# Efficiency of Data Structures in an SQL Genetic Algorithm

**Chris Dilger (101133703)**
COS10012 – Object Oriented Programming

## Abstract
A system for mining large flat datasets for useful data in relation to a natural language research question is examined in the context of computational efficiency. Using a GA to test individuals with a mocked fitness function for testing many iterations of both a Tree based individual and a Flat individual implementation is compared. Tree based individuals were found to take less time to crossover than the Flat individuals. This unexpected difference was attributed to the fact that the list was dynamically allocated which was supported by further investigation. Further research can be done to determine the number of generations after which each implementation takes to reach a fitness plateau.

## 1. Introduction
A need for tools to reduce the size and scope of flat datasets has been identified. In several disciplines large flat data tables containing large amounts of useful data, useless metadata and often many sparse or unpopulated columns need to be evaluated by a human in the context of research in that discipline. An example research question might be, "How many discoveries has Martin made since 1990?" which a human will interpret, create a relevant SQL query to return the data allowing the user to answer this research question. This human intuition step is a viable candidate for machine learning, as there is a large amount of data that must be processed manually by a human in order to generate the relevant query.

One such approach would use a genetic algorithm (GA), which is a population-based global search technique based on Darwinian evolutionary theory (Holland 1975; Goldberg 2006). A GA has three elementary steps, namely selection, crossover and mutation, each outlined with pseudocode in Figure 1. A genetic algorithm works by defining a population of individuals, each of which are a potential solution to the specified problem. Each generation of the population goes through a the steps of keeping the best individuals as evaluated by some fitness function (selection), generating new individuals using the traits of fit individuals (crossover) and introduction of some random variations to the individuals (mutation). Provided that a fitness landscape can be defined, a GA is suited to finding optimal solutions in that fitness landscape.

SGA ($P_i$, NG, SP, $F_i$, $\text{Par}_{\text{sel}}$, $\text{Par}_{\text{mut}}$, $\text{Par}_{\text{cr}}$)
If $[P_i] = []$, Initialize ($P_i$, SP, NDV)
for $i = 1$: NG
$[P_i^d] = P_i$
If required, $[P_i^d]$ = Decoding ($P_i$)
If $[F_i] = []$, $[F_i]$ = Fitness_Calculation ($P_i^d$)
$[P_{i-\text{sel}}] = \text{Selection}(P_i, F_i, \text{Par}_{\text{sel}})$
$[P_{i+1-\text{sel}}] = \text{Selection}(P_i, F_i, \text{Par}_{\text{sel}})$
$[P_{\text{new}}] = P_{\text{new}}$ U Crossover ($P_{i-\text{sel}}$, $P_{i+1-\text{sel}}$, $\text{Par}_{\text{cr}}$)
U Mutation($P_{i-\text{sel}}$, $P_{i+1-\text{sel}}$, $\text{Par}_{\text{mut}}$)
$[P_i] = [P_{\text{new}}]$
end

*Figure 1 - SGA Pseudocode (Talaslioglu 2009)*

This fitness landscape is defined by an associated Java program DBMiner. This fitness evaluation program has been developed alongside the GA with the role of evaluating the fitness of individuals in the population, using Stanford's CoreNLP (Manning et al. 2014) to extract keywords from the question and ConceptNet (Liu & Singh 2004) to compare the semantic meaning of resulting columns and rows from the resultant dataset after executing the SQL generated by the individual. Parallel inter-application communication with DBMiner is done using TCP Sockets to evaluate individuals. This process is outlined in Figure 2.

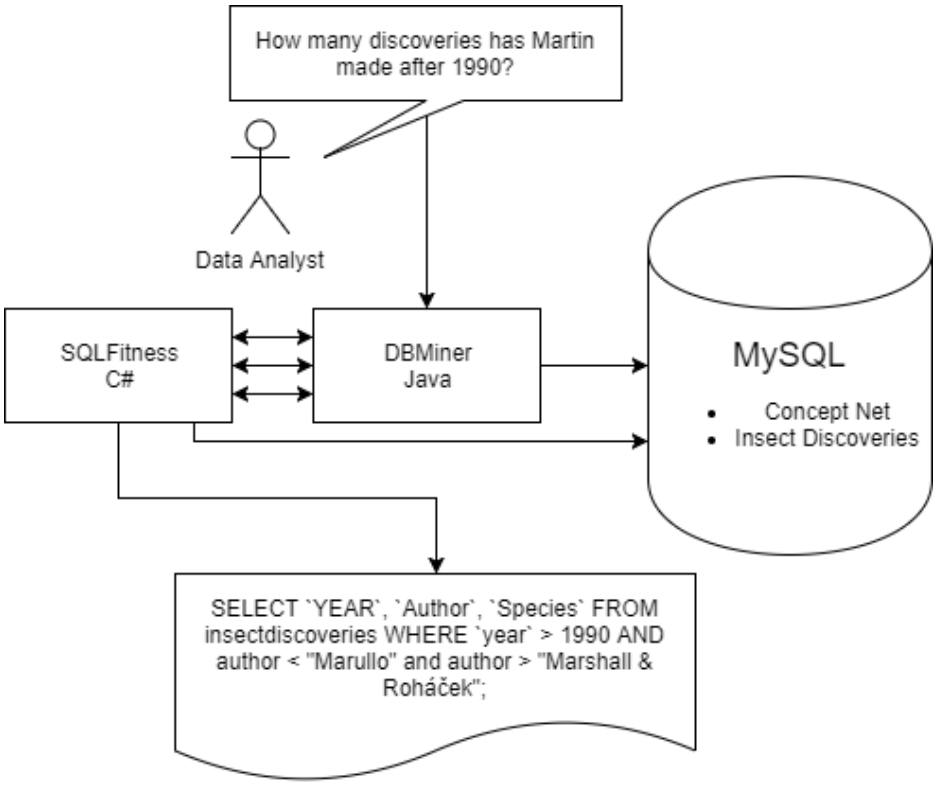



*Figure 2 - Architecture of the data summarisation system*

### 1.1. Research Question

The broader goal of this project is to implement a pipeline which will take a user's question, and return a subset of a flat database as a query to the user which allows the user to perform some trivial further analysis to answer their research question. For example, returning the count of discoveries that Martin made is not as useful as returning the set of all of Martin's discoveries and allowing the user to first verify the correctness of the solution and then performing a trivial count operation. The solution takes more than 10 hours to find a plateau of the fitness of generated individuals based on 9 preliminary trials on NeCTAR and a workstation computer. As is shown in Figure 1 all GA operations depend on the individual and is thus a viable candidate for optimisation efforts.

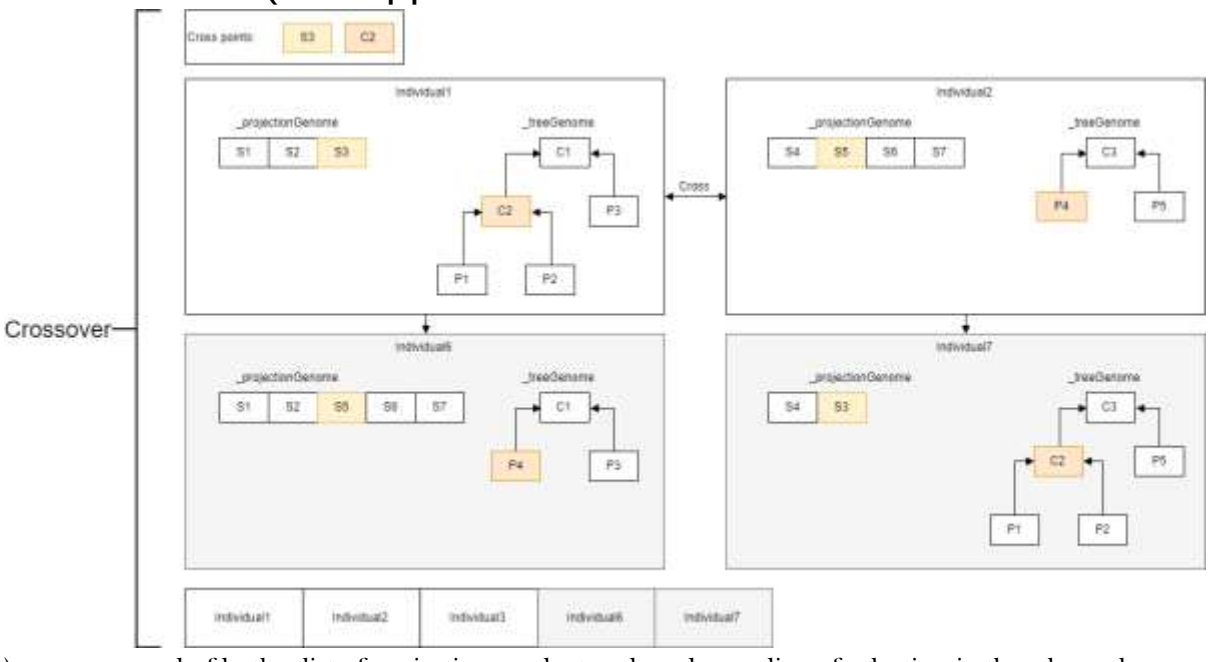
This investigation seeks to determine whether a flat or tree based implementation of an individual is more computationally efficient. As the problem bears some similarity to previous studies done by (Palmer & Kershenbaum 1994) it would be expected that the tree based individual would be a more suitable unit for the GA design, as it should converge on a global optimum in less iterations than a more standard GA implementation. It is expected to be more computationally expensive when compared to a flat individual per generation since the immutable node based structure requires the creation of a considerable number of new objects during crossover and mutation steps.

## 2. Methods

### 2.1. Individual composition

Design of the GA used existing literature to select implementations of the elementary operations best suited to the chosen problem domain. An elitist selection method was chosen for the selection step, as this guarantees an the fittest individual will be at least as fit as the previous generation, while tending to find local maxima (Palmer & Kershenbaum 1994).

## 3. Two independent implementations of an individual are implemented and compared in this investigation. Tree based individuals (see Appendix




) are composed of both a list of projections and a tree based encoding of selection in the where clause, shown in Figure 3. Each of these Nodes is immutable and if a branch on a Node is replaced, every parent node and that immutable node itself must be instantiated with the new references. A Flat individual implementation is the same as the list based projection chromosome, however selection

genes otherwise in the Tree structure are now added into the list. Note that the order of operation information encoded in the tree structure is lost in the Flat individual implementation, as all the WHERE clause is combined using AND operations.

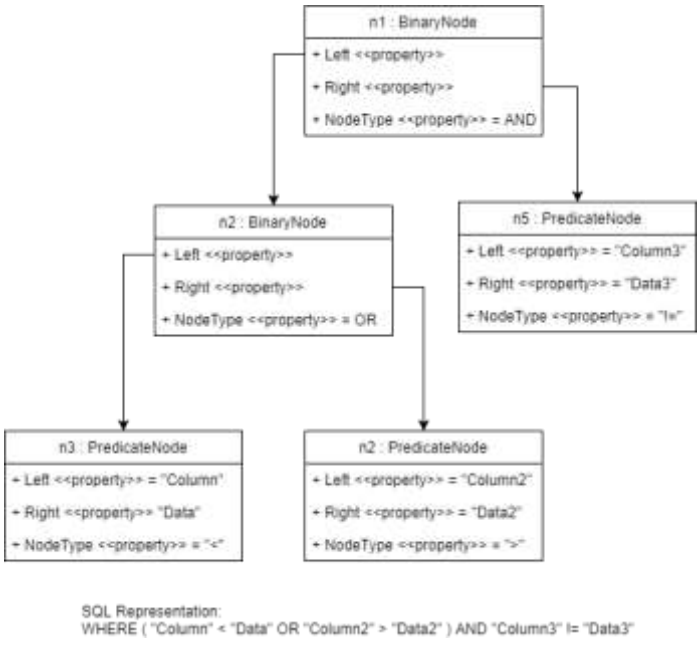



*Figure 3 - How an SQL WHERE clause is represented in a TreeIndividual*

One point crossover was implemented, however due to the compound nature of the Tree based individual the projection chromosome and tree based selection chromosome must be crossed over separately. Flat individuals are crossed over in exactly the same way as the projection component of the Tree individual is crossed. Specific mechanisms for the crossover operation are detailed in the Appendix.

Mutation is controlled by a rate parameter, which will cause a random individual to have a random gene in the chromosome reinitialised. In a Flat individual, this replaces the single gene in the chromosome array and a Tree individual will replace the Node and all predicate nodes. The tree structure cannot be modified, so that a the type of the Node is guaranteed not to change.

## 3.1. Modifications for Testing

In order to test the runtime efficiency without having to wait for the DBMiner to perform a relatively expensive fitness evaluation, several modifications were made to test only the SQLFitness program. As a result of the object oriented design, a mock fitness function was introduced which evaluates every individual at 0. Additional calls to the MySQL database were removed and replaced with simple lists of approximate size of data in the dataset to reduce the number of variables effecting runtime.

Additional logs were written to file to note the time taken for each generation's steps to complete, which were then averaged and graphed over 1000 iterations. The "Performance Profiler" in Visual Studio 2017 was run separately to further investigate discrepancies in the crossover operations of each individual to generate tabulated cumulative exclusive runtime data.

# 4. Results

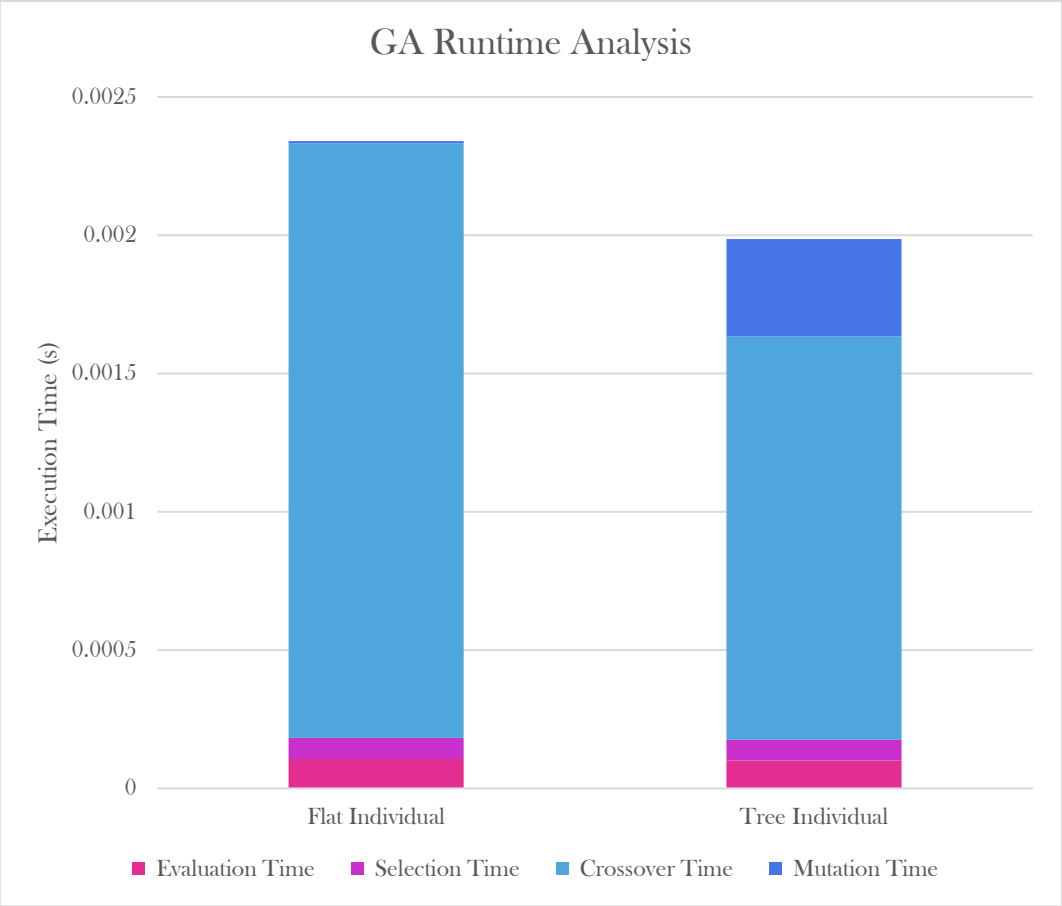



*Figure 4 - Genetic Algorithm average execution time i=1000, size n=100.*

Figure 4 shows that the crossover operation with a FlatIndividual data structure is more expensive with a 100 element chromosome. A TreeIndividual takes less time to perform crossover with the predicate of size 49 and Node based tree structure with a branch size of 51 elements.

*Table 1 - Further investigation of runtime efficiency of FlatIndividual with 1000 iterations*

**Flat Individual**

| Approx. Function Name | Elapsed Exclusive Time % |
|---|---|
| System.Console.WriteLine(string) | 78.07 |
| SQLFitness.Population.<>c.<Sort>() | 2.67 |
| System.Linq.Enumerable.ToArray() | 1.63 |
| SQLFitness.FlatIndividual.CrossWithSpouse() | 1.19 |
| SQLFitness.StubIndividual.get_Fitness() | 1.12 |

*Table 2 - Further investigation of runtime efficiency of TreeIndividual with 1000 iterations*

**Tree Individual**

| Approx. Function Name | Elapsed Exclusive Time % |
|---|---|
| System.Console.WriteLine(string) | 73.88 |

| | |
|---|---|
| SQLFitness.Population.<>c.<Sort>() | 2.39 |
| SQLFitness.TreeIndividual.CrossWithSpouse() | 1.37 |
| SQLFitness.Utility.GetRandomValue() | 1.21 |
| SQLFitness.StubIndividual.get_Fitness() | 1.17 |

Table 1 and Table 2 show the time spent executing the body of each function, not including functions called within the function block. Note that the System.Linq.Enumerable.ToArray() call in Table 1, which occurs in the crossover step of the GA. This also shows there is a difference of 1.37 – 1.19 = 0.18s for $2 \times 10^4$ function calls. A typical evaluation is expected to take approximately 15 times this amount and thus the difference amounts to 2.7s total.

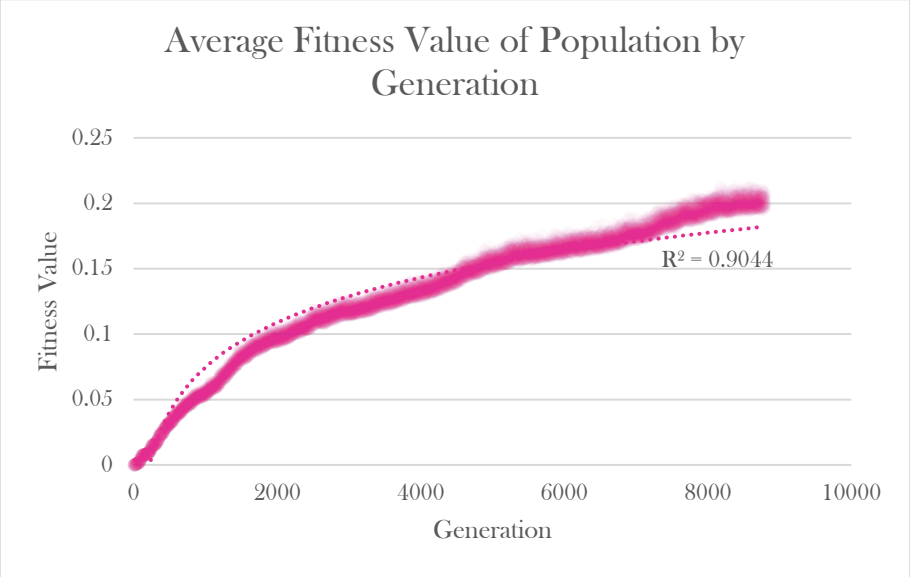



*Figure 5 - Average Fitness value of population by iteration*

Figure 5 shows the increase of fitness value over time with a population size n=100, with a projection chromosome length of 3 and selection tree of size 5. Approx. 9000 iterations were completed over approx. 13 hours of computation. A logarithmic trendline is able to explain 90.4% of the variation of the average fitness value with respect to the generation. At 1500 iterations the expected value of the fitness function would be approx. 0.2 based on the trendline, which is approximately a plateau of the fitness function for this algorithm. This extrapolation is not ideal as there may be slight differences during the experiment.

# 5. Discussion

Most of the runtime cost of the SQLFitness application was due to logging and string operations. Earlier trials in which there was more IO, printing every as it was evaluated was removed due to the large impact on the results. The only console IO left in was the iteration number, which was still the most expensive operation as was shown in Table 1 and Table 2. MSDN notes that IO is synchronised, that is multiple threads writing to the output will be synchronised (Anon n.d.). The multithreaded component of the evaluation previously was logging individuals and thus had to be removed.

It is expected that the crossover operation is the most expensive, as selection is a sort, and evaluation in this testing environment is negligible aside from the thread creation overhead. Mutation in a list would be expected to be fast, as replacing one gene in a list based chromosome would be expected to be

performed in $O(1)$ time. As mutation of a tree individual requires the reinitialization of all parent nodes, this is executed in approximately $O(n^2)$ where n is the depth of the tree where it's mutated from.

When it became evident that the Flat individuals were on average more expensive to cross over, further analysis was conducted. The call to ToArray() and the fact that C# lists will require the entire array to be copied before inserting a new value (Anon n.d.).

# 6. Conclusion

The conclusion that can be drawn is that while Tree based individuals had a faster crossover step on average, it is expected that the flat individual can be made more performant by calculating the length of the resultant list and allocating memory once, rather than dynamically updating the lists each time an element is added. Further analysis showed that the CrossWithSpouse operation was slightly more expensive in total excluding calls to external functions in the Tree individual than the Flat individual which supports this claim.

In order to decide on which method would best suit the SQLFitness GA, more extensive testing must be done in order to determine the difference in convergence time for each implementation which can be done on the NeCTAR cloud research platform.

Boolean algebra can be used to determine whether the AND combination of predicates is able to represent all possible subsets of the database when encoding is not used, and perhaps a proof demonstrating the chances of producing this via the GA investigated when compared to a Tree based individual.

# 7. Bibliography


Anon n.d., 'BigQuery - Analytics Data Warehouse', *Google Cloud Platform*, viewed 6 November, 2017a, <https://cloud.google.com/bigquery/>.

Anon n.d., 'c# - Using LINQ to remove any value that is a duplicate - Stack Overflow', viewed 6 November, 2017b, <https://stackoverflow.com/questions/20906701/using-linq-to-remove-any-value-that-is-a-duplicate>.

Anon n.d., 'Cloud Natural Language API', *Google Cloud Platform*, viewed 6 November, 2017c, <https://cloud.google.com/natural-language/>.

Anon n.d., 'Console Class (System)', viewed 12 November, 2017d, <https://msdn.microsoft.com/en-us/library/system.console.aspx>.

Anon n.d., 'Google Testing Blog', *Google Testing Blog*, viewed 8 November, 2017e, <https://testing.googleblog.com/>.

Anon n.d., 'List(T).Add Method (T) (System.Collections.Generic)', viewed 11 November, 2017f, <https://msdn.microsoft.com/en-us/library/3wcytfd1(v=vs.110).aspx>.

Anon n.d., 'Multithreaded Applications (C# and Visual Basic)', viewed 22 October, 2017g, <https://msdn.microsoft.com/en-us/library/ck8bc5c6(v=vs.110).aspx>.

Anon n.d., 'Natural Language Interface for Databases | KUERI.ME', viewed 6 November, 2017h, <http://kueri.me/>.

Anon n.d., 'Resolvr', *Devpost*, viewed 6 November, 2017i, <http://devpost.com/software/resolvr>.

Anon n.d., 'StringBuilder Class (System.Text)', viewed 3 November, 2017j, <https://msdn.microsoft.com/en-us/library/system.text.stringbuilder(v=vs.110).aspx>.

Anon n.d., 'Text Classification made easy with Elasticsearch | Elastic', viewed 6 November, 2017k, <https://www.elastic.co/blog/text-classification-made-easy-with-elasticsearch>.

Anon n.d., 'When to use Dependency Injection', *Google Testing Blog*, viewed 8 November, 2017l, <https://testing.googleblog.com/2009/01/when-to-use-dependency-injection.html>.

Butey, PK, Meshram, S & Sonolikar, RL 2012, 'Query Optimization by Genetic Algorithm', *Journal of Information Technology and Engineering*, vol. 3, no. 1.

dotnet-bot n.d., 'Pattern Matching - C# Guide', viewed 3 November, 2017, <https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching>.

Elsevier n.d., '11 steps to structuring a science paper editors will take seriously', *Elsevier Connect*, viewed 10 November, 2017, <https://www.elsevier.com/connect/11-steps-to-structuring-a-science-paper-editors-will-take-seriously>.

Gamma, E, Helm, R, Johnson, R & Vlissides, J 1995, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Goldberg, DE 2006, *Genetic algorithms*, Pearson Education India.

Holland, JH 1975, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI.

Irene Moser 2017, 'Research Meeting'.

Knoernschild, K 2002, *Java Design: Objects, UML, and Process*, Addison-Wesley, viewed <https://books.google.com.au/books?id=4pjbgVHzomsC>.

Liu, H & Singh, P 2004, 'ConceptNet &Mdash; A Practical Commonsense Reasoning Tool-Kit', *BT Technology Journal*, vol. 22, no. 4, pp. 211–226.

Manning, CD, Surdeanu, M, Bauer, J, Finkel, J, Bethard, SJ & McClosky, D 2014, 'The Stanford CoreNLP Natural Language Processing Toolkit', *Association for Computational Linguistics (ACL) System Demonstrations*, pp. 55–60, viewed <http://www.aclweb.org/anthology/P/P14/P14-5010>.

Mesibov, R 2014, 'A dataset for examining trends in publication of new Australian insects', *Biodiversity Data Journal*, vol. 2, p. e1160.

Palmer, CC & Kershenbaum, A 1994, 'Representing trees in genetic algorithms', *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, IEEE, pp. 379–384.

rpetrusha n.d., 'Managed Execution Process', viewed 23 October, 2017, <https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process>.

Salim, M & Yao, X 2002, 'Evolving SQL queries for data mining', *International Conference on Intelligent Data Engineering and Automated Learning*, Springer, pp. 62–67.

Talaslioglu, T 2009, 'A New Genetic Algorithm Methodology for Design Optimization of Truss Structures: Bipopulation-Based Genetic Algorithm with Enhanced Interval Search', *Modelling and Simulation in Engineering*, Research article, viewed 11 November, 2017, <https://www.hindawi.com/journals/mse/2009/615162/>.

tutorialspoint.com n.d., 'C# - Multithreading', *www.tutorialspoint.com*, viewed 22 October, 2017, <https://www.tutorialspoint.com/csharp/csharp_multithreading.htm>.

Watson, T & Rakowski, T 1995, 'Data mining with an evolving population of database queries', *Proceedings of MENDEL*, pp. 26–28.

Whitley, D 1994, 'A genetic algorithm tutorial', *Statistics and computing*, vol. 4, no. 2, pp. 65–85.

# 8. Appendix

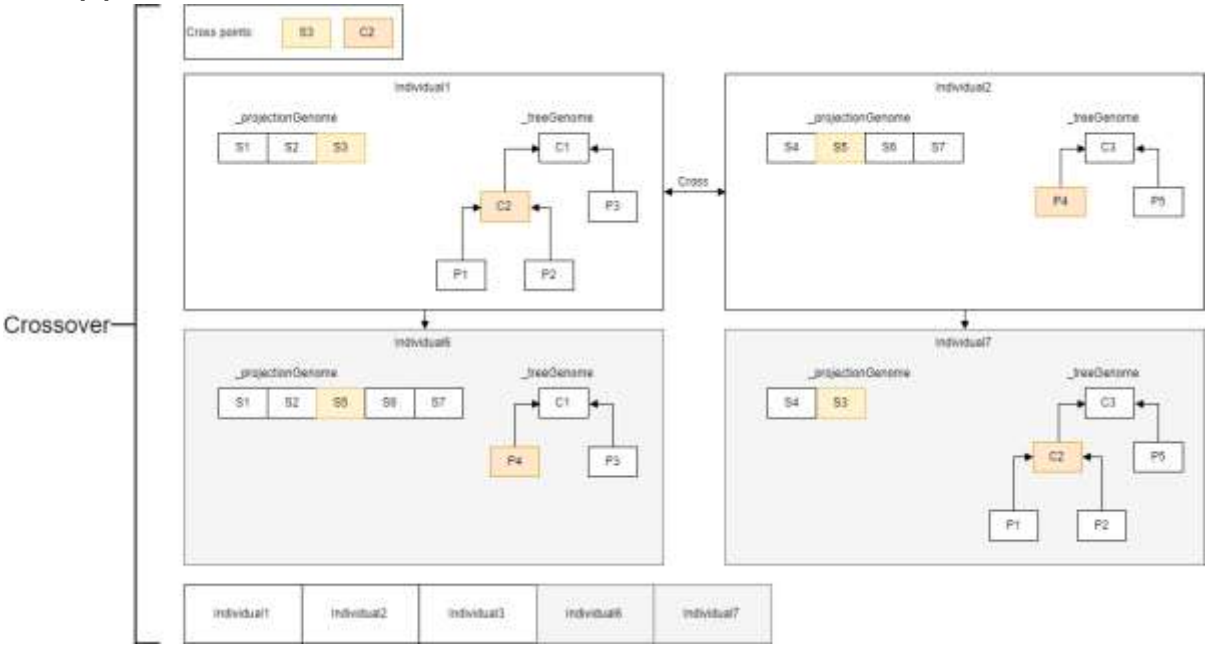



*Figure 6 - A tree individual in this GA has a hybrid chromosome, with both a list based collection of projections as genes and a tree based representation of selection with nodes as genes. Crossover must therefore operate on each of these structures in isolation.*