CS 307: Genetic Programming — Fall 2021
Prof. Helmuth, Hamilton College

Name: Adam Valencia
Date: December 7, 2021
Technical Report

1 Introduction

Genetic Programming aims to provide a useful program over generations of "unfit" programs. Carrying over concepts from biology, as the generation of our programs increase we should see a steady increase in our fitness. In other words, the longer a program is given to evolve, we should see our program provide better results on the problem it aims to solve. In my project, I evolve SQL queries that, given a set of positive examples, should return the positive individuals who match the evolved SQL query. The database was randomly generated based on 7 attributes; Name, Age, Major, Race, Income, Country, Glasses, Blood Pressure (BP), and Tobacco use. In my case, I would expect the SQL to return a query that focuses on age, income, BP, and tobacco use. These are all indicators that a person is at high risk of having a heart attack.

name	major	age	income	race	country	glasses	bp	tobacco
Adams	PSYCH	50	6620	PACISL	AUSTRALIA	TRUE	107	TRUE
Adkins	CPSCI	42	41665	PACISL	JAPAN	FALSE	141	FALSE
Alexander	CPSCI	68	13337	PACISL	AUSTRALIA	FALSE	88	TRUE
Allen	CINEMA	52	32333	ASIAN	RUSSIA	FALSE	104	FALSE
Alvarado	CINEMA	19	9677	NATAM	MEX	FALSE	142	FALSE
Ashley	MATH	28	41727	ASIAN	AUSTRALIA	FALSE	115	TRUE
Bailey	CINEMA	38	15948	NATAM	JAPAN	FALSE	102	TRUE
Baker	PSYCH	46	48800	WHITE	AUSTRALIA	TRUE	110	TRUE
Banks	MATH	32	23715	NATAM	USA	TRUE	125	FALSE
Banksy	PSYCH	36	12759	ASIAN	MEX	TRUE	97	TRUE
Beasley	MATH	29	13377	PACISL	USA	FALSE	107	FALSE
Bender	MATH	29	22145	PACISL	JAPAN	FALSE	92	TRUE
Benson	LIT	44	43749	BLACK	AUSTRALIA	TRUE	96	TRUE
Bentley	MATH	48	52336	BLACK	AUSTRALIA	FALSE	99	TRUE
Benton	PSYCH	39	14082	PACISL	RUSSIA	TRUE	132	TRUE
Bishop	LIT	57	48182	PACISL	RUSSIA	FALSE	122	FALSE
Black	CPSCI	70	16551	ASIAN	RUSSIA	FALSE	143	FALSE
Blackwell	CPSCI	64	50012	WHITE	USA	TRUE	130	TRUE

Figure 1: Rows from the sample database

2 Implementation

The idea of evolving SQL queries came out of my previous knowledge of Database theory and the introduction of PUSH. The goal was to use the final programs as an embryo for our actual solution. In this case, given a Database D I wanted my final program to return a where clause. This clause would then be inserted into our embryo to query the database

such as *SELECT * FROM D WHERE (age=62)*. Thus, whatever is inside of the parenthesis in the final query comes from our evolved programs. I could accomplish this my including multiple stacks, each representing a datatype that would be necessary to query a database.

2.1 Evolution

Some stacks used included **:WHERE, :QUERY-RESULTS, :BOOLEAN, :STRING**. The incorporation of boolean and string were necessary since some attributes of the table had boolean and string values so it was necessary to save that information in another stack. In my implementation, I included three new instructions, (Helmuth, T. Spector, L. 2012.) condition_from_stack, condition_from_index, condition_distinct_from_index. Condition_from_stack will pop two integers off the integer stack and use them as indices for the SQL operators . One index will be used to pull a known attribute from the database while the second index will be used to indicate the SQL operator being used (¡,¿,¡=,¿=,=). Lastly, depending on the datatype of the attribute, a third value will be popped off the stack that corresponds to the datatype. If any of these stacks are empty during these operations, the program no-ops.

```
(defn condition_from_stack
 "Condition clause makrers. Pops two integers off the integer stack to index an attribute and operator from the table.
  Depending on the attribute type, pushes a where clause to the :where stack that matches the datatype."
 [state]
 (if (< (stack-depth state :integer) 3)
   state
   (let [attribute_idx (mod (peek-stack state :integer) (count database-attributes))
         operator (operators (mod (peek-stack (pop-stack state :integer) :integer) (count operators)))
         condition_type ((nth (keys database-attribute-types) attribute_idx) database-attribute-types)
         new-state (pop-items-from-stack state :integer 2)]
     (cond
       (= "class java.lang.String" condition_type) (if (empty-stack? new-state :string)
                                                     state
                                                     (push-to-stack (pop-stack new-state :string) :where (str "(" "'" (peek-stack new-state :string) "'"
                                                                                                              operator
                                                                                                              (database-attributes attribute_idx) ")")))
       (= "class java.lang.Integer" condition_type) (push-to-stack (pop-stack new-state :integer) :where (str "(" (peek-stack new-state :integer)
                                                                                                             operator
                                                                                                             (database-attributes attribute_idx) ")"))
       (= "class java.lang.Boolean" condition_type) (if (empty-stack? new-state :boolean)
                                                      state
                                                      (push-to-stack (pop-stack new-state :boolean) :where (str "(" (peek-stack new-state :boolean)
                                                                                                                operator
                                                                                                                (database-attributes attribute_idx) ")")))))
```

Figure 2: Rows from the sample database

Condition_from_index and Condition_distinct_from_index are two instructions that help in guiding the program towards favorable attributes in the table. Like condition_from_stack, two integers are popped off the integer stack to return both an attribute and a table. The difference is that these two then use the database to return a value that exists rather than one randomly generated from popping off a stack. What makes Condition_from_index different from Condition_distinct_from_index is that the value that makes the condition is equally likely of being chosen depending on the database. Therefore, if the age of our demographic has many people that are 62, the chances of returning 62 as a value is alot higher. This contrasts Condition_distinct_from_index as the values for each attribute are in a set. This

makes the value that complete the condition equally likely of being chosen which can be useful in favoring clauses that query successfully via the database.

3 Modifications

There were three major changes to my GP implementation that were specific to my project; lexicase selection, initialization method, and fitness evaluation. Out of these three changes, lexicase and the fitness evaluation proved the most useful in generating meaningful queries.

3.1 Lexicase Selection

Once I implemented my base system there was a very strong inclination for the program to prematurely converge around generation 50. At the beginning, the cause of this was my fitness evaluation which I restructured and will explain later in this paper. Once this change was made, individuals were returned who made meaningful queries. The database used consisted of 300 people. An individual represents a query on the database. I used lexicase

```
(defn lexicase-selection
 "Lexicase Parent Selection for an individual"
 [population]
 (loop
  [testCases (shuffle (concat incorrect-query correct-query))
   eligible population]
   (cond
     (= 1 (count eligible)) (first eligible)
     (empty? testCases) (if (empty? eligible) (rand-nth population) (rand-nth eligible))
     :else (let [testCase (first testCases)
                 survivors (filter (fn [x] (let [result (:results x)
                                                 category (:category testCase)]
                                             (cond
                                               ;;if nil? returns true then that means we could not find our testCase in the query results
                                               ;;if some? returns true, then that means that we found our testcase in the query results
                                               (and (some? (some #{testCase} result)) (= 0 category)) false
                                               (and (nil? (some #{testCase} result)) (= 0 category)) true
                                               (and (nil? (some #{testCase} result)) (= 1 category)) false
                                               (and (some? (some #{testCase} result)) (= 1 category)) true
                                               :else true)))
                                   eligible)]
             (recur (rest testCases) survivors)))))
```

Figure 3: Lexicase Parent Selection

selection by treating all 300 people in the database as a test-case. In the first test-case, I checked to see if the person was included in my program's query and whether or not it belonged in the query. This was made to favor individuals who either had true positives or true negatives thus creating more complex conditions. On a specific test-case, if the program had a true positive or true negative, it was allowed to continue into the next iteration of the lexicase selection. I believed that using lexicase for both parent selections would help further improve fitness but when this was done, the population was essentially eliminated and replaced with maximum error individuals. I think the cause for this could be in the case selection for individuals. Since we are not strictly looking at fitness in our lexicase selection,

low fitness individuals aren't generally considered so they are loud to reproduce further. Another issue I encountered with this is that lexicase selection made multiple queries thus dramatically improving the run-time of my program. To mitigate this, whenever a program made a query, I saved their results in a map that way access speeds were much faster in this parent selection method. This improvement benefited my program as the best error returned after 400 generations went down from 34 to 28 with a program size of 16 to 6. Lexicase helped improved program evaluation at the expense of run-time (before optimization) and increased program size.

3.2 Initialization and Fitness

When running tests for Genetic Programming I noticed that premature convergence was happening especially after generation 50. To mitigate this, I thought about favoring conditions that used the AND and OR operators. These operators pop two things off the :where stack, combines them via AND or OR, and then pushes the result onto to :where stack once more. The issue with this is my fitness function was adding false positives and false negatives. If a condition was empty, the fitness would be 34 since an empty list only contains false negatives. If a clause becomes more complex via AND and OR, this will always result in a fitness that is lower since the error will always be higher unless the more complex query returns something truly meaningful. The error of a program could range anywhere between 0-34, highly favored since we have only true positives, or 34-300, less favored since we have an increased amount of false positives. The favoring of AND and OR in turn didn't actually provide meaningful results. What stood our in my fitness function is that age was being highly favored in my clauses. Given that $\frac{2}{5}$ of my instructions manipulate the :where stack given a table I decided to alter these. I thought it would prove useful to remove condition_from_index and condition_distinct_from_index as these allow a larger search space. Although, this increased the run-time, I was noticing individuals who had greater diversity in their conditions.

4 Next Steps

There are three major things that I know will further improve this project. A larger database, a better fitness function, and more diverse attributes can all help improve the results.

4.1 Database

The database was randomly generated for me and this poses two significant issues.

1. The number of attributes where extremely restricted in my program. In my data, age was a huge factor present in whether or not an individual would be at a high risk for a heart attack. Sine programs were discouraged from exploring more conditions because of the fitness function, age was always where my program would converge. This was

partially mitigated by removing the two instructions that referred to the table but after approximately 100 generations programs would once again begin to converge.

2. Having more people in the data-set would also be beneficial since this would dramatically increase the expected range for what the error could be for a program. If we had 1000 persons in our data-set then our error range has more than tripled which will also encourage more diverse programs since they are necessary to avoid local optima.

4.2 Fitness Evaluation

A hiccup in my implementation was how I was measuring fitness. By only looking at true positives, I had an error range of 0-34. Empty condition clauses were favored since an empty list will always be less than an error greater than 34. When the range was increased to 300 via including false positives, conditions were being created. Therefore, if we look at both true negatives and true positives, we could have a fitness evaluation that encourages diversity and mistakes that are overshadowed by only looking at what the programs got right.

5 Conclusion

Evolving SQL queries was engaging as it showed me the various ways we can use GP to create meaningful results in areas where we don't question research and methods. Although my project did not work out as expected, I was able to see the errors that many of our research papers aim to solve further pushing me to find interest in the boundaries GP can push. Given more time, and hopefully not another new laptop, I would spend more time creating a rich database and creating a foolproof fitness function that can supplement the queries the program is making.

6 References

1. Helmuth, Thomas Spector, Lee. (2013). Evolving SQL Queries from Examples with Developmental Genetic Programming. 10.1007/978-1-4614-6846-2_1.