Heterogeneous student grouping by a genetic algorithm

Rajan Kapila
Athasbasca University
rajankapila989@hotmail.com

Paul Maguire
Athabasca University
paul@pmetal.ca

Abstract: Students within a cohort are described by a list of scores for various personality attributes. Our project team elected to use a genetic algorithm to discover good student groupings. A good grouping assembles four students with two outliers surrounding a coherent core. A "Goodness of Heterogeneity" score for each candidate solution was maximized by a genetic algorithm using various different combinations of parameter values. By varying the control parameters, we were able to determine which parameters contribute to algorithmic performance and discover several satisfactory groupings.

Introduction

The problem addressed by this paper is the implementation of a Computational Intelligence (CI) algorithm of our choice to search for good solutions to a student grouping problem. At first glance, the reader might suspect the student grouping problem specification describes a clustering problem. The success of artificial neural nets with categorization problems [1] has early appeal. However, this grouping is a bit of a red herring. The goodness of heterogeneity alters the perceived problem by requiring differences between group members. The student grouping is revealed as an optimization problem.

The objective function is clearly articulated. A student cohort must be divided into groups, subject to several constraints. Each student belongs to exactly one group; each group has exactly four students; and at least one student pair must be separated by a Euclidean distance (in attribute scores) of at least 2. In addition, each group must have a goodness of heterogeneity (GH)[1] score greater than 0.5. The objective function maximizes the sum of the group GH scores. Anecdotal evidence suggests that, for this problem, "very good solutions have a heterogeneity of over 600" [2, p. 3].

Optimization problems are well within the realm of suitable application areas for genetic algorithms (GAs). However, in this, GAs do not stand out. Evolutionary algorithms in general can be relied upon to discover solutions for optimization problems [3].

So how do we differentiate between evolutionary algorithms in choosing the mechanism to use for this project? GAs have an elegant simplicity that appears easy to program. The model is intuitive to anybody with an elementary awareness of biological reproduction, perhaps more so than neural nets or particle swarms. And both team members are personally interest in GAs.

The genetic algorithm

---

[1] This is a measure of variation of the students' attribute scores. For more information, please see the assignment: "COMP658 F13: Assignment: Assignment 2 – Programming Exercise 1." http://scis.lms.athabascau.ca/mod/assignment/view.php?id=16823.

Evolutionary computing has developed over several years. Holland [4] is generally credited with introducing the genetic algorithm. However, Potvin [5] discusses Fogel's earlier work[2] as a precursor to modern GA approaches. Of particular interest is Fogel's research on asexual reproduction in algorithms similar to GA. This was instructive for our development as will be shown later.

Today's standard genetic algorithm is an effective tool for optimization and search [6]. Although Holland's seminal GA work predates Adams's fiction [7], they share a common root: recognition that the planet itself is the greatest computational device known to humankind. As evolution has produced incredibly complex and successful organisms, it is a viable model for an automated problem-solving system.

GAs operate by mimicking evolution's natural selection in several ways (discussion to follow):

> *Survival of the fittest*. In the real world, a population's more successful organisms have a better chance of survival and, therefore, propagation – in either case, representation in the next generation. Similarly, a GA has a population of candidate solutions to the problem it is trying to solve. Better solutions tend to live longer during the run.
>
> *Reproductive process*. In nature, parents contribute genetic material to their offspring. A GA's candidate solutions are also the building material from which a new generation of candidate solutions is constructed.
>
> *Random influences, perhaps external*. In nature, mutation occasionally introduces changes to a genetic signature [8]. GA practitioners also typically apply a mutation to "infant" offspring.

The GA candidate solution is encoded into a form that the algorithm can process. In GA parlance, the encoded candidate solution is called a chromosome. As in biology, a chromosome is a combination of genes. In a GA, a gene is somewhat analogous to one of a mathematical function's terms. In the GA's search for solutions, various combinations of genes are tested. Superior evolutionary biological outcomes develop in a similar way.

A GA's key characteristics operate together. We introduce the elements here to clarify later discussion:

> *Encoding*. The encoding is a translation of a candidate solution into a form the algorithm can work with. The most common encoding, and easiest to work with, is the bit string.
>
> *Genetic operators*. Nature's genetic crossover is shared contribution of parental genetic material to the offspring. Crossover and mutation functions are mirrored in GAs.
>
> *Selection mechanism*. Not all chromosomes are suitable for reproduction. Various approaches exist to nominate superior candidate "parents".
>
> *Fitness score*. Each chromosome is evaluated by a fitness function that describes the quality of the encoded candidate solution.

Several key control parameters influence GA performance in affecting how quickly a good solution is found.

```
Generate random initial population.
Evaluate population.
While termination criteria not achieved:
        Select parents.
        Crossover.
        Mutate.
        Evaluate.
```

Figure 1 - basic genetic algorithm

[2] We have not read the referenced text: Fogel, Lawrence J., Alvin J. Owens, and Michael J. Walsh. *Artificial intelligence through simulated evolution*. Chichester: John Wiley & Sons, 1967.

Population size. Several chromosomes exist in any given generation. The size of the population affects the GA performance.

Probability of crossover (P_c). The genetic sequence of a child chromosome generally represents a combination (crossover) of its parental chromosomes. The crossover occurrence is stochastically determined, with P_c acting as a key control.

Probility of mutation (P_m). Mutation is also stochastically determined, applied with P_m.

With the key players introduced, we can now outline general GA operation (*see figure 1*).

The GA implementation starts with determining the encoding for candidate solutions. Although real-valued and tree organizations have been investigated, bit string representations are simpler to program, and match up better conceptually with the biological analogue.

Various stochastic approaches to parent selection exist. In each case, some probability for selection is allocated to each chromosome in the population. The Roulette system allocates a probability of selection to each chromosome determined by the proportion of the population's total fitness (sum of all chromosomes' fitness scores) that was contributed by that chromosome. Tournament systems randomly select population subgroups and nominate the chromosome with the highest fitness score in a group as the winner.

Crossover happens with probability P_c. The crossover operator (typically) takes two chromosomes as parents. One or more random crossover points are selected. The crossover points are used to splice out pieces of each chromosome. The splices are exchanged, creating two new chromosome offspring (see figure 2).

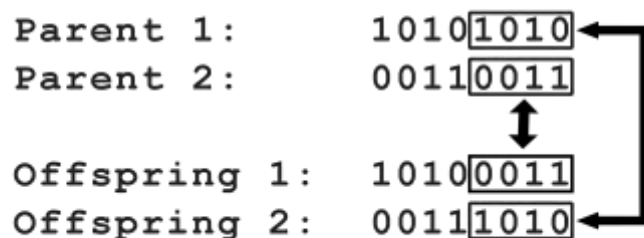

Figure 2 - Crossover at position 4

A mutation operator is applied with probability P_m. With a bit string encoding, the mutation operator will reverse a randomly chosen bit.

New chromosomes are generated in this way until a new population is ready for evaluation.

The parametric values for P_c, P_m, etc., can greatly influence the algorithm's performance. Researchers have looked for optimal settings and have concluded that none exist [9]. Instead, to achieve best results, practitioners are forced to use heuristic settings, perhaps based on problem type, and manually tweak the parameters to see what works well.

The chief remaining question is how or when to stop the GA. As it typically seeks an optimal solution that is not already known or verifiable, other termination criteria are needed. One possibility is termination after a specified number of iterations. Another is when the fitness score exceeds some threshold. Alternatively, the algorithm can be halted when the chromosomes' fitness scores do not show significant change over several consecutive generations.

Unchanging fitness scores define the state where chromosomes have converged. While the goal of the GA is to converge a population of chromosomes to some optimal value, premature convergence suggests that the GA got stuck at a local optimum. The mutation operator is intended to help break out of such a situation. The opposite problem is slow finishing, which happens when convergence to a solution is too slow in coming. Careful manipulation of the control parameters seeks to find balance between these two problems.

Our approach

Several decisions went into defining the student grouping problem in terms that a genetic algorithm could address efficiently.

Encoding

Determining an effective encoding "remains an art" [10]. We realized early how heavily the genetic operators depend on the encoding mechanism.

A candidate solution to the student grouping problem is an ordering of all students into groups. Each chromosome must represent a candidate solution. Therefore, a single chromosome comprises an ordering of all students. In our configuration, each gene is a student. Student groupings are implicitly defined within the ordering of the chromosome. In other words, the first four genes in the chromosome make up the first group, and so on.

In keeping with the predominant practice, we first sought to construct our chromosome as a bit string. The data provided includes student numbers and an array of attribute scores for each student. As the attribute scores do not change, we chose to construct the chromosome including only student numbers. This brought to light new challenges. The standard bitwise crossover and mutation operations would function poorly, for they would alter student numbers rather than relocate genes in the chromosome.

Ultimately, we chose to treat the genes as atomic and define the genetic operators accordingly.

Crossover

A typical single-point crossover operation operates as shown in figure 2. Alternative approaches with varying numbers of crossover points have been researched [11]. Two-point crossover avoids positional and distributional bias [6] by splicing pieces from the middle of each parent into the other parent.

We began by examining whether the typical crossover model would work for our algorithm.

It is easy to demonstrate that traditional bitwise crossover will not work when the genes are student numbers. Consider this 8-bit example with four student numbers from 0 to 3 (00 to 11 binary). From these two random parent chromosomes:

Parent 1: 00 01 1**0 11**
Parent 2: 10 01 1**1 00**

if we select bit 5 as our single crossover point, we get these offspring:

Offspring 1: 00 01 1**1 00**
Offspring 2: 10 01 1**0 11**

Clearly, all four student numbers are no longer represented in either offspring chromosome. In the first offspring chromosome, student 00 is repeated and student 10 is absent. In the second offspring chromosome, student 10 is repeated and student 00 is absent. In fact, with this example, no single crossover point yields valid offspring[3].

---

[3] Here, we are defining a valid chromosome as one that has all student numbers encoded.

While we could simply discount such chromosomes as invalid, awarding very low fitness scores, it is probable a high rate of invalid chromosomes would negatively affect algorithmic performance.

We also investigated single-parent models, reasoning that, with the full range of numbers in [000000000 – 111111111] (the provided student numbers range consecutively from 1 to 512) provided in the student grouping problem, it would be impossible for traditional crossover to generate invalid student numbers. Asexual reproduction is nature's analogue. Under that biological model, the child is the product of only one parent – in essence, the child is a clone of the parent. This is in line with Fogel's work referred to above. However, single parent models eliminate crossover altogether, and leave mutation as the only avenue for genetic diversity [12]. Single-bit mutation would be guaranteed to fail as the total number of 0s and 1s in the bitstring would get out of balance. (A bit string of all 1s is a clear example of an invalid chromosome.)

Having rediscovered known issues with traditional crossover operation for a purely ordered problem [5], we chose to investigate research on Traveling Salesman Problem (TSP) implementations using GA solutions. A review of some early work classifies crossover operations as preserving position and preserving order, before going on to discuss "adjacency representation", which we consider edge-based operations [5]. The TSP analogy with our current problem breaks down when considering relationships between genes. Where edges (with costs) connect TSP genes, no such explicit relationship exists in the student grouping problem. Although edge-based operations outperform the other approaches, they are unavailable to this problem. Instead, we target Order Crossover, which significantly outperforms the position preservation crossover approaches [5].

Order Crossover (OX) creates a single offspring from two parents. Two crossover points (called "cut" points in OX) are chosen at random. The substring defined in the first parent by the cut points is copied to the same position in the offspring. Then, starting after the second cut point and wrapping around to the start, the offspring is populated with the corresponding gene from the second parent. Duplicates (i.e., genes that were already copied from the first parent) are not repeated – instead, those genes from the second parent are skipped. The offspring ends up with a complete set of genes, with no duplicates [5] (see figure 3). As illustrated, once the two genes are copied from parent 1 to the offspring, parent 2 fills the remaining spaces starting after the second cut point. The gene sequence used by parent 2 to complete the offspring is 362415, starting after the second cut point and wrapping around to the start. When the gene valued 4 is reached, it is skipped because it is already in the offspring.

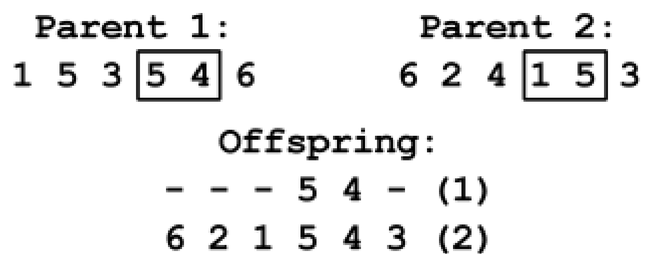

Figure 3 - Order crossover (OX)

Mutation

Standard bitwise mutation will not work for our chromosome for reasons explored above. Accordingly, we devised our own simple mutation operator. As determined by P_m, we randomly select and exchange two genes. We did not research this, but feel confident others have explored this implementation.

Selection mechanism

We explored different options for selection. Our initial implementation featured random selection of parents (and showed poor convergence). We also implemented Roulette and Tournament selection, and enabled the choice of selection algorithm a parameter of our program. Tournament size is also a run time parameter for our program. (The operation of these selection algorithms is discussed above.)

Elitism is a mechanism that helps speed convergence by promoting (copying, not moving – they are still available for reproduction) best chromosomes directly into the next generation before the reproductive cycle. We

implement elitism, but with a variation. Our preference is for valid chromosomes, above even invalid chromosomes with higher fitness scores.

Fitness function

The fitness function is implemented as given in the assignment, with a small, optional adaptation. As penalty function can adjust a fitness score so as to promote valid chromosomes [13], we allowed for the imposition of a penalty which would devalue invalid chromosomes. The decision whether or not to impose the penalty, and the size of the penalty, are both program parameters. We also inherently penalize invalid groups by excluding their GH score from their chromosome's aggregate GH score.

Results

To determine the best parameters to utilize in the GA, we ran some preliminary tests to determine a parent selection method, crossover rate, mutation rate, and a population size. The charts below show our findings. Data was averaged out based on 10 trial runs.

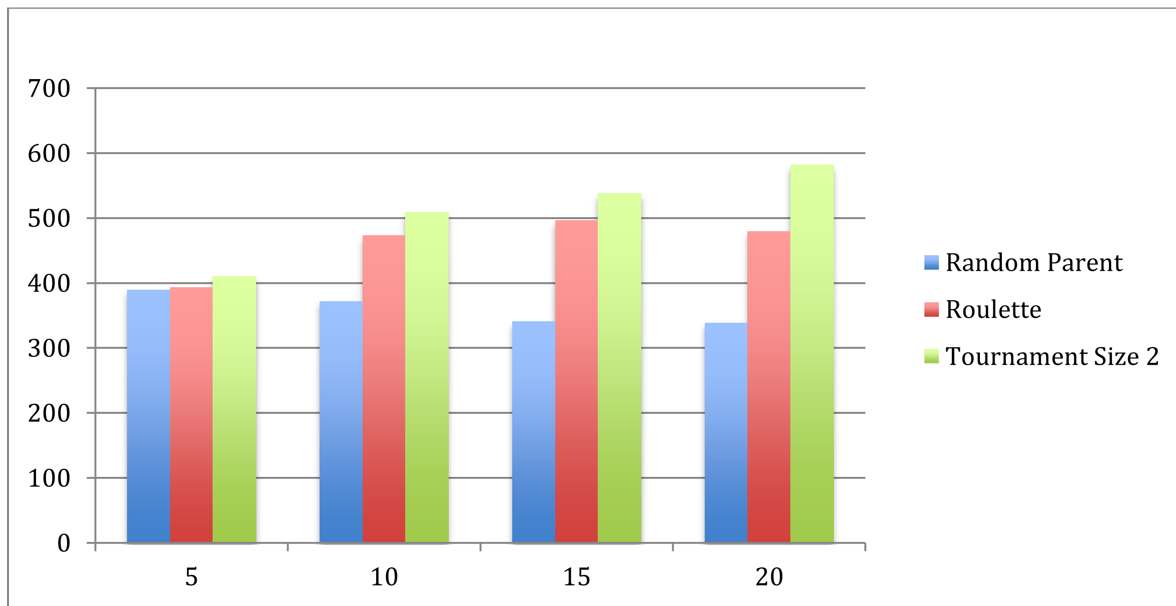

Figure 4 - Ten trial average of max GH vs. Population size for each parent selection mode over ten thousand generations.

The results in figure 4 show that tournament based parent selection outperforms the roulette and random parent selection method quite handily at the each population size but most noticeably at the twenty chromosome population size. These results allowed us to focus on tournament style selection. The result also showed that a population size of twenty appears to give the largest GH at the end of ten thousand generations, which allowed us to focus on the twenty chromosomes. Due to time constraints and computational power, we did not explore beyond twenty chromosomes.

Although the numbers are close, based on figure 5, we decided to use a tournament size of two for our parent selection algorithm in our quest for largest GH.

Tests showed that parental crossover rates did not dramatically affect max GH outcomes so we settled on using 0.7. Figure 6 shows that mutation rate does affect max GH outcomes. A 0.3 mutation rate yielded the highest GH in ten thousand generations.

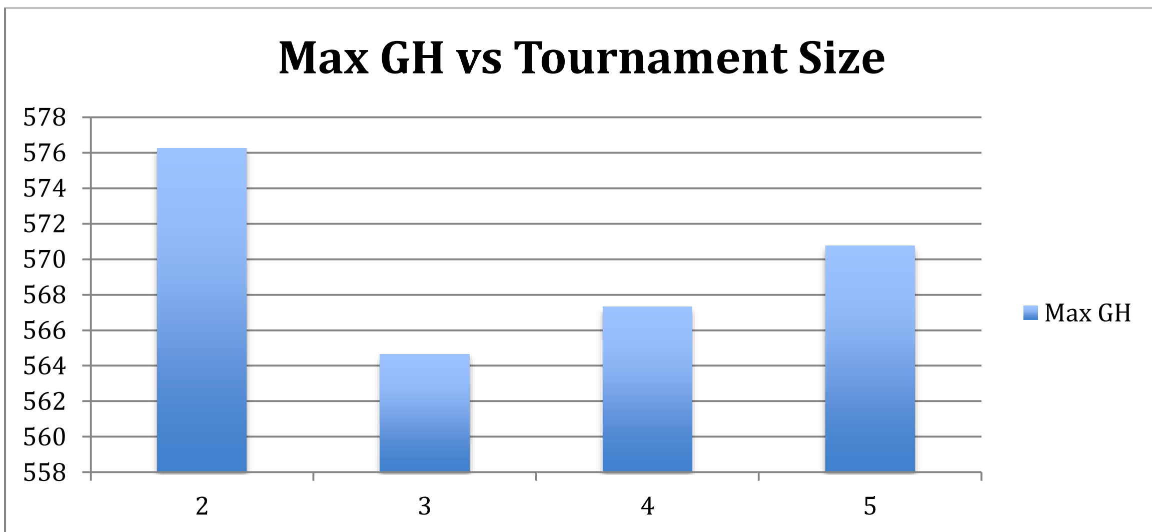

Figure 5 - Max GH vs. tournament size achieved after ten thousand generations.

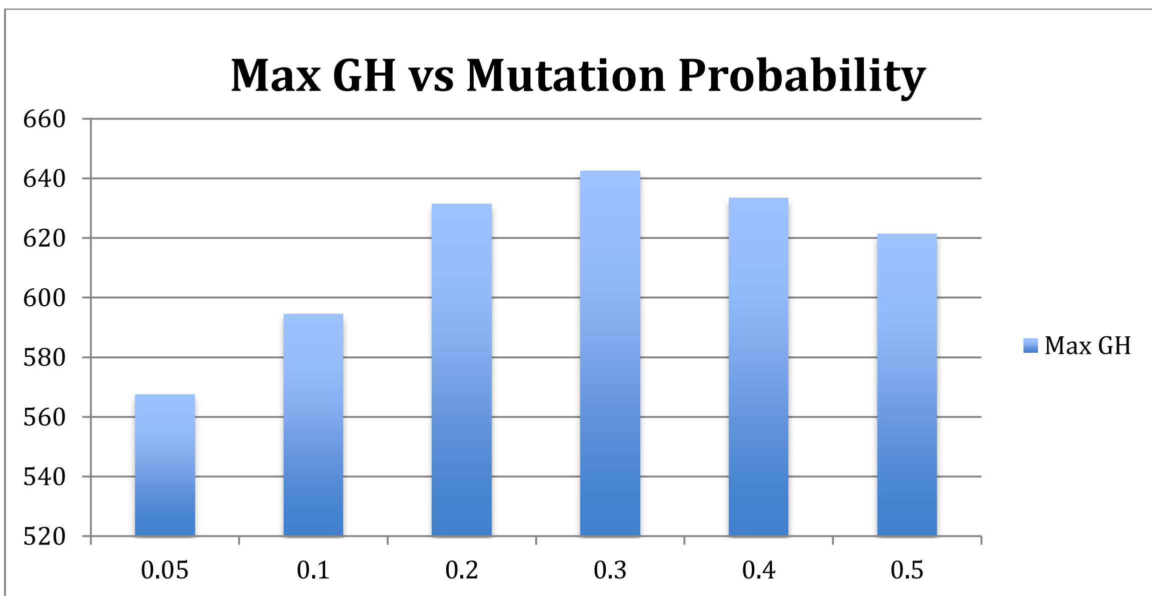

Figure 6 - Max GH vs. mutation rate achieved after ten thousand generations

Along with the penalty of not counting GH values for groups that are invalid in the total sum of GH for the chromosome, we also penalize the chromosome for being invalid overall. This penalty appears to give higher chromosome GH values and is included in the algorithm.

Along with tournament style parent selection, we also promote elite chromosomes to the next generation by first promoting valid chromosomes with high scores and if we could not find two valid chromosomes, invalid chromosomes with the highest scores were promoted.

Our first testing model selected random parents. Our expectation was that no convergence would be observed, and that GH scores would hover around an average value for a random group of chromosomes. This was tested and observed, with $P_c = 0.65$ and $P_m = 0.05$; using population sizes of 10 and 100; and running generations of 100 and then 1000. All showed similar results, with average GH ranging from 196.164413 to 207.984909.

After testing, we selected our best performing parameters to find a maximum GH:

- Population Size = 20
- Parent Selection Mode = Tournament Style with Size 2
- Crossover Probability = 0.7
- Mutation Probability = 0.3
- Penalty for Invalid Chromosomes = ON
- Generation Size = 100000

The maximum GH we achieved was 652.25 in between generation 20000 and 21000.

How to run our program

Our program has two modes of operation. At present, the choice between DIRECT and EXPLORE must be made at compile time. DIRECT mode reads from a file provided through a command line argument when running the program:

```
java –jar comp658-rp5 runs.txt
```

In the `runs.txt` file, each line is a set of parameter and value pairs separated with " : ", and pairs are comma separated. For example, one line might be:

```
fileName : input.txt, numberOfElites : 2, populationSize : 10, maxGenerations : 10,
dataPointFrequency : 1, numberOfParents : 2, numberOfCrossoverPoints : 2,
probabilityCrossover : 0.8, probabilityMutate : 0.05, paTournamentSize : 5, parentAlgo : 2,
cpMode : 0
```

The `runs.txt` file can be several lines long, resulting one program run for each line. This shows the level of parameterization of our program.

Despite its presence in the `runs` file, the input data file is not parameterized. An input file named `input.txt` must be present in the same directory as the program jar file. The input data file expects a file where each line contains a student number and seven attribute scores, all comma separated.

The EXPLORE mode, which can be activated by a programming change at `main.java` line 35, selects run time parameters through a `ParmSet` class. The same parameters can be set up there and the program will iterate through them.

As provided, the program will run 10 trials with the same parameter values while operating in DIRECT mode. Only one trial per set of parameters will occur in EXPLORE mode.

Conclusions

In conclusion, the genetic algorithm was very well suited for this exercise problem. The parameters of population size and mutation along with parent selection method were the biggest factors in reaching optimums. As expected, the ability to create variation amongst chromosomes in each generation appears to be the key factor in pushing the algorithm to find optimums. GA does have a tendency to create long periods of generations with no valid chromosomes. But the ability to create random variation allows the algorithm does find its way back to valid chromosomes. The variation also mitigates against premature convergence of chromosomes. Although we did optimize our parameters to create max GH values in reasonable time, we did notice that the max GH for a particular run was found at random generations. Another issue we ran into is that invalid chromosomes with higher GHs than the highest valid chromosome GH could take over a generation. The GA would usually find

valid chromosomes again eventually, but this invalid chromosome pattern could persist for many generations. For this reason, we found it important to run searches for global optima over 100,000 generations, even though highest GHs were sometimes found quite early in the process. The tricky part of using GA to solve a problem is that it doesn't give the one correct answer, but a very good one - hence we cannot be sure our solution is the best, but we do believe that the max GH our GA discovered is very high. We believe with more simulations and more refined tweaks to the parameters, a higher max GH could possibly be found.

Reflections

Raj: Overall the project went very smoothly. Watching a biological theory solve or maximize solutions to real world problem is quite wonderful. The difficult part of the assignment is to know when to quit running trials with different parameter permutations. Time constraints limit what you can run but there was an almost sick desire to keep running simulations to see if we could push the max GH even higher.

Paul: I looked forward to a reason to program a genetic algorithm, and I am not disappointed. I enjoyed the assignment. Despite familiarity with GA operation gained in this course, I was found the effects of parameter change interesting to watch. Our project team worked well together and created an equitable division of labour – and I got to work on the parts I wanted to! I suppose the biggest challenge was finding the time to start and learning to work with Git. (I'm a Subversion user.) However, an online repository and code management tool is critical to the success of a project like this.

References

[1] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: a tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, 1996.

[2] "COMP658 F13: Assignment 2 Questions." [Online]. Available: http://scis.lms.athabascau.ca/mod/forum/discuss.php?d=38038&parent=112134. [Accessed: 11-Nov-2013].

[3] Y. Shi, "Particle swarm optimization: developments, applications and resources," in *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, 2001, vol. 1, pp. 81–86.

[4] J. H. Holland, *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[5] J.-Y. Potvin, "Genetic algorithms for the traveling salesman problem," *Ann. Oper. Res.*, vol. 63, no. 3, pp. 337–370, 1996.

[6] M. Srinivas and L. M. Patnaik, "Genetic algorithms: a survey," *Computer*, vol. 27, no. 6, pp. 17–26, 1994.

[7] D. Adams, *Life, the universe and everything*, vol. 3. Tor UK, 1984.

[8] "How do Mutations Occur?" [Online]. Available: http://learn.genetics.utah.edu/archive/sloozeworm/. [Accessed: 10-Nov-2013].

[9] K. A. DeJong, "Analysis of Behavior of a Class of Genetic Adaptive Systems." PhD thesis, University of Michigan, Ann Arbor, MI, 1975.

[10] S. Chatterjee, C. Carrera, and L. A. Lynch, "Genetic algorithms and traveling salesman problems," *Eur. J. Oper. Res.*, vol. 93, no. 3, pp. 490–510, Sep. 1996.

[11] M. Srinivas and L. M. Patnaik, “Adaptive probabilities of crossover and mutation in genetic algorithms,” *IEEE Trans. Syst. Man Cybern.*, vol. 24, no. 4, pp. 656–667, 1994.

[12] D. B. Fogel, “What is evolutionary computation?,” *Spectr. IEEE*, vol. 37, no. 2, pp. 26–28, 2000.

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II,” *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002.