SQL FITNESS GENETIC ALGORITHM DESIGN REPORT

Using a genetic algorithm to generate SQL queries to mine datasets for relevant information to a natural language research question

NOVEMBER 9, 2017

COS10012
Chris Dilger (101133703)

Contents

Problem Definition

A genetic algorithm has been implemented as a tool to mine sufficiently large datasets of unknown quality for useful data, which can be used to answer a given research question. A user case is outlined as follows:

"Some user has a large flat dataset which the user believes has data useful to the user's research proposal 'x'. The user inputs a natural language research question 'x' into a system which returns a subset of the existing dataset which contains most relevant data and an associated SQL query"

Outlined in this Design Report is an attempt at providing an Object Oriented system to address the above user story. See the associated Research Report for a quantitative analysis on the applicability of the system presented here to address the general user's problem.

Existing Implementations and Solutions

Business Intelligence (BI) solutions to problems in this domain are numerous. To list but a few pipelines developed as commercial products, Google's BigQuery (Anon n.d.), ElasticSearch (Anon n.d.), kueri.me (Anon n.d.) exist to allow natural language queries to search large datasets, be it in an unrelated domain. Even hackathon submissions such as the resolvr (Anon n.d.) project have made some attempt at solutions of a similar nature. Prior work using Evolutionary Algorithms to generate SQL code have been done by (Salim & Yao 2002) in which the resultant dataset is known, and the SQL for generating the set is unknown. The system designed extends on this such that a natural language query is evaluated semantically. The fitness function is implemented in a multithreaded companion application, DBMiner written in Java.

Summary

This SQL Fitness program is designed to generate SQL statements which, when executed return some subset of a database. An associated Java program is used to evaluate how closely a generated SQL query relates to the user's research question 'x'. The associated Java program will return a double value, indicating how well the generated SQL string matches 'x'. Thus, a Genetic Algorithm has been implemented in order to quickly find SQL queries which optimise this fitness function defined by the Java program. An architectural diagram is provided in Figure 1.

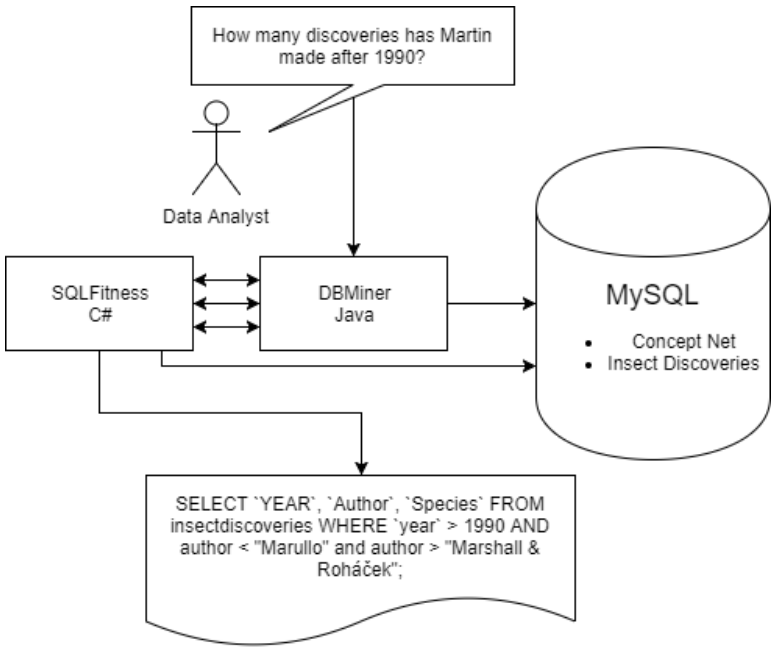

Figure 1 - Architecture of the system this OOP solution is a part of

A Genetic Algorithm (GA) is a member of a class of algorithms designed to find optima in a fitness landscape by randomly generating, combining and then evaluating a population of individuals (Whitley 1994). The fundamental operations are, Crossover, Mutation and Selection.

Selection is the first operation, which operates on a population to remove the least fit individuals. This is shown in Figure 2, which will remove individuals that are the least fit and keep the elite individuals between generations (Palmer & Kershenbaum 1994). In practice, individuals are first ordered by fitness, then a proportion are removed from the population

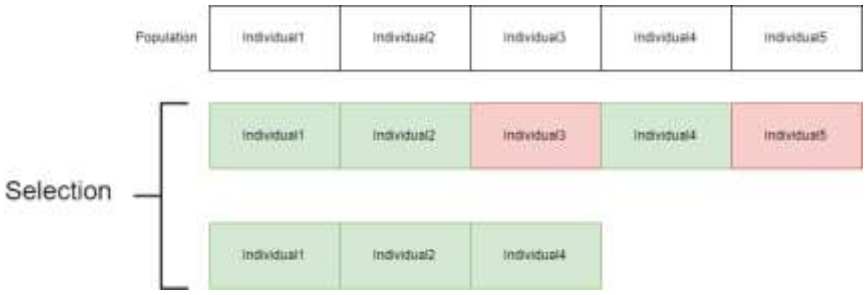

Figure 2 - Selection step from an initial population

Crossover is an operation which creates new individuals from the remaining pool of individuals following selection. Both a flat and a tree based crossover operation are shown in Figure 3 below. Sn refers to a projection, or component of the SELECT statement, Cn refers to a binary combination, either AND or OR in the selection, or WHERE clause in the SQL query. Finally Pn refers to a predicate also in the WHERE clause.

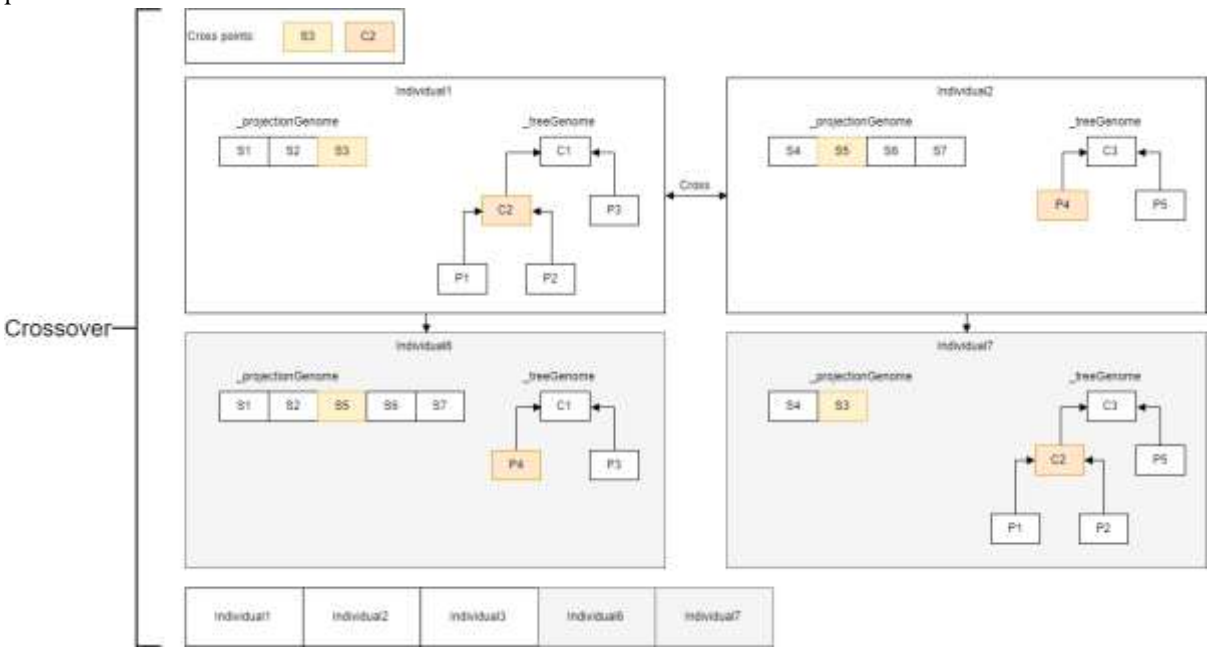

Figure 3 - Crossover Operation with a Tree individual

The final operation in a GA is mutation, which will reinitialise a given part of a chromosome in an individual, much alike the biological analogue in which one small part of an organism's genetic code is changed. This process is illustrated in Figure 4 below, again with a tree based individual, as this implementation makes use of the tree based individual.

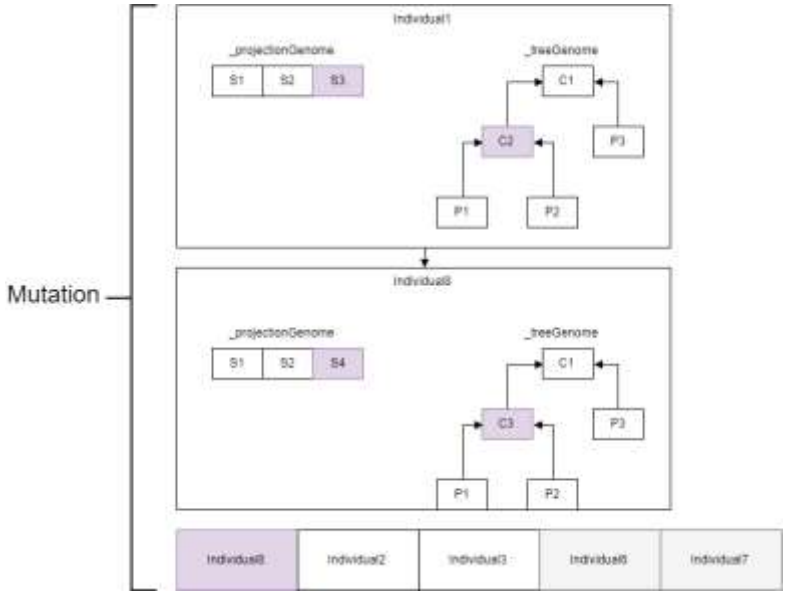

Figure 4 - Mutation of a Tree individual

Several criterion for the application of a genetic algorithm to the problem outlined in the problem definition arise. Namely:

- Communicate with a Java Program using a Client-Server architecture over TCPSockets to evaluate individuals
- Generate and crossover individuals randomly from data stored in a database to ensure all individuals are valid
- Evaluate individuals in parallel
- Reach a reasonable fitness in a short period of time to present a result to the user

Thus, a genetic algorithm has been implemented with these specifications informing all design decisions. In a follow up research report these criterion are used to partially evaluate the applicability of the SQL Fitness program to solve the defined problem.

Technology

Various technology in the form of .NET and MySql libraries, external programs and development methodologies have been incorporated into the program design. Several of these technologies are introduced here conceptually with concise definitions and use cases to outline components of the resulting program.

MySql.Data.MySql client is used to communicate with the database via an adapter object to retrieve valid columns and valid data for the projection (Sn) or predicate (Pn) components of an individual's genome. This allows the GA to produce individuals representing valid queries, with projection and selection data sourced from the dataset itself.

TCP Sockets are used for interapplication communication between this SQL Fitness program and the associated Java program. Multiple sockets are used in parallel to evaluate a number of individuals concurrently. The .NET Framework 4.5 is used to manage TCP connections, and manage threading.

Extension methods are used to extend existing classes and provide easy to use access to IEnumerable elements while accessing random elements. Reflection is used to populate an IEnumerable of enumerable values, so that when a Predicate is instantiated one of the comparison operators is chosen at random.

Each of these technologies contributed to the development of the SQL Fitness program, reusing published and tested code wherever possible, and allowing two developers with diverse programming backgrounds to collaborate on a single project by separating fitness evaluation and the implementation of the GA into separate programs.

Roles and responsibilities

Defining the roles and responsibilities of objects is one of the core tenets of object oriented programming. Any genetic algorithm will have the same general 3 steps (Whitley 1994). Most illustratively of the inner workings of the GA will be the Evaluation and Crossover steps, as these are the most novel in this implementation.

First evaluation of individuals is discussed in the context of OOP design decisions. Many of the more interesting design decisions involve this step, as well as this being the most expensive computationally at runtime. Figure 5 illustrates an interaction with the TreeSelectionAlgorithm communicating with the associated Java program, which executes the individual interpreted SQL to return a double fitness value. See Appendix I for a class diagram showing the inheritance relationships.

Here we see very clearly how the ClientFitness takes care of the evaluation step from the perspective of the TreeSelectionAlgorithm. Here the ClientFitness has the clear role of evaluating Individuals, which it takes responsibility for. As is shown in the diagram, this has resulted in loose coupling. An algorithm needn't know about NetworkStream communication or other implementation details in order to evaluate an Individual.

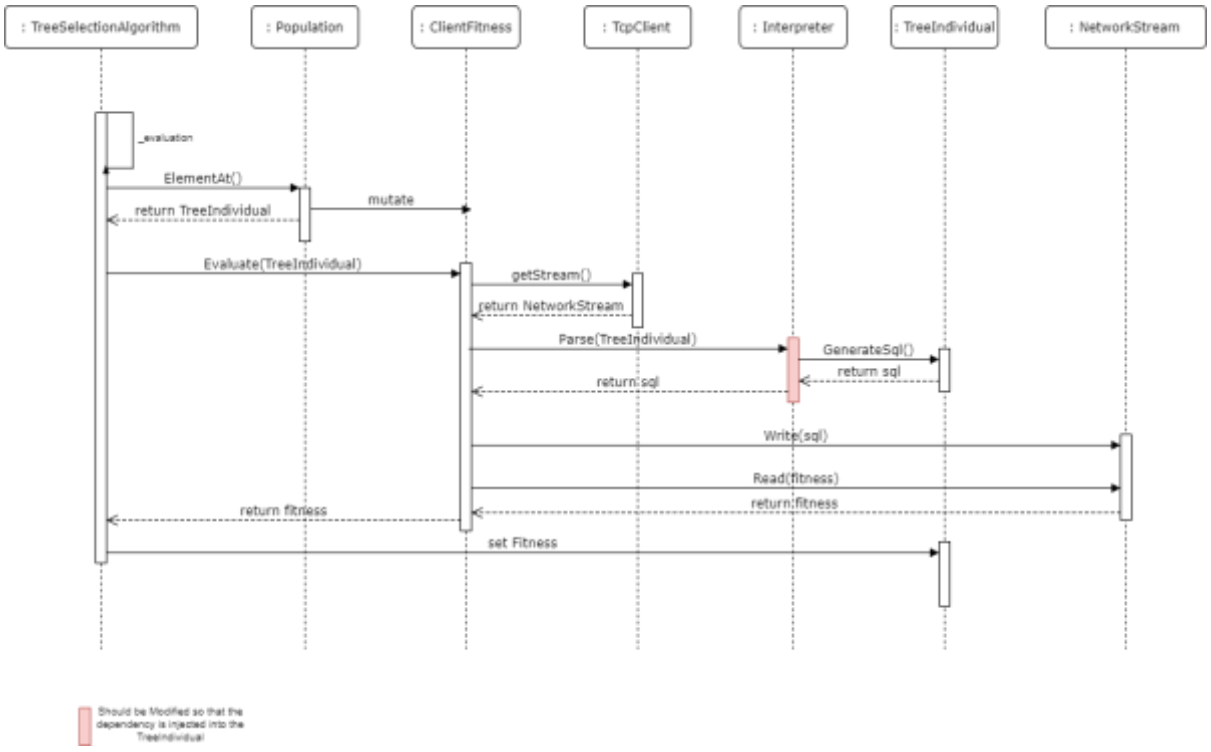

Figure 5 – Evaluation step in the Genetic Algorithm

Design Patterns

Examining the crossover operation shows how the more influential and less natural design patterns have been applied to this program to create extensible, communicable and maintainable solutions to problems. Figure 6 shows how the CrossWalker and AddBranchWalker visitors have been assigned the responsibility of extracting a subtree and appending a subtree to an object collaboration, thus completing the cross over of two Chromosomes composed of a tree of Node objects. The tree structure

is outlined in the Object Collaboration section which examines the implementation of the Node object based trees.

The visitor pattern is typically implemented by giving an "Element" a "Visitor" which will then be used by the element to visit many other elements. Visitor is best applied when there exists a data structure that is relatively unchanging which has an extensible or unknown number of operations applied to it (Gamma et al. 1995). In this implementation, this has been modified such that a Visitor is given the responsibility of visiting Node objects. This modification simplified the implementation for Nodes, as the language features of C# did this work, arising from the ability to overload functions and treat a visit operation polymorphically. Instead of requiring a Node to know which visitor method to call for the Node to be visited, instead the Visitor uses function overloading to define different visit methods based on the type of the visited Node.

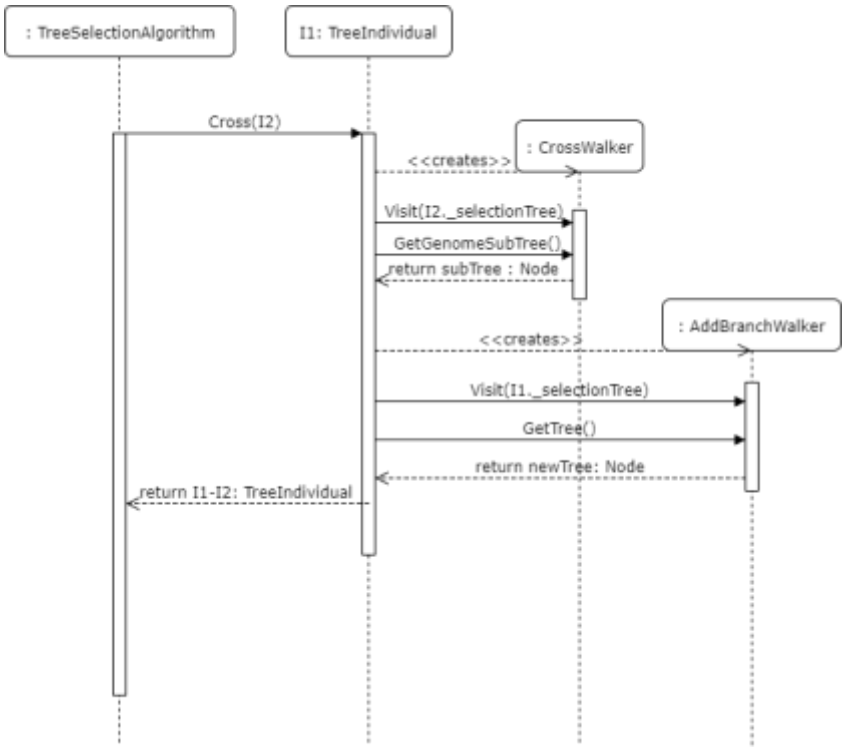

Figure 6 - Crossover operation

In addition to the two operations defined earlier, in which a Node based Chromosome is cut and then appended to, an InterpretVisitor has been defined to generate WHERE clauses for the evaluation step. An example data structure is outlined in Figure 7 with an associated SQL WHERE clause output. Adding, for example an XQuery interpreter would be almost trivial, as the existing data structure can be used and another visitor operation added.

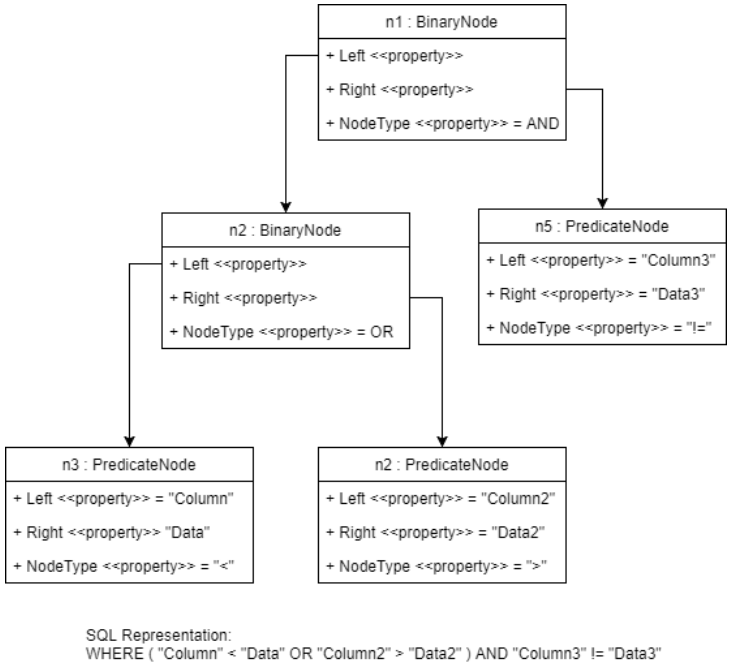

Figure 7 - Representation of an SQL WHERE clause using a Binary Tree

The visitor pattern makes the definition of this type of operation trivial. Extensive discussions with members of faculty around the requirements of the crossover, mutation and random initialisation of this data structure shaped the selection of this pattern. Genetic Algorithms can only find optima in a search space if the randomly generated individuals are able to generate individuals that span the entire search space. A programmatic approach was required which both allowed the crossover operation and allowed the combination of predicates in such a way that order and binary information was retained between generations. Thus a tree solution, in which both the order of operations and the predicates themselves could be represented is presentedabove.

During the initialisation step of the GA, these tree structures must be generated randomly. To achieve this a modified Builder design pattern was used, while still using recursive calls to its build operation, instead of being passed some source from which a tree structure must be generated, as is the case with RTF parsers which are required to allow different representations that are constructed (Gamma et al. 1995), only simple constraints on the resulting tree are passed to the builder. The result is an object similar in many ways to the Visitor defined in this implementation, as the builder is asked to build the resulting object in a similar way to the Visitor is asked to visit a node. Using a builder in conjunction with a visitor is common, as the builder generating the data structure for the visitor to operate on is frequently desirable.

Encapsulation

Encapsulation is the containment of data and behaviours in objects in such a way as to promote object cohesion, reduce coupling and thus create extensible and maintainable solutions. During the development of this program, at its inception there was no recognised need to include Tree individuals as outlined in Appendix I, and thus all individuals were Flat individuals. Flat individuals are conceptually identical to the _projectionGenome section of the Tree individual. All Flat individual WHERE clauses are combined with "AND" only and have no operator precedence encoding. Once the necessity to combine the predicates in a way that doesn't loose information and provide better search space coverage, a Tree individual was designed.

In practice, because a StubIndividual exists as a type of abstract definition of an Individual, it was simple to connect the existing LoggingAlgorithm to the new Tree individual. Examination of the Population datatype in Appendix I show that almost no changes to the code implementing the Algorithm itself was

required. In fact, provided that a Flat individual is not crossed over with a Tree individual, it would be possible for both Individuals to be included in another algorithm implementation. These properties show how tight the encapsulation around the StubIndividual is.

Object Collaboration

Creating reusable code is often done by creating collaborations between objects. "The Composite Reuse Principle prevents us from making one of the most catastrophic mistakes that contribute to the demise of an object-oriented system: using inheritance as the primary reuse mechanism." (Knoernschild 2002). In conjunction with the Visitor pattern outlined earlier, we use the principle of Composite Reuse to define tree structures of nodes. In this case, we see a collaboration of Node objects, defined in the UML Class Diagram in Appendix I and use demonstrated in the UML Object Diagram in Figure 7. Rather than using inheritance to define our types of trees, we instead define small units of objects that create a larger data structure themselves. Instead of using single tree objects, object collaborations are defined to represent data in a reusable way.

Inheritance and Polymorphism

Any good OOP design will apply Inheritance and Polymorphism to create reusable and easy to use code. The LoggingAlgorithm as an abstract base class for all Algorithm implementations allows the predefined set of GA operations of Selection, Crossover, Mutation and Evaluation to be enforced by the compiler. Thus, it is impossible to create a LoggingAlgorithm without all of the required steps to implement a GA. This is done by allowing a virtual function in the LoggingAlgorithm, Evolve() to call each of these in sequence as is shown in Figure 8. See Appendix I for an overview of the LoggingAlgorithm in the context of it's derived classes, which allows all Algorithms to get automatic logging provided they inherit from the LoggingAlgorithm.

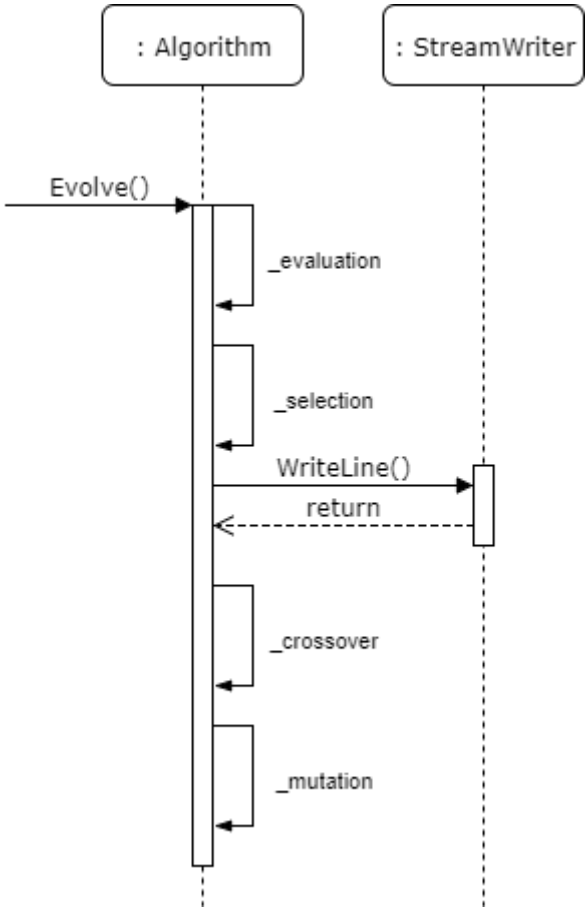

Figure 8 - Implementation of the LoggingAlgoritj6hm for a GA

Application of Delegates and Interfaces

Development of the genetic algorithm presented itself with many opportunities to make use of delegate types and interfaces to have dependencies injected into existing objects. Many situations presented themselves, to which a heuristic approach was applied making use of either Object Methods as delegates, lambda expressions or complete interface definitions.

Lambdas were used in cases in which code implementation only required a single line. This is the case for the Population object, which is asked to initialise a population in it's constructor. The population is passed a factory lambda function, which simply initialises the type of StubIndividual, either a Tree individual or a Flat individual. Code in this case creates more verbose calls to the constructor, which are inherently self documenting because all information is defined at the constructor call.

In cases where an operation required several steps, or ongoing communication was required, or dependencies on other objects existed, an interface was declared. This is the case for the IFitness, which is only required to implement an Evaluate() method. While conceptually possible to use a lambda expression for this purpose, and may even have some advantages, the existence of an explicit contract between the algorithm and the IFitness for how an Individual can be evaluated is beneficial in this case. As the code is read more often than it is written, reading over the IFitness interface provides a level of abstraction that the developer needn't read past, provided the implementation of the IFitness is not of concern. Each of these technologies was evaluated carefully when applied to solve OOP problems when passing data between programs.

Testing and Future Improvements

Unit tests and Visual Studio's debugger integrations allowed for a streamlined and productive developer experience. Unit tests were defined for all tree based operations. This was a good application of the Unit Testing Framework, NUnit as it provided the ability to map out the tree structures, plan the expected outcome of a given visitor operation and then make use of the debugger to investigate where the implementation differed from the specification. In addition to this, as a reader the Unit Tests are a form of documentation, and provide examples of the classes functionality without giving an example of the use case.

Debugging malformed output, race conditions and uncommon exceptions relied heavily on Visual Studio's Debugger. Cases in which the generated SQL was malformed were identified by using conditional breakpoints matching specific occurrences in the SQL. Special characters were causing parsing issues in the ClientFitness function when sending generated SQL over a socket. Race conditions arising from the use of one instance of the ClientFitness caused repeated read errors, in which multiple individuals were returned the same fitness. Further, exceptions which were uncommon were debugged by examining the stack trace and the state of variables in scope. Errors handling a cut point of 0 were found in all of the Visitor classes. Applying the debugging and development tools to provide correct and timely solutions to programming errors by examining the program state benefitted the resulting program.

A number of critiques can be made of the object oriented design. Illustrated in red in Figure 5 is the Interpreter class. Individuals are responsible for generating SQL from their contents, thus the Interpreter class is almost redundant. To improve this OOP design, I would propose a solution in which an interpreter is given to the Individual, which then uses the Interpreter to generate the output. In this way, Individuals could be more output agnostic, and be used to generate XPath or other queries without being modified. For each type of output only a new Visitor and a new Interpreter would be required for Tree individuals.

Interdisciplinary Collaboration

Java program by Farhad Zafari designed to semantically compare the resulting dataset from SQL individuals to a question posed by a user. An understanding of the implementation of the Java program is outlined here in brief. Stanford's CoreNLP (Manning et al. 2014) is used to extract meaning from user queries. Then, ConceptNet (Liu & Singh 2004) is loaded into a ConcurrentHashMap from MySQL which is then used to test the semantic similarity between each of the words in the query and every word in the resulting dataset once an SQL individual is executed. Finally a socket connection transfers the calculated fitness value back to the SQL Fitness program where the GA continues.

Bibliography

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

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

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

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

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.

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>.

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.

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

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

Appendix I

UML Class Diagram for the core functionality of the SQL Fitness program