Helmuth, T., and L. Spector. 2012. Evolving SQL Queries from Examples with Developmental Genetic Programming. In Genetic Programming Theory and Practice X, edited by R. L. Riolo, M. Ritchie, J. Moore, and E. Vladislavleva. New York: Springer. In Press.

Chapter 1

EVOLVING SQL QUERIES FROM EXAMPLES WITH DEVELOPMENTAL GENETIC PROGRAMMING

Thomas Helmuth[1] and Lee Spector[2,1]

[1] *Department of Computer Science, University of Massachusetts, Amherst 01003 USA;*
[2] *School of Cognitive Science, Hampshire College, Amherst, MA 01002 USA.*

Abstract

Large databases are becoming ever more ubiquitous, as are the opportunities for discovering useful knowledge within them. Evolutionary computation methods such as genetic programming have previously been applied to several aspects of the problem of discovering knowledge in databases. The more specific task of producing human-comprehensible SQL queries has several potential applications but has thus far been explored only to a limited extent. In this chapter we show how developmental genetic programming can automatically generate SQL queries from sets of positive and negative examples. We show that a developmental genetic programming system can produce queries that are reasonably accurate while excelling in human comprehensibility relative to the well-known C5.0 decision tree generation system.

Keywords: genetic programming, data mining, classification, SQL, Push, PushGP

1. Introduction

In the emerging era of "big data," vast amounts of data are available in many kinds of databases. Unfortunately, many users who have access to this data are unable to use it effectively because they do not know how to extract relevant, concise and comprehensible features or summaries of the data; that is, they do not know what queries to formulate in order to discover novel and useful aspects of the data. This issue can be addressed in part by a system that takes positive and negative example tuples—which it is generally easy for users to provide—and returns concise, comprehensible SQL queries that classify the provided tuples in simple and potentially interesting ways.

The creation of queries from examples can be thought of as a data mining classification problem, which is often one task within a larger "knowledge discovery in databases" process (Freitas, 2002). In this task the objective is to create a comprehensible and interesting query that correctly classifies the given examples. In many cases we have no reason to expect there to be a simple query that perfectly classifies the examples, but we would nonetheless like to create a reasonably simple query that both does a good job at classifying the examples and is concise enough to be easily interpreted by the user.

To make the general problem more concrete, we seek a system that takes as inputs a database D and training example tuples $E = E^+ \cup E^-$ where $E \subseteq D$ and $E^+ \cap E^- = \emptyset$. Here, E^+ is the set of positive examples and E^- is the set of negative examples. The goal of the system is to discover a concise and potentially interesting query Q such that $E^+ \subseteq Q(D)$ and $E^- \cap Q(D) = \emptyset$.

We have developed a system called Query From Examples (QFE) that takes the set of examples E and searches for a query Q that satisfies the above properties. It does this by means of developmental genetic programming. In QFE, each program P creates (or "develops") a query Q_P that is then evaluated on how well it correctly classifies the given example tuples E.

In contrast to other approaches to the production of database queries with GP (see below), this form of developmental GP allows QFE to use standard program representations and genetic operators, along with standard population and evolutionary control parameters. The only change required to use this approach in conjunction with most GP systems is to include new developmental functions in the system's function set. The developmental approach makes it easy to implement systems like QFE on top of existing GP systems and thereby to take advantage of advances in the general state of the art of GP. In addition, it may make

it easier to evolve queries of arbitrary structure, thereby enhancing the generality of the system for a wide range of applications.

In the work described in this chapter we ran QFE on a standard data mining classification task and compared its results to those given by the decision tree classifier C5.0. We find that although QFE does not produce quite as accurate a classifier as C5.0, the classifier that it produces is more concise and comprehensible than the one produced by C5.0. We therefore believe that developmental GP is competitive with, and in some ways superior to, other modern data mining systems on the creation of classifiers.

The remainder of the chapter is structured as follows. The next section describes work that others have done evolving SQL queries. Section 3 describes our QFE system and its implementation. Our experiments and results are given in Sections 4 and 5. Finally, we discuss limitations of QFE, possible improvements to QFE (including generalizations that QFE makes possible but that competing approaches would not), and our general conclusions.

2. Related Work

A variety of research has been conducted that uses GP either for the creation of queries (Castro da Silva and Thomas, 2010; Acar and Motro, 2005) or for data mining (Freitas, 2002; Freitas, 1997; Ishida and Pozo, 2002; Doucette et al., 2012; Veeramachaneni et al., 2012, among many others). Because this literature is quite voluminous and varied we will comment specifically only on those systems most closely related to QFE.

Castro da Silva and Thomas (Castro da Silva and Thomas, 2010) directly evolve queries as individuals with the goal of generating queries for inexperienced SQL users. In order to ensure that evolved queries are syntactically correct they implement numerous non-standard genetic operators to combine and mutate individuals. This approach requires significant re-design of any existing GP system and, we would argue, limits the system's generality. Interestingly, this system seems to be the only prior work in which queries are allowed to include joins across tables, leaving the joining attribute up to evolution.

Acar and Motro (Acar and Motro, 2005) frame their work as trying to provide an alternative equivalent query to a given query by creating the alternative using the results of the original as positive examples. Although their motivation is different from ours, the resulting system has many similarities. Their method assumes that the sets of positive examples and negative examples cover the entire database, instead of a small subset of it. The user must provide the entire set of example

tuples that are in the database, which is probably impossible without using an *a priori* query to fetch them. The given system evolves actual queries as individuals, but can only handle queries expressible as trees of relational algebra expressions.

Freitas describes a GP system that evolves programs that can be interpreted as SQL queries to be used in the data mining tasks of classification and generalized rule induction (Freitas, 1997). Individuals are represented as trees that directly correspond to WHERE clauses of queries. Unlike QFE, this work allows for the evolution of non-binary classifiers via niches that correspond to classes of the goal attribute. This paper was pioneering insofar as it introduced the idea of evolving SQL queries but it presents no experiments or results and it does not make clear how one can deal with practical issues such as the choosing of constants, the design of an appropriate fitness function, the alterations that must be made to standard genetic operators, etc. Because there are no results one cannot judge the system with respect to query accuracy, comprehensibility, conciseness, and time. Additionally, this approach is limited (unlike the developmental approach that we present below) to the production of queries over a single table with WHERE clauses that can be expressed as trees. Freitas has continued to produce a great deal of significant related work but not, so far as we are aware, additional work on the use of GP to evolve SQL queries.

GPSQL is a data mining system that uses grammar genetic programming (different from grammatical evolution) to create SQL queries for classification (Ishida and Pozo, 2002). This GP system uses individuals that are composed of grammar-based derivation trees, where the grammar underlying each tree allows for problem-specific SQL queries to be formed. The use of derivation trees allows genetic operators to replace a node in a tree only with a node that is generated using the same production rule from the grammar, meaning that the resulting children must be syntactically correct. In this respect the system is somewhat like strongly-typed GP (Montana, 1995) with a very large number of types, one for each production rule. Unfortunately, this means that each problem requires an extensive BNF grammar to be defined by the user. The BNF grammars described in this work appear to be highly specialized to specific problems, defining how each condition is formed and with what values an attribute may be compared.

3. Evolving Queries from Examples

We used the PushGP genetic programming system to evolve individuals that create queries. Each individual is a program that manipulates

Table 1-1. Instructions used in our PushGP runs.

Stack	Instructions
integer	add, sub, mult, div, mod, stackdepth, dup, swap, rot
string	length, stackdepth
where	condition_from_stack, condition_from_index, condition_-distinct_from_index, condition_from_pos_ex, condition_-from_neg_ex, and, or

a state that can be interpreted as a SQL query after the program terminates. This type of GP system, in which individuals create executable structures via state manipulation, and in which the resulting structures are subsequently executed (e.g. as database queries) to produce desired outputs, is known as *developmental GP* (Gruau, 1994; Koza et al., 1999). We assign fitnesses to individuals based on how many of the positive and negative example data points the queries that they produce return when run on the database.

Push and PushGP

PushGP is in many respects a generic GP system except that its individuals are represented in the Push programming language (Spector, 2001; Spector et al., 2005). Push is a stack-based language in which instructions fetch arguments from stacks and return results to stacks; each type has its own stack. In Push, programs consist of nested lists of intermingled instructions and literals. Strongly-typed instructions are able to either retrieve the correctly typed arguments if they are available, or act as "no-ops" (and do nothing) if they are not. Push has been implemented in many languages; this work uses the Clojure implementation, which may be freely downloaded at the Push project page[1].

Push allows for many different types to be used within one program, each of which has its own stack. Common types such as integers, floats, strings, and booleans are often used, as are "code" and "exec" types that allow for the evolution of self-modifying programs and novel control structures. Additional problem-specific types can be added when necessary. For evolving queries, we have added stacks for the SELECT, FROM, and WHERE clauses of an SQL query, although we primarily use the "where" stack along with the standard integer and string stacks. Table 1-1 contains the instructions used in the runs reported below.

[1]http://hampshire.edu/lspector/push.html

Programs must also contain literals of the data types that they use. When the Push interpreter encounters a literal within a program it simply pushes it onto the stack of the appropriate type. For the evolution of queries we only use literals that come from ephemeral random constants (ERCs), which are random number or string generators that produce constant literals when they are selected for inclusion in new code. For integer literals, we include two ERCs: one that produces integers uniformly in the range $[0, 100000)$, and one that uses more of a logarithmic scale, in that it chooses a range uniformly from $[0, 10)$, $[0, 100)$, $[0, 1000)$, $[0, 10000)$, and $[0, 100000)$ and then chooses a constant uniformly from within the chosen range. The logarithmic ERC makes small integers, which may be important for use in WHERE clause conditions, more common than with the ERC over the entire range $[0, 100000)$. Additionally, we include a string ERC that produces strings between 1 and 10 characters long that may include any uppercase or lowercase letters as well as any numerical digits.

Developmental GP

As described above, QFE creates an SQL WHERE clause by manipulating a state through developmental instructions. The state is kept and manipulated on the where stack of the Push interpreter state. The where stack can have any number of items pushed onto it, where each item is either a single condition on one attribute or any number of conditions joined by the logical operators AND, OR, and NOT. Each condition may be over any attribute of the table, and is constructed as described below. Examples of possible items on the where stack include `(age > 37)`, `(occupation <= 'jP8WKq')`, and `(((education = 'Masters') OR (hours_per_week < 2999)) AND (sex <> 'h7Fm'))`. To create an SQL query from a Push program, QFE runs the program and takes the top item on the where stack and uses it as the WHERE clause of the query. For our experiments, the SELECT clause always just has "*", and the FROM clause references the only table in the database. By adding new instructions, we could generalize QFE so that it could evolve queries over a database with multiple tables by evolving the SELECT and FROM clauses, as discussed in Section 6.

The instructions that create and connect conditions are given as the where stack instructions in Table 1-1. The instructions `where_condition_from_stack`, `where_condition_from_index`, and `where_condition_distinct_from_index` each create a condition and push it on the where stack. Each of these instructions creates a condition by using three literals off of the integer and possibly string stacks. `where_condition_-`

`from_stack` first pops an integer off of the integer stack and uses it as an index into the attributes of the table, taken modulo the number of attributes in the table. A second integer is popped and taken modulo 6 as an index to decide which comparator will be used, from the set $\{=, <, >, <=, >=, <>\}$. Finally, a value is popped off of whichever stack is of the same type as the chosen attribute and is compared to that attribute to create the condition. The condition composed of the chosen attribute, comparator, and constant is pushed onto the where stack. It should be noted that if this instruction or any of the others do not find the number of arguments they require on the stacks, they act as no-ops.

The two instructions `where_condition_from_index` and `where_condition_distinct_from_index` both operate similarly to `where_condition_from_stack`, except in the way they choose a constant to constrain the condition. Like `where_condition_from_stack`, each of these instructions uses the first two integers on the integer stack to determine the attribute and comparator to use for the condition. However, `where_condition_from_index` and `where_condition_distinct_from_index` select a constant not from a value on a stack, but from a tuple indexed in the database. Both of these instructions use the third integer on the integer stack as an index to a tuple from the relevant table; then, the tuples's value of the selected attribute is used as the condition's constant. The only difference between these two instructions is that `where_condition_from_index` uses the index to select a tuple from the entire table D, where `where_condition_distinct_from_index` indexes into a list of distinct values in the table for that attribute. For example, the entire table may have many tuples where the value of the attribute `age` is 35; these are all kept by the first instruction, where the second only indexes in a list of distinct ages, and therefore only has one index where `age` is 35, or any other age for that matter. The first instruction can be thought of as giving weight to a particular value equal to the number of times it appears in the database, while the second makes all values equally likely no matter how often they occur.

Our QFE runs use two other WHERE-condition creation instructions that act like `where_condition_from_index` but retrieve the constant from the positive examples E^+ or the negative examples E^- instead of the entire table D. These instructions, `where_condition_from_pos_ex` and `where_condition_from_neg_ex`, make it so that only values in the example tuples can be indexed to use for the constant. This bias may make it easier to create conditions that specifically relate to the positive or negative examples.

Each of the above instructions creates a single condition and leaves it on the top of the where stack. The instructions `where_and`, `where_or`,

and `where_not` allow for arbitrarily connected conditions. `where_and` and `where_or` take the top two items on the where stack, join them with `AND` or `OR` respectively, and push the result onto the where stack. Similarly, `where_not` takes the top item on the where stack, puts `NOT` in front of it, and pushes the result onto the where stack. These instructions allow for arbitrary combinations of conditions to be formed. Though we implemented all three instructions, we found `where_not` to be more of a hindrance than a help, since WHERE clauses tended to be clogged by nested NOT calls that cancel each other out. We therefore leave `where_not` out of our instruction set for our experiments.

Fitness Testing

QFE uses the common convention of using error as the fitness measure, with lower fitness indicating lower error and fitness of zero meaning a perfect solution. In order to determine the fitness of an evolved Push program P, QFE first runs the program; after it is finishes executing, the final state is interpreted as an SQL query Q_P as described above. We then run the query over the training examples to get the set of tuples $Q_P(E)$. We use the F_1 score as a measure of fitness of the program. The F_1 score, developed to evaluate classification accuracy in information retrieval settings, is the harmonic mean of precision and recall (Van Rijsbergen, 1979). Precision and recall are defined over true positives, false positives, and false negatives defined as

$$\begin{aligned} true\_positives &= ||E^+ \cap Q_P(E)|| \\ true\_negatives &= ||E^- - Q_P(E)|| \\ false\_positives &= ||E^- \cap Q_P(E)|| \\ false\_negatives &= ||E^+ - Q_P(E)||. \end{aligned}$$

We can then define precision, recall, and F_1 score as

$$\begin{aligned} precision &= \frac{true\_positives}{true\_positives + false\_positives} \\ recall &= \frac{true\_positives}{true\_positives + false\_negatives} \\ F_1 &= \frac{2 \cdot precision \cdot recall}{precision + recall}. \end{aligned}$$

Some programs, when executed, result in problematic stack states or queries that must be handled separately. One such degenerate case is when nothing is left on the where stack of the Push state at the end of program execution. In this case, the program has not produced a

WHERE clause, and is given a penalty fitness that is a worse fitness than will be given to any program that produces a non-empty WHERE clause. A second degenerate case occurs when a query takes more time to run than is allowed by QFE. These queries also receive a penalty fitness, worse than any query that finishes running, but better than a query with an empty WHERE clause.

We initially predicted that overfitting of queries to the training examples would be a problem, creating very large queries that only classify the training examples well and do poorly on unseen test data. To combat this it would be possible to add a parsimony term to the fitness function, scaling a query's fitness based on how concise it is. This might force evolution to focus on sufficiently simple queries, avoiding overfitting to the training data. In practice, we found that QFE evolved sufficiently simple queries without such a term. This may be due to the bounds placed on maximum Push program sizes, or by other dynamics of the system that tend to favor concise queries. In any event, we did not use a parsimony term for the runs described below.

Database Use

Well-designed queries tend to be fast, but poorly designed queries can take a long time to run. Since GP tends to produce and test many strange and bad programs while searching for good ones, fitness testing by running queries can be slow. We extended the implementation of fitness testing in a few ways to speed up the evolutionary process.

Some GP implementations cache the fitnesses of evaluated programs so that if the same program is evaluated more than once, the fitness can be quickly retrieved and not re-calculated from scratch. For the problem of evolving programs that create queries we found that many times there are different programs that produce the same query. We altered the caching so that the system caches the fitness of a query instead of the fitness of a program. In this way, different programs that produce the same query can use the same cached fitness value.

Even with these improvements, some queries run for far too long, significantly slowing down QFE. These anomalous queries, if left to run until finished, would dominate the time QFE takes to evolve a query. We decided to give each query only a certain length of time (0.5 seconds) to run, after which it is cut off and given a penalty fitness value. We found this limit to allow most queries to finish without letting extremely slow queries slow down a run.

Table 1-2. Attributes for the adult data set.

Attribute	Type	Values
age	integer	73
workclass	string	9
fnlwgt	integer	21648
education	string	16
education_num	integer	16
marital_status	string	7
occupation	string	15
relationship	string	6
race	string	5
sex	string	2
capital_gain	integer	119
capital_loss	integer	92
hours_per_week	integer	94
native_country	string	42
greater_50k	string	2

4. Experimental Design

To determine how well QFE finds queries that correctly classify results we compare QFE with C5.0,[2] a modern data mining classification system that produces decision trees that classify data in a way similar to the queries produced with QFE. C5.0 is derived from the widely-used C4.5 system (Quinlan, 1993). C5.0 creates decision trees that identify patterns in the training examples. Each decision tree's internal nodes represent boolean tests over a tuple's attributes and leaves give the predicted class of the given tuple. A decision tree can be used to classify examples or to identify patterns in a way that make sense to a human user. Even though decision trees differ from SQL queries in many aspects, they offer a similar enough alternative to compare with QFE's evolved queries. We will compare these systems on the accuracy, conciseness, and time metrics presented below.

For our experiments, we used the Adult Data Set from the UC Irvine Machine Learning Repository (Frank and Asuncion, 2010). This data set has a single table with 15 attributes and 32561 tuples and is often used to test classification systems. Each tuple contains census data about a person, including 6 integer attributes and 9 string attributes, where string attributes come from discrete sets of options and most integer attributes have wider ranges. Table 1-2 gives the 15 attributes and how many discrete values occur for each in the database. Note that

[2]C5.0 is available at http://rulequest.com/see5-info.html.

Table 1-3. The PushGP parameters used for the 50k-Classification problem.
[1]See (Spector and Klein, 2005) for information on trivial geography.

Parameter	Value
Population Size	1000
Maximum Generations	150
Maximum Program Size	300
Crossover Probability	0.80
Mutation Probability	0.12
Simplification Probability	0.05
Reproduction Probability	0.03
Tournament Size	6
Trivial Geography Radius[1]	10
Node Selection	Unbiased
Fitness Function	F_1 Score

some tuples are missing values for some string attributes, filled in by the string '?'.

We tested QFE on a classification problem presented by the adult data set. This problem, which we call the 50k-Classification problem, requires a system to predict the final attribute of each tuple, which is whether the person represented by that tuple makes more than $50,000 per year. For QFE to evolve queries from examples, we need as inputs a set of positive example tuples E^+ and a set of negative example tuples E^-. For this problem, the entire training database D is used as the example set E, with each tuple placed in E^+ or E^- based on the attribute `greater_50k`. Additionally, the evolved query is not allowed to access the attribute `greater_50k`, since otherwise the problem would be trivial. PushGP parameters for the 50k-Classification problem are given in Table 1-3. Simplification is a genetic operator unique to PushGP, in which sections from the individual's program are randomly removed in an attempt to shorten the program without lowering its fitness (Klein and Spector, 2007). Since each simplification step requires a fitness test, we only used one simplification step per simplification operator, which likely had a very small, if any, effect on the conciseness of evolved queries.

We use a variety of metrics to evaluate queries produced by QFE and compare them to the results produced by C5.0. We are primarily interested in measuring the accuracy and conciseness of a query or decision tree, and the time required by the system. Our primary metric of accuracy is defined as

$$accuracy = \frac{true\_positives + true\_negatives}{||E||}.$$

We are interested not only in how well our evolved queries perform on the training examples, but also how well they generalize to other data. We present accuracy results over both the example tuples, which we consider training data, and over a set of test data. The adult data set comes with separate training and testing sets, where only the training set is available to QFE and C5.0 during the creation of classifiers. For the queries we produce, conciseness is a count of the number of conditions in the query's WHERE clause; for decision trees produced by C5.0, we give the number of leaves in the decision tree. Even though these measures of conciseness are not equivalent they do at least give an idea of the complexity of the results. Finally, we give a rough estimate of the time required to produce the results on a modest machine, though it should be noted that QFE is a rough proof of concept implementation whereas C5.0 has been highly optimized.

5. Results

While we have run QFE repeatedly we present here the results of just one representative run. On the 50k-Classification problem, a QFE run created the following query:

```
SELECT *                                                    (1.1)
FROM adult
WHERE (((((education_num >= 10) AND (marital_status =
      'Married-civ-spouse')) OR (education_num >= 15))
      AND (age >= 28)) OR (capital_gain > 4787))
```

Of all the queries examined during the run, this query had the best fitness on the training data. This query has found some interesting conditions that are good predictors of whether or not a person makes more than $50,000 per year. First of all, it returns all tuples where `(capital_gain > 4787)`. It also returns tuples where `(age >= 28)` and `(education_num >= 15)`. Finally, it returns tuples where `(age >= 28)` and `(education_num >= 10)` and `(marital_status = 'Married-civ-spouse')`. Each of these three sets of conditions lays out an interesting description of people who make more than $50,000 per year. Additionally, this query is easy to break apart into these sets of conditions, making it easily comprehensible.

Performance results for query (1.1) and the decision tree created by C5.0 are given in Table 1-4. C5.0 gives slightly better accuracy and F_1 score results, but QFE is close behind. Interestingly, QFE's accuracy increases between the training and test data, where C5.0's decreases. This may indicate that C5.0 is overfitting the training data more than

Table 1-4. Performance measures for the 50k-Classification problem for the solution query (1.1) evolved by QFE and for the C5.0 decision tree. E columns give measures over the training examples while T columns give measures over the test database which is unseen by the algorithms. For descriptions of metrics, see Section 4.

Algorithm	QFE		C5.0	
Table	E	T	E	T
Conciseness	5 conditions		124 leaves	
Positives in Table	7841	3846	7841	3846
Negatives in Table	24720	12435	24720	12435
Tuples in Table	32561	16281	32561	16281
True Positives	5507	2703	5313	2490
True Negatives	21428	10814	23334	11612
False Negatives	2334	1143	2528	1356
False Positives	3292	1621	1386	823
Accuracy	0.8272	0.8302	0.8798	0.8662
Precision	0.6259	0.6251	0.7931	0.7516
Recall	0.7023	0.7028	0.6776	0.6474
F_1 Score	0.6619	0.6617	0.7308	0.6956

QFE. QFE took about 10 hours to produce its query, whereas C5.0 took less than 1 second. Regardless of the optimizations that could be made to QFE, C5.0 is certainly substantially faster. With respect to conciseness, QFE produced a query with 5 conditions that are easy to understand, as described above. On the other hand, C5.0 produced a decision tree with 124 leaves. Even though this decision tree is more accurate than query (1.1), it does not provide a concise summary of the data in a way that is easily understood by humans.

We must consider why the queries evolved by QFE are so concise despite there being no incentive for more concise queries in the fitness function. Program sizes in PushGP bloat for most problems, and this is no exception; the mean program size increased during the run that produced query (1.1). Throughout the run, the Push program generating the best query tended to be larger than the average program size; even so, the best found query remained relatively concise. We believe the developmental approach taken by QFE allows programs to bloat while using a small number of developmental instructions, resulting in a small evolved query. There is probably also some evolutionary pressure against overly large queries, which may be more likely to be degenerate.

6. Conclusions and Future Work

We have presented a system called Query From Examples (QFE) that takes, as input, a database and sets of positive and negative examples

and produces, as output, an SQL query that characterizes the classification implied by the examples in a concise and human-readable form. We used developmental GP to implement QFE on top of an existing GP system (PushGP) with little modification. Compared to the well-known C5.0 decision tree system, QFE is substantially slower and slightly but not unreasonably less accurate. On the other hand QFE can produce queries that are far more concise and comprehensible to humans and that are expressed in the widely understood and practically useful form of SQL queries. For many conceivable applications the latter criteria are of paramount importance; this argues for continued exploration of approaches like that taken with QFE.

Certainly the performance of QFE must be improved to support some kinds of applications, but we are confident that evolutionary search times can be reduced from hours to minutes through the use of modern hardware and straightforward software optimizations. This will enable many applications in which a human-comprehensible insight about the structure of a data set has substantial value.

Our current implementation of QFE has several limitations, but the developmental GP approach will make it easy to remove many of these. For example the current implementation does not allow for the evolution of queries that perform joins across multiple tables, create projections by selecting specific attributes in the SELECT clause, use SQL's GROUP BY or HAVING clauses, or use aggregate functions such as COUNT() or AVG(). But each of these capabilities could be provided simply by writing additional developmental instructions; no other changes would have to be made to Push program representations and no changes would have to be made to the evolutionary algorithm or to the fitness assessment procedures. Of course the addition of such capabilities would change the evolutionary space, and it is possible that some of these changes detract from, rather than enhance, the system's ability to find good queries. But the developmental framework makes it simple to add additional query components and to conduct runs to explore their effects.

QFE should have no problem with a database that has an extremely large number of tuples, as long as the example training set is not also extremely large. We have seen that QFE performs well on a problem in which the example set contained over 30,000 tuples in the 50k-Classification problem. If the example set were orders of magnitude larger, then QFE may take prohibitively long to run, particularly if individual queries take a long time. However, this problem could be ameliorated by using a high-performance distributed system, conducting multiple fitness tests in parallel and also submitting queries to a parallel database server. We performed our runs using a local SQLite

database because it was easiest to set up for our proof-of-principle runs, and a parallel database server would speed things up dramatically. A different limitation may stem from example sets that are too selective. For example, if a database has a limited number of positive examples, there may not be enough examples to accurately evolve a query that satisfies those examples in a general way. Nonetheless our work indicates that developmental GP has the potential to contribute to the discovery and exploitation of knowledge in databases in significant ways.

Acknowledgment

We thank Gerome Miklau for advice regarding databases and the UCI Machine Learning Repository for use of the adult dataset; see http://archive.ics.uci.edu/ml/index.html. This material is based upon work supported by the National Science Foundation under Grant No. 1017817. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Acar, Aybar C. and Motro, Amihai (2005). Intensional encapsulations of database subsets by genetic programming. Technical Report ISE-TR-05-01, Information and Software Engineering Department, The Volgenau School of Information Technology and Engineering, George Mason University.

Castro da Silva, Bruno and Thomas, Philip (2010). Automatic query generation. "Unpublished manuscript".

Doucette, John A., McIntyre, Andrew R., Lichodzijewski, Peter, and Heywood, Malcolm I. (2012). Symbiotic coevolutionary genetic programming: a benchmarking study under large attribute spaces. *Genetic Programming and Evolvable Machines*, 13(1):71–101. Special Section on Evolutionary Algorithms for Data Mining.

Frank, A. and Asuncion, A. (2010). UCI machine learning repository.

Freitas, Alex (2002). A survey of evolutionary algorithms for data mining and knowledge discovery. In Ghosh, A. and Tsutsui, S., editors, *Advances in Evolutionary Computation*, chapter 33, pages 819–845. Springer-Verlag.

Freitas, Alex A. (1997). A genetic programming framework for two data mining tasks: Classification and generalized rule induction. In Koza, John R. et al., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 96–101, Stanford University, CA, USA. Morgan Kaufmann.

Gruau, F. (1994). *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France.

Ishida, Celso Yoshikazu and Pozo, Aurora Trinidad Ramirez (2002). GP-SQL miner: SQL-grammar genetic programming in data mining. In Fogel, David B. et al., editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 1226–1231. IEEE Press.

Klein, Jon and Spector, Lee (2007). Unwitting distributed genetic programming via asynchronous JavaScript and XML. In Thierens, Dirk et al., editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1628–1635, London. ACM Press.

Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman.

Montana, David J. (1995). Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230.

Quinlan, J. Ross (1993). *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Spector, Lee (2001). Autoconstructive evolution: Push, pushGP, and pushpop. In Spector, Lee et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA. Morgan Kaufmann.

Spector, Lee and Klein, Jon (2005). Trivial geography in genetic programming. In Yu, Tina, Riolo, Rick L., and Worzel, Bill, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 8, pages 109–123. Springer, Ann Arbor.

Spector, Lee, Klein, Jon, and Keijzer, Maarten (2005). The push3 execution stack and the evolution of control. In Beyer, Hans-Georg et al., editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA. ACM Press.

Van Rijsbergen, C.J. (1979). *Information retrieval*. Butterworths, London.

Veeramachaneni, Kalyan, Vladislavleva, Ekaterina, and O'Reilly, Una-May (2012). Knowledge mining sensory evaluation data: genetic programming, statistical techniques, and swarm optimization. *Genetic Programming and Evolvable Machines*, 13(1):103–133. Special Section on Evolutionary Algorithms for Data Mining.