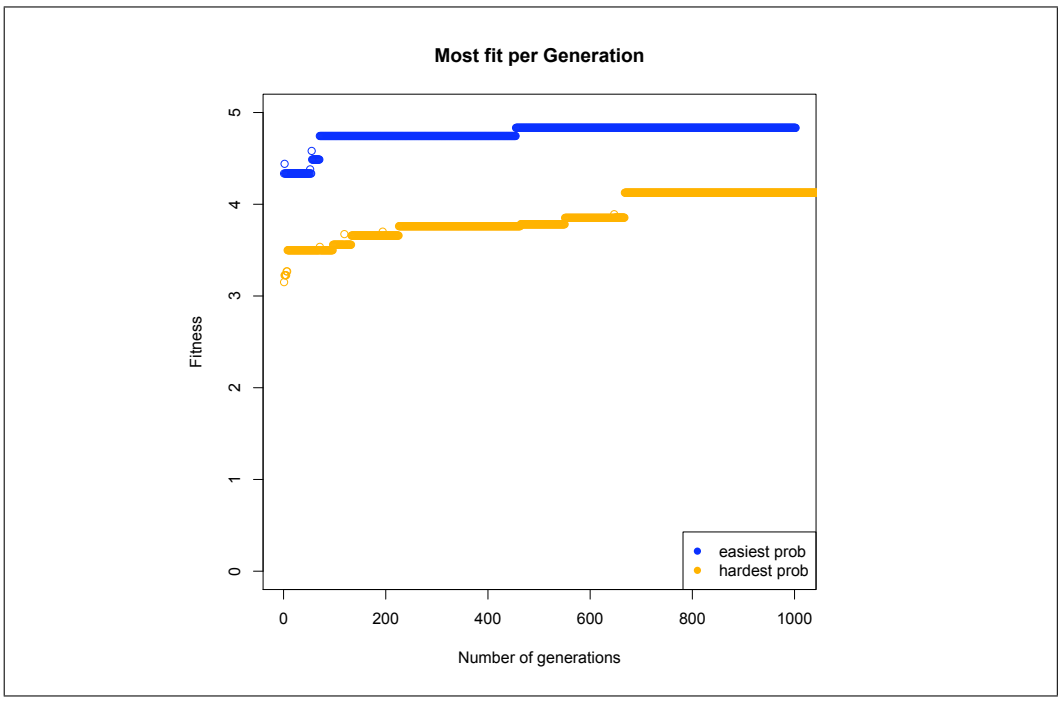
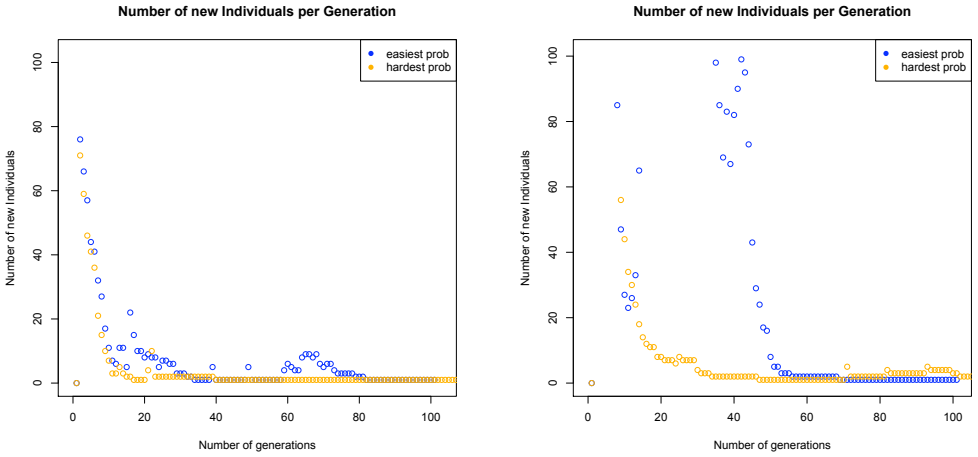


Figure 16: Best fitness per generation for a 1000 generations

Besides measuring the performance of the algorithm, I was also interested in learning about intrinsic properties of the algorithm. In order to do so, I measured the number of unique chromosomes added in each generation, and the time it takes to execute the different benchmarks. Figures 17(a) and 19(a), show that the number of unique chromosomes added to each generation falls exponentially for benchmarks 1 through 5. This means that as the number of generations increase the less diverse the population becomes. There are two implications of the lower population diversity: The first is that it indicates that the algorithm is converging into the fittest chromosome or set of chromosomes that are then provided as a solution. Second, it shows the biased nature of the genetic algorithm. Because we keep the best five chromosomes from the previous population, and also because we only add a child if it is better than a parent, or else we add the parent back into the population. Diversity decreases because eventually the strongest chromosomes ends up dominating the population. Note however that this does not mean that the algorithm does not explore the whole search space, it only means that it leans more towards the stronger chromosomes, quickly excluding the worse

chromosomes. Figure 18 shows a sample solution produced by the algorithm using benchmark three. This particular timetable had a fitness of 4.73 out of 5. From Figure 18, we can see that the penalty on the solution is mostly generated by the violation of the student constraint.



(a) New chromosomes per Generation for benchmarks 1 to 4

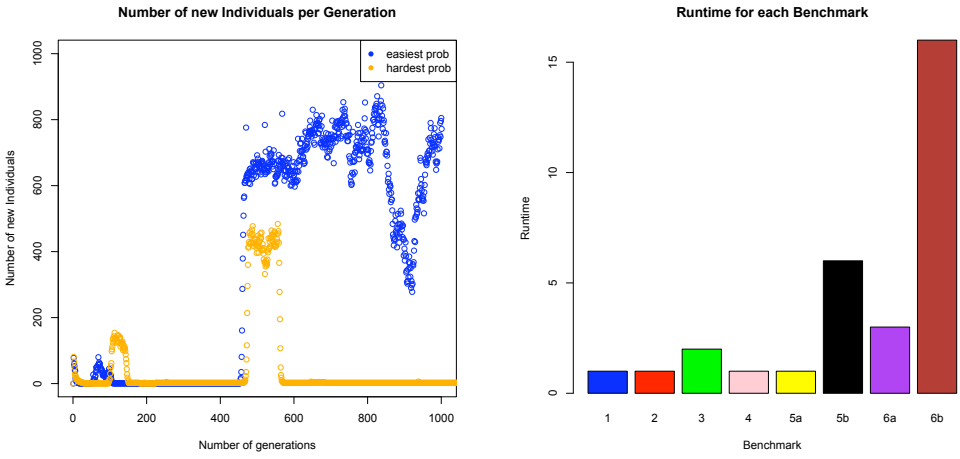
(b) New chromosomes per Generation for benchmarks 5a and 5b

Figure 17: Figures showing unique elements for all of the benchmark

Times	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30 AM		Math 236 Tom Halverson, CS 482 Susan Fox		Math 236 Tom Halverson CS 482 Susan Fox	
8:30:9:30 AM	Math 135 Tchad Topaz Math 136 David Bressoud Math 365 Danny Kaplan		Math 135 Tchad Topaz Math 136 David Bressoud Math 365 Danny Kaplan		Math 135 Tchad Topaz Math 136 David Bressoud Math 365 Danny Kaplan
9:40-11:10 AM		Comp 121 Eric Theriault Math 461 Tchad Topaz Math 237 Dan Flath Comp 342 Shilad Sen		Comp 121 Eric Theriault Math 461 Tchad Topaz Math 237 Dan Flath Comp 342 Shilad Sen	
9:40:10:40 AM	Math 412 Alicia Johnson Math 137 Karen Saxe Math 108 Dan Kaplan		Math 412 Alicia Johnson Math 137 Karen Saxe Math 108 Dan Kaplan		Math 412 Alicia Johnson Math 137 Karen Saxe Math 108 Dan Kaplan
10:50-11:50 AM	Math 312 Dan Flath Comp 120 Libby Shoop Comp 240 Susan Fox		Math 312 Dan Flath Comp 120 Libby Shoop Comp 240 Susan Fox		Math 312 Dan Flath Comp 120 Libby Shoop Comp 240 Susan Fox
12:00-1:00 PM	Comp 225 Libby Shoop Comp 221		Comp 225 Libby Shoop Comp 221		Comp 225 Libby Shoop Comp 221

Figure 18: Output of the algorithm with fitness of 4.73

Contrary to the results from previous benchmarks, the results for benchmark 6 did not conform to my expectations. As we can see from Figure 19(a) the algorithm behaved as expected until about the first 100 generations. The longer the algorithm ran, the results for benchmarks 6a and 6b diverged. The former generates many more new chromosomes between generations 500 and 1000 than the latter. The behavior of benchmark 6a is actually different than the behavior of any other benchmark. In every other benchmark the algorithm converges to one strongest candidate, while for benchmark 6a the algorithm ultimately finds a set of strongest candidates (showing greater diversity than the other benchmarks).



(a) New chromosomes per Generation for benchmarks 6a and 6b

(b) Runtime for benchmarks

Figure 19: Figures showing unique elements for all of the benchmark and runtime per benchmarks

Another property intrinsic to the algorithm was the algorithm runtime. For most of the benchmarks the algorithm runs under 3 seconds, however for the hardest of all benchmarks, benchmark 6b, the algorithm runs for about 16 seconds. Figure 19(b) shows the times for the different benchmarks. Given the time constraints imposed by web systems, the algorithm should be set up to run only benchmarks one through four. With these benchmarks the algorithm runs under 3 seconds, which is desirable for a system that is to be integrated with a web system.

In order to investigate which constraint taxes solutions more harshly, I decided to measure the penalty that the system applies to every chromosome in the search-space. Figure 20 shows the graphs of the student and instructor penalties for chromosomes in the search space. The experiment was done using the benchmark 4. The graphs tell us that the student constraint taxes chromosomes more harshly than the instructor constraint does. The student constraint subtracted between 1.5 to 4 points off of the fitness of chromosome solutions. In contrast, the maximum penalty for the instructor constraint is around 1 fitness unit. If we look at the density of the graphs, we can observe that the highest density is around 0.5 for the blue points (points that rep-

resent the instructor constraint), and the density for the student constraint is in the interval from 2 to 3 (as shown by the red points in the graph). Thus, by observation, the student penalties are almost twice as larger as the instructor penalties. Intuitively, a course has only one instructor, but it however has a larger number of students. Because of that, the probability of student collision is higher than the probability of instructor collision. Extending this analysis to other constraints I have not implemented, it makes sense that the student constraint would be harder to satisfy than others, because while room and instructor constraints focus on one variable (room and Professor), the student constraint deals with a set of variables (each student is a variable). This makes the student constraint multidimensional, and the instructor constraint unidimensional. This observation answers research question 3, and concludes that the student constraint is harder to satisfy than the instructor constraint.

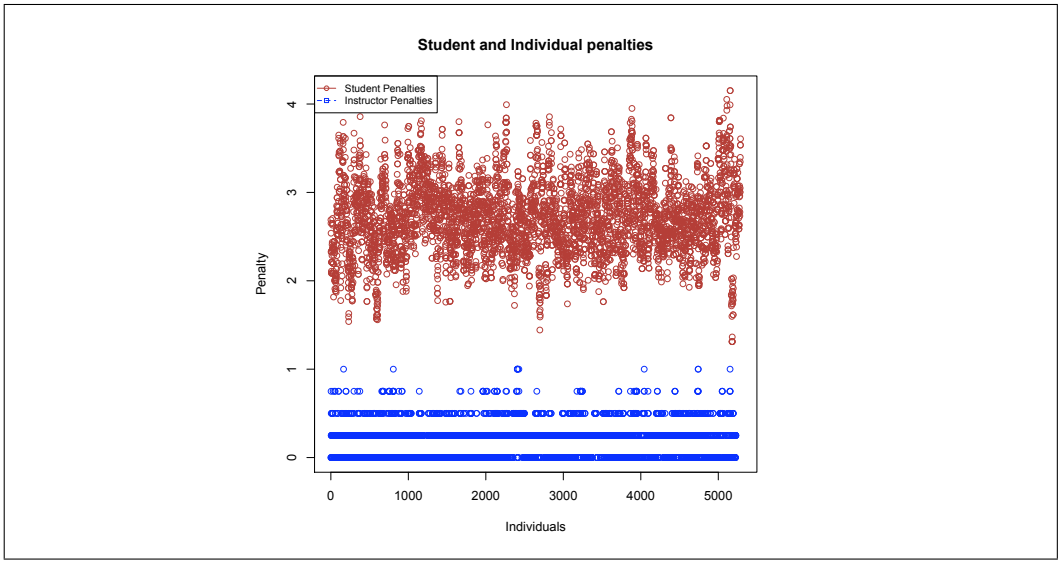


Figure 20: Student and Instructor penalties for each chromosome in the search-space

The results for the elitist version allow us to positively answer research questions 1, 2 and 4. It is possible to generate timetables that accommodate administrative needs and students' planning needs. However, it is important to note that the system has its limitations, one being that it will give more or less optimal solutions depending on how constrained the problem is. In

real terms, if a department wants collision-free timetables, then they must reserve some open time slots. From my observations, the algorithm returns a fitness of 4.7 or higher if one sets aside four or more free time slots. For a collision free timetable, one would need to set aside at least fifteen free time slots. On the positive side, even if the department has no free time slots, the algorithm will still return a near to collision-free timetable, which would require only few changes. Another observation is that from experiments I have found that in order for the algorithm to run with lowest runtime and attain the highest timetable quality, it needs to apply any of the first, second, third or fourth benchmarks.

Because of its supporting nature, the case-based reasoning acts as a secondary system in this project. I have not considered a comprehensive study of its properties. As such, I tested the case-based reasoning system as described in the experiment, and I found that if the case has been saved into the case-base, then the system will always retrieve it (a hit rate of one). Because the system retrieves cases based on the parameters of a case, we are guaranteed to get potential solution candidates, if a new problem matches any of the parameters of a case in the case-base.

8 Conclusion and Future Work

This project created a hybrid genetic algorithm that combined a genetic algorithm and a case-based reasoning algorithm to solve an instance of the UCTP that is aware of students' scheduling needs. I implemented two algorithms: the elitist algorithm, and the non-elitist algorithm. The elitist algorithm had a better performance that the non-elitist version for all of the test benchmarks. Below are some of the findings of my study:

- The elitist algorithm found feasible solutions even for the most constrained benchmarks. The quality of the solution depends largely on the parameters of the elitist genetic algorithms. Experimental values suggested that the algorithm performs best when the population size and the number of generations are set to 100, and the number of available time slots for 20 course events is set to 30 time slots (though it produces good results even for 24 available time slots).

- Moreover, I found that regardless how constrained a problem is, the elitist algorithm always improves the overall fitness of a population over a fixed number of generations. This signifies that the algorithm is actually learning about the best chromosomes in the search space.

- In addition, I found that the student constraint is harder to satisfy than the instructor constraint. This is so because the the instructor constraint only evaluates for one variable (the instructor), the student constraint evaluates for a set of variables (one variable per student for many students). Because of its multidimensionality, there is a greater chance that a student constraint will be violated more often than the instructor constraint. Extending this knowledge to other constraints, I hypothesize that the student constraint will be harder to satisfy than any other unidimensional (or single variable) constraint.

One area of future research is the data design and collection process. As previously stated, I designed my own benchmarks, based on knowledge of scheduling at Macalester. However, in order to get more realistic results, I would need to collect real data from Macalester College students and administrators. I have introduced the MAP system as the system that will allow us to improve data collection. We at Macalester are implementing the MAP system. Upon completion of the MAP project, my algorithm will be integrated with MAP to produce more realistic timetables for Macalester College.

Another area of improvement is concerned with the case-based reasoning system. As of now the case-based reasoning system is serving as a object oriented database, with standard save and retrieve functionalities. In the future I am hoping to improve the case-based reasoning system to include an adaptation system. This change will allow me to change a previous timetable to adapt to new department needs, a task that currently the system is leaving to the administrator. Moreover, I would like to change my index to include other variables that better identify a case.

In addition, in the future, one could improve the system by implementing a more sophisticated selection technique, such as simulated annealing (Boltzmann selection), or other selection techniques of the same caliber. This change might help improve the machine learning process. One could also strengthen the results of this study by comparing the algorithm with an-

other algorithm that implements the standard crossover operator, instead of the self-fertilization operator. Future studies comparing the hybrid genetic algorithm I have developed with other more recent techniques, such as Particle Swarm Optimization might proof insightful.

A possible future study could investigate what the optimal collision cutoff should be, before two classes can be offered in the same time slot. Such an investigation could be helpful to inform administrators' decision during the process of setting enrollment numbers for classes. In addition, the system could also be modified to tell administrators what students have been affected by a collision between two classes. This way departments could advise these students to make different plans. Another change could be to change the system to include hierarchy in the student constraint. Such that, if two classes that have enrolled mostly senior students collide, then they would be more harshly penalized than one that had mostly under class-men. This would be realistic in that freshmen and sophomores still have a chance to re-plan the classes for their academic programs, while seniors have less opportunity to do that, as they are more concerned about graduating. An additional future change would be to label timetables that violate the instructor constraint as infeasible, this way, there would be a greater chance that the best possible timetable would not violate the constraint. In summary, there is an exciting number of directions this project could take,but it is the complex nature of the UCTP that will keep drawing researchers to come-up with the most creative solutions to the problem.

A Extracts of code

A.1 Selection code

```
public List<ScheduleChromosome> rouletteWheelSelection(List<ScheduleChromosome>
                                                        evaluatedPopulation) {

populationwithSelection=new ArrayList<ScheduleChromosome>();
matingPool=new ArrayList<ScheduleChromosome>();
double cumulativefitness=getMaximumExpectedValue(evaluatedPopulation);
int iters=0;

for (ScheduleChromosome chromossome : evaluatedPopulation) {

double selProbability = Math.pow(chromossome.getFitness(),2) / cumulativefitness;
chromossome.setSelectionProbability(selProbability);
populationwithSelection.add(chromossome);
iters++;

}
int repeat=0;

for (int i=0;i<populationwithSelection.size();i++) {
/*
 * Trying to make the way we pick a potential parent
 * to flip the coin on it randomly
 * the code below selects  a random element and checks
 * if the element is fit enough
 * generate random between 0-1 and add selection porbs
 until greater or equal to gen num
 */
Random rand = new Random();
double randNum=rand.nextDouble();
double cumSel=0.0;
ScheduleChromosome tempSelection=null;

for(int k=0;k<populationwithSelection.size();k++){

cumSel+=populationwithSelection.get(k).getSelectionProbability();
if(cumSel>randNum){


tempSelection=populationwithSelection.get(k);
matingPool.add(tempSelection);
break;

}

}

}

return  matingPool;
}
```

A.2 Crossover code

```
public List<ScheduleChromosome> crossover(List<ScheduleChromosome> matingPool,
boolean elitist) {
/*We need to pick the elements from the mating pool with a crossover probability*/
nextGeneration=new ArrayList<ScheduleChromosome>();
List<ScheduleChromosome> bestInds=new ArrayList<ScheduleChromosome> ();
Double crossoverSize=crossoverProbability*matingPool.size();
GAContraintsEngine constraints = new GAContraintsEngine();
ScheduleChromosome prev=null;
newInd=0;
this.sortChromossomes(matingPool);
List<ScheduleChromosome> crossoverList=matingPool.subList(0,crossoverSize.intValue());
if(elitist){
bestInds=choseBestIndividuals(matingPool);

this.sortChromossomes(bestInds);
for(int i=0;i<4;i++){
nextGeneration.add(bestInds.get(i));
}
}

for(ScheduleChromosome chromossome:crossoverList){
ScheduleChromosome parent=null;
ScheduleChromosome offspring1=null;

parent=chromossome;

offspring1=(ScheduleChromosome) parent.deepClone();

Random geneRand=new Random();
Random geneRand2=new Random();
int crossoverPicker=geneRand.nextInt(chromossome.getScheduleGenes().length);
int crossoverPicker2=geneRand2.nextInt(chromossome.getScheduleGenes().length);


Event class1=offspring1.getScheduleGenes()[crossoverPicker][crossoverPicker2];


Random slotPickerRand=new Random();
int slotPicker1=slotPickerRand.nextInt(chromossome.getScheduleGenes().length);
int slotPicker2=slotPickerRand.nextInt(chromossome.getScheduleGenes().length);
if(this.numEvents < (this.numRooms*this.availableTSlots)){
if(this.isSlotFree(offspring1.getScheduleGenes(), slotPicker1, slotPicker2)){

offspring1.getScheduleGenes()[slotPicker1][slotPicker2]=class1;
}else{

this.placeInFreeSlot(offspring1.getScheduleGenes(), class1,
offspring1.getRows(), offspring1.getCols());
}

offspring1.getScheduleGenes()[crossoverPicker][crossoverPicker2]=null;
}
else{
Event class2=offspring1.getScheduleGenes()[slotPicker1][slotPicker2];
```

```
offspring1.getScheduleGenes()[slotPicker1][slotPicker2]=class1;
offspring1.getScheduleGenes()[crossoverPicker][crossoverPicker2]=class2;

}
offspring1.setFitness(constraints.FindParentFitness(offspring1));

if(elitist){
if(offspring1.getFitness()>parent.getFitness()){
nextGeneration.add(offspring1);
if(prev!=offspring1){
newInd+=1;
}
numChildren+=1;
prev=offspring1;
}else{
nextGeneration.add(parent);
if(prev!=parent){
newInd+=1;
}
numParent+=1;
prev=parent;
}
}else{
nextGeneration.add(offspring1);
if(prev!=offspring1){
newInd+=1;
}
numChildren+=1;
prev=offspring1;
}

}
return nextGeneration;

}
```

A.3 Constraints Implementation Code

```
public double studentConstraint(ScheduleChromosome parent){
double studentPoints=2.5;
double dup=0.0;
double jaccard=0.0;
if(parent !=null && parent.getScheduleGenes().length>=1){
for (int i = 0; i < parent.getRows(); i++) {
for(int j=0;j<parent.getCols()-1;j++){
//check to see if the same class is been offered at the same time
try{
List<Integer> students1=parent.getScheduleGenes()[i][j].getStudents();
List<Integer> students2=parent.getScheduleGenes()[i][j+1].getStudents();
for(int studentID1: students1){
for(int studentID2:students2){
if(studentID1==studentID2){
dup+=1.0;
}
```

```
}
}
jaccard=Math.max(dup/students1.size(), dup/students2.size());
studentPoints-=jaccard;
dup=0;

}catch(Exception e){
System.out.println("ERROR IN CONSTRAINT");

}

}
}


}
else{
System.out.println("Message: Not enough Genes to evaluate.
Gene count=0 or 1 or the current parent is null");

}

return studentPoints;
}


public double instructorConstraint(ScheduleChromosome parent){
double instructorPoints=2.5;
double dup=0.0;
double jaccard=0.0;
int parentLength=parent.getScheduleGenes().length;
if(parent !=null && parentLength>=1){
for (int i = 0; i < parent.getRows(); i++) {
for(int j=i;j<parent.getCols()-1;j++){
//check to see if the same class is been offered at the same time
try{

if(parent.getScheduleGenes()[i][j].getInstructor()==
parent.getScheduleGenes()[i][j+1].getInstructor()){
dup+=1;
}


}catch(Exception e){
System.out.println("ERROR IN CONSTRAINT");

}

}
}
jaccard=Math.max(dup/parentLength, dup/parentLength);
instructorPoints-=jaccard;

}

return instructorPoints;
}
```

```
public double FindParentFitness(ScheduleChromosome parent){

double totalfitness=0;
double studentPenalty=studentConstraint(parent);
stuConstraints.add(studentPenalty);
double instructorPenalty=instructorConstraint(parent);
instructorConstraints.add(instructorPenalty);

totalfitness=studentPenalty+instructorPenalty;//+classPenalty;
if(totalfitness<=0){
totalfitness=0.00000000001;
}
return totalfitness;
}
```

A.4 GA Routine Code

```
public void ga(List<ScheduleChromosome> nextGenPop,GAUtil gaUtil,int numSteps,
int iter,boolean cbrTraining) throws Exception{

numNewIndvs.add(gaUtil.getNewInd());
nextGenPop=gaUtil.crossover(gaUtil.rouletteWheelSelection(nextGenPop),true);
ScheduleChromosome best=gaUtil.getBestChromossome(nextGenPop);
fitnessAvgGen.add(gaUtil.calculateAvgGenFitness(nextGenPop));
bestFitness.add(best.getFitness());
if(best.getFitness()>=4.5 && numSteps==bparser.getNUM_GEN()){
if(cbrTraining){
cbr.saveCase(best, this.numSlots, this.numRooms, this.numInstructors);
}
else{
printer.printBestChromossome(best);
data=grapher.buildFitBarGraph(nextGenPop);
System.out.println("Child nums: "+gaUtil.getNumChildren());
System.out.println("Parent nums: "+gaUtil.getNumParent());
}
return;

}else if (best.getFitness()<=3.50 && numSteps==2*bparser.getNUM_GEN()  ){
System.out.println(" WARNING: The algorithm did not converge,
   please run again for better results");
if(cbrTraining){
cbr.saveCase(best, this.numSlots, this.numRooms, this.numInstructors);
}
else{
printer.printBestChromossome(best);
data=grapher.buildFitBarGraph(nextGenPop);
System.out.println("Child nums: "+gaUtil.getNumChildren());
System.out.println("Parent nums: "+gaUtil.getNumParent());
}

return;
```

```
}else if(numSteps==4*bparser.getNUM_GEN()){
System.out.println(" WARNING: It took longer than the number of steps allowed");
if(cbrTraining){
cbr.saveCase(best, this.numSlots, this.numRooms, this.numInstructors);
}
else{
printer.printBestChromossome(best);
data=grapher.buildFitBarGraph(nextGenPop);
System.out.println("Child nums: "+gaUtil.getNumChildren());
System.out.println("Parent nums: "+gaUtil.getNumParent());
}
return;
}


numSteps++;
genCounter++;
iter--;
this.ga(nextGenPop, gaUtil,numSteps,iter,cbrTraining);


}
```

B XML files

B.1 GA XML file

```
<?xml version="1.0" encoding="UTF-8"?><!--This is a setting to test the algorithm-->
<parameters>
<parameter id="1">
<parameterName>POP_SIZE</parameterName>
<value>100</value>
</parameter>
<parameter id="2">
<parameterName>NUM_GENERATIONS</parameterName>
<value>100</value>
</parameter>
<parameter id="3">
<parameterName>NUM_TIMESLOTS</parameterName>
<value>6</value>
</parameter>
<parameter id="4">
<parameterName>NUM_STUDS</parameterName>
<value>200</value>
</parameter>
<parameter id="5">
<parameterName>NUM_ROOMS</parameterName>
<value>6</value>
</parameter>
</parameters>
```

B.2 Events XML file

```
<?xml version="1.0" encoding="UTF-8"?><!--This is a setting to test the algorithm-->
<events>
   <event id="1">
      <courseID>1</courseID>
      <courseName>Comp 120</courseName>
      <instructorID>1</instructorID>
      <instructorName>Elizabeth Shoop</instructorName>
  <classSize>20</classSize>
  <frequency>3</frequency>

   </event>
   <event id="2">
      <courseID>2</courseID>
      <courseName>Comp 121</courseName>
      <instructorID>5</instructorID>
      <instructorName>Eric Theriault</instructorName>
  <classSize>36</classSize>
  <frequency>3</frequency>

   </event>
   <event id="3">
      <courseID>3</courseID>
      <courseName>Comp 123</courseName>
      <instructorID>2</instructorID>
      <instructorName>Susan Fox</instructorName>
  <classSize>25</classSize>
  <frequency>3</frequency>

   </event>
   <event id="4">
      <courseID>4</courseID>
      <courseName>Comp 221</courseName>
      <instructorID>3</instructorID>
      <instructorName>Shilad Sen</instructorName>
  <classSize>12</classSize>
  <frequency>3</frequency>

   </event>
</events>
```

References

Abramson, D. (1991). Constructing school timetables using simulated annealing: sequential and parallel algorithms. *Management Science 37*(1), 98–113.

Abramson, D. and J. Abela (1992). A parallel genetic algorithm for solving the school timetabling problem. In *Proceedings of the Fifteenth Australian Computer Science Conference (ACSC-15)*, Volume 14, pp. 1–11.

Burke, E., B. MacCarthy, S. Petrovic, and R. Qu (2000). Structured cases in case-based reasoning re-using and adapting cases for time-tabling problems. *Knowledge-Based Systems*, 159–165.

Burke, E., B. MacCarthy, S. Petrovic, and R. Qu (2001). Case-based reasoning in course timetabling: an attribute graph approach. *Lecture notes in computer science*, 90–104.

Burke, E. K., D. G. Elliman, and R. Weare (1994). A genetic algorithm based university timetabling system. In *East-West Conference on Computer Technologies in Education, Crimea, Ukraine pp35-40*.

Burke, E. K., D. G. Elliman, and R. F. Weare (1995). A hybrid genetic algorithm for highly constrained timetabling problems. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pp. 605–610.

Burke, E. K., J. P. Newall, and R. F. Weare (1996). A memetic algorithm for university exam timetabling. *Lecture notes in computer science 1153*, 241–250.

Burke, E. K. and S. Petrovic (2002). Recent research directions in automated timetabling. *European Journal of Operational Research 140*(2), 266–280.

Costa, D. (1994). A tabu search algorithm for computing an operational timetable. *European Journal of Operational Research 76*(1), 98–110.

Cotta, C. and A. Fernandez (2007). Memetic Algorithms in Planning, Scheduling, and Timetabling. *Computational Intelligence (SCI) 49*, 1–30.

Dalton, J. (2010, April). Roulette wheel selection. `http://www.edc.ncl.ac.uk/highlight/rhjanuary2007g02.php/`.

Dorigo, M. and K. Socha. An introduction to ant colony optimization. *Handbook of Approximation Algorithms and Metaheuristics*, 26–1.

Dowsland, K. and J. Thompson (2005). Ant colony optimization for the examination scheduling problem. *The Journal of the Operational Research Society 56*(4), 426–438.

Fen, H., S. Deris, M. Hashim, and S. Zaiton (2009). University course timetable planning using hybrid particle swarm optimization. *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, 239–246.

Fleischer, M. (1995). Simulated annealing: past, present, and future. In *WSC '95: Proceedings of the 27th conference on Winter simulation*, Washington, DC, USA, pp. 155–161. IEEE Computer Society.

Gaspero, L. D. and A. Schaerf (2001). Tabu search techniques for examination timetabling. *Lecture notes in computer science*, 104117.

Goltz, H. J. and D. Matzke (1999). University timetabling using constraint logic programming. *Lecture notes in computer science*, 320–334.

Goodall, D. (1966). A new similarity index based on probability. *Biometrics 22*(4), 882–907.

Hertz, A., E. Taillard, and D. De Werra (1992). A tutorial on tabu search. In *Proc. of Giornate di Lavoro AIRO*, Volume 95. Citeseer.

Kendall, G. and N. Hussin (2005). A tabu search hyper-heuristic approach to the examination timetabling problem at the MARA university of technology. *Lecture notes in computer science 3616*, 270.

Lewis, R. and B. Paechter (2005). Application of the grouping genetic algorithm to university course timetabling. *Lecture notes in computer science 3448*, 144–153.

Lewis, R. and B. Paechter (2007). Finding feasible timetables using group-based operators. *IEEE Transactions on Evolutionary Computation 11*(3), 397–413.

Liu, Y., D. Zhang, and S. Leung (2009). A simulated annealing algorithm

with a new neighborhood structure for the timetabling problem. *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, 381–386.

Lucas, D. C. (2000). Algoritmos genéticos: um estudo de seus conceitos fundamentais e aplicaç\ ao no problema de grade horária. *Monografia de Graduaç\ ao. Pelotas*.

McCollum, B., P. McMullan, E. K. Burke, A. J. Parkes, and R. Qu (2008). A new model for automated examination timetabling. *An extended version of McCollum et al (2007). Under Review*.

Myszkowski, P. and M. Norberciak (2003). Evolutionary Algorithms for Timetable Problems. *Annales UMCS, Sectio Informatica I*, 115–125.

Perzina, R. (2006). Solving the university timetabling problem with optimized enrollment of students by a parallel self-adaptive genetic algorithm. In *Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling*. Citeseer.

Petrovic, S. and E. K. Burke (2004). University timetabling. *Handbook of scheduling: algorithms, models, and performance analysis*.

Qu, R. (2002). *Case based reasoning for course timetabling problems*. Citeseer.

Qu, R., E. K. Burke, B. McCollum, L. T. Merlot, and S. Y. Lee (2006). A survey of search methodologies and automated approaches for examination timetabling. *Computer Science Technical Report No. NOTTCS-TR-2006-4*.

Raghavjee, R. and N. Pillay (2008). An application of genetic algorithms to the school timetabling problem. In *SAICSIT '08: Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries*, New York, NY, USA, pp. 193–199. ACM.

Rich, D. C. (1996). A smart genetic algorithm for university timetabling. In *Practice and Theory of Automated Timetabling: First International Conference, Edinburgh, UK, August 29-Septmber 1, 1995: Selected Papers*,

pp. 181.

Ross, P., E. Hart, and D. Corne (1998). Some observations about GA-based exam timetabling. *Lecture notes in computer science*, 115–129.

Rossi-Doria, O. and B. Paechter (2004). A memetic algorithm for university course timetabling. *Combinatorial Optimisation*.

Russell, S. and P. Norvig (2003). *Artificial Intelligence : A Modern Approach*. Upper Saddle River, NJ : Prentice Hall.

Schaerf, A. (1999). A survey of automated timetabling. *Artificial Intelligence Review 13*(2), 87–127.

Sipser, M. (1996). *Introduction to the Theory of Computation*. International Thomson Publishing.

Socha, K., M. Sampels, and M. Manfrin (2003). Ant algorithms for the university course timetabling problem with regard to the state-of-the-art. *Lecture Notes in Computer science*, 334–345.

Wang, Y. Z. (2003). Using genetic algorithm methods to solve course scheduling problems. *Expert Systems with Applications 25*(1), 39–50.

Wang, Z., J. Liu, and X. Yu (2009). Self-fertilization based genetic algorithm for university timetabling problem. *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, 1001–1004.

Watson, I. and F. Marir (1994). Case-based reasoning: A review. *Knowledge Engineering Review 9*(4), 327–354.

Wilke, P., M. Grobner, and N. Oster (2002). A hybrid genetic algorithm for school timetabling. *Lecture notes in computer science*, 455–464.