

# Intensional Encapsulations of Database Subsets via Genetic Programming

Aybar C. Acar and Amihai Motro

March 8, 2005

## Abstract

Finding intensional encapsulations of database subsets is the inverse of query evaluation. Whereas query evaluation transforms an intensional expression (the query) to its extension (a set of data values), intensional encapsulation assigns an intensional expression to a given set of data values. We describe a method for deriving intensional representations of subsets of records in large database tables. Our method is based on the paradigm of genetic programming. It is shown to achieve high accuracy and maintain compact expression size, while requiring cost that is acceptable to all applications, but those that require instantaneous results. Intensional encapsulation has a broad range of applications including cooperative answering, information integration, security and data mining.

## 1 Introduction

The problem of finding intensional encapsulations of database subsets has attracted considerable interest for almost two decades. Essentially, intensional encapsulation of data is the *inverse* of query evaluation. Whereas query evaluation substitutes an intensional expression (the query) with its extension (a set of data values), intensional encapsulation assigns an intensional expression to a given set of data values.

The original application of intensional encapsulation was in *cooperative answering systems*, proactive systems that help users achieve their retrieval goals efficiently. The common term for the method was *intensional answering*, and the idea was to respond to database queries with concise expressions that describe, as accurately as possible, the usual (extensional) answers to these queries. The user would thus receive two complementary responses: the usual answer, and a compact description of the answer. For example, a query about the employees who earn over \$80,000 would be answered extensionally by the appropriate set of employees, and intensionally by an expression such as “all the engineers, except John, but also Mary”.

This paper describes a novel method for generating intensional encapsulations using the paradigm of genetic programming. Given a set of database records, intensional expressions are generated that attempt to “cover” the given set. These expressions are evolved and recombined with each other until a satisfactory result is obtained. The attributes and attribute values used in these expressions are selected using a Bayesian approach that uses the probability distributions of the attribute values in the database.

The advantage of using genetic programming is that it does not require any semantic information about the content of the database, the meaning of its attributes, and so on. In comparison with other methods that are “blind” to semantics, genetic programming offers much more precision. Genetic programming is especially adept at finding intensional encapsulations for sets with small disjuncts (e.g., a set of records containing all engineers and also Mary), where more traditional approaches like decision trees tend to generalize and neglect the exceptions.

Any method for intensional encapsulation is subject to three performance measures. The first measure is *accuracy*. How well does the intensional expression obtained represent the given set. This is interpreted as the *similarity* of two sets: the given set, and the extension of the intensional expression. We adopt a common measure of set similarity, which is the harmonic mean of the relative containments of each set in the other set. Our experiments achieved mean accuracy of 94.6%. Clearly, compact intensional encapsulations are preferred as they are more comprehensible and more likely to be meaningful to the application. Our second performance measure is therefore *conciseness*. Our measure for conciseness is comparative: Our experiments begin with sets that are themselves generated by intensional expressions. We then compare the complexity of the discovered intension with the complexity of the *a priori* intension. In 90.4% of the experiments, conciseness has either remained the same or has actually improved. The third performance measure is *time*: How much effort is spent in obtaining acceptable encapsulations. Genetic programming processes are measured by “generations”, and the mean number of generations required was 2.33. Using a desktop computer of modest specifications, a 10 MB database required under 4 seconds on average; for a 100 MB database the average was around 60 seconds. We believe that these initial results prove the viability of our methods, particularly for classes of applications that do not require instantaneous responses.

Our methodology is described in Section 3. Section 4 details our experiments. Summary and conclusions are given in Section 5. We begin with a brief review of related work.

## 2 Background

The two main subject areas of this paper are genetic programming and intensional encapsulation. The authors are not aware of any previous work that combined these two areas. Hence this section briefly reviews each of these areas separately.

Genetic programming has developed over the last two decades as a local-optimization

method for generating simple computer programs that provide solutions for “black-box problems”; i.e., problems that seek to find the correct output for given input, without the need for a general algorithm. The extent to which a program can achieve the correct mapping of its input to the required output defines its *fitness* to the problem. In this process, initial programs are modified and combined with each other to generate “offspring programs”, in an attempt to find programs that achieve even higher fitness. This evolution process terminates when a program is obtained that performs above a certain threshold of fitness.

The basic principles of genetic programming were defined in a landmark book by Koza [7], which laid the foundations of genetic programming, including the tree generation algorithms, breeding operations and evolutionary cycle used in this paper. A second publication by this author [8] is also relevant to our work, in that it described mechanisms for using the optimal substructure properties of the problem and defined reusable program portions called *automatically defined functions (ADFs)*. The attribute selection operators, described in Section 3.2, are essentially ADFs, and our approach tries to exploit optimal substructures by starting from simple subqueries and building accurate and concise intensional expressions in a bottom-up fashion.

Genetic programming has been used to some extent in data mining. In some cases researchers have preferred genetic programming as a classification method in place of or along with more traditional methods such as decision trees [3]. The main motivation seems to be to increase the accuracy of prediction. Genetic algorithms have been claimed to capture trends and rules in smaller subsets of the dataset that conventional methods may overlook [4]. Genetic programming and algorithms have been successful enough in data mining to bring about complete commercial systems based on this paradigm [5]. A more complete treatment of genetic programming and genetic algorithms in data mining can be found in [6].

The problem of finding compact descriptions for database subsets received considerable attention in the last two decades. The problem is usually framed in the context of cooperative query systems [11], and the idea is to annotate answer sets provided by a database system with compact descriptions that provide additional insight and interpretation to these sets. Borrowed from logic, the terms *intension* and *extension* are used in the database literature to describe, respectively, the definition of a database predicate (e.g., a query, a view, or a constraint), and the population of database items that satisfy the predicate (the answer to a query, the materialization of a view, or the values conforming to the constraint). Hence, these compact descriptions have been termed *intensional answers*. The term implies, however, that the process is applicable only when the given set has been generated by an *a priori* query, whereas it is just as useful when the given set is entirely *ad hoc*. We therefore adopt the more general term *intensional encapsulations*.

This problem has been tackled in different database models, including relational databases, logic databases, and databases that utilize concept taxonomies. Some methods find encapsulations that are purely intensional, whereas other also incorporate extensional information into their encapsulations; some methods find encapsulations that characterize the extensions *perfectly* (i.e., the extensions of the discovered intensions are identical to the original

extensions), whereas other methods only find *applicable* characterizations (i.e., they only characterize subsets of the given extensions); by their nature, encapsulations of query extensions may need to be updated when the underlying database changes, yet some methods derive their intensional expressions from data-independent information (e.g., from database constraints), indeed thus providing equivalent formulations of the original queries. A survey of a large number of intensional answering methods may be found in [10]. Of particular relevance is [12], where two key quality aspects of intensional descriptions, accuracy and conciseness, are discussed, and their often conflicting nature is observed.

In addition to cooperative answering, intensional encapsulations have other practical uses. They are related to data mining in that it too seeks to derive an underlying explanation from a given collection of data. In the area of information integration, newly discovered information is assigned intensional descriptions, as part of a system that automatically incorporates new sources into a virtual database [2]. In the area of database security, intensional encapsulations may be used to analyze the set of records that have been delivered to a user over a period of time, to determine whether the user has surreptitious intents [1].

## 3 Methodology

### 3.1 Overview

We assume that the data is stored in a relational database, and we shall adopt the terminology and conventions of relational databases. We assume a single database table, denoted  $R$ , and a *target* set of records  $T \subseteq R$ ; i.e.,  $T$  is the set of records of  $R$  for which an encapsulated description is sought.

Our method creates an initial population of intensional expressions (indeed, they are *queries* against the table  $R$ ), and evolves this population until one or more of these queries performs satisfactorily (i.e., with acceptable accuracy). Let  $Q_i$  denote a query of this population, and let  $P_i \subseteq R$  denote its extension in  $R$ ; i.e.,  $Q_i(R) = P_i$ . The accuracy of  $Q_i$  is measured by the similarity of the sets  $P_i$  and  $T$ . Perfect accuracy is achieved when  $P_i = T$ .

The process begins with the generation of a set of random queries  $Q_i$ . These arbitrary queries are fully correct queries on the table  $R$  that are synthesized from primitives that have been previously adopted. The judicious selection of these primitives and their combination into queries will be discussed later.

Next, each query in the initial population is evaluated and a *fitness* value is computed. Fitness is defined as the similarity of the query's extension  $P_i$  and the target set  $T$ . The measure to be used will be discussed later, but for now we assume that it is a value between 0 and 1, where 0 denotes complete disjointness of the sets and 1 denotes their complete overlap.

The queries with the highest fitness are then bred with each other using either direct

combination, crossover or mutation, to generate a new population of queries. This new generation is then evaluated in the same manner as its predecessor, and the evaluation is used to produce yet a newer generation of queries. Typically, this evolutionary process gradually increases the mean fitness of its population. The process is halted once a member of the current population achieves satisfactory fitness. Satisfactory fitness is defined with a similarity threshold. This threshold may be fixed, or it may depend on the sizes of  $R$  and  $T$ . This iterative process is illustrated in Figure 1. As there is no guarantee that the process will converge, it will in practice be stopped if fitness does not improve within a given number of generations. In such cases the process will only be able to derive intensional encapsulations of limited accuracy.

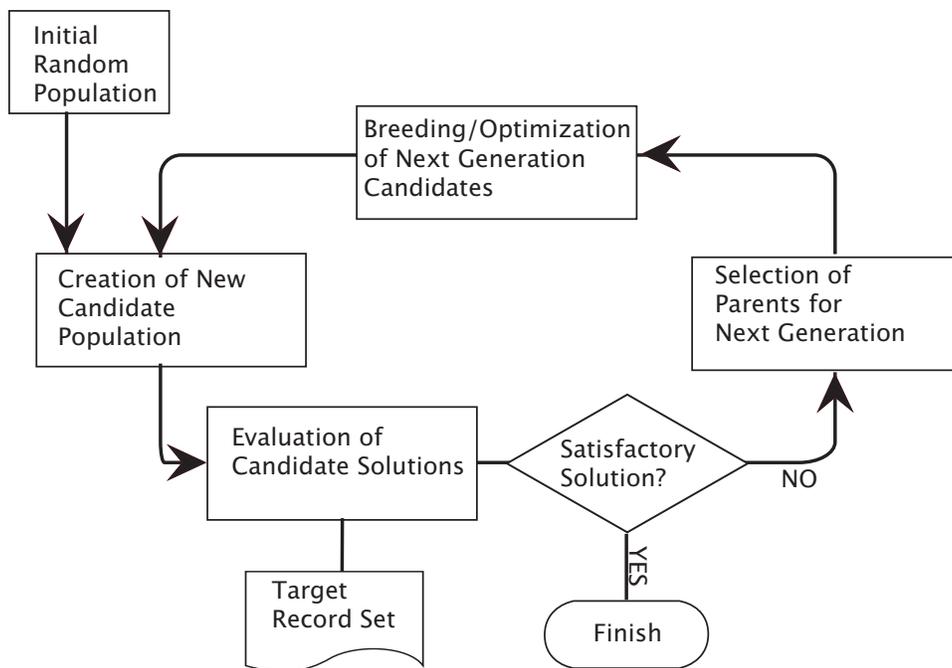


Figure 1: Overview of the Method

## 3.2 The Population

As common in genetic programming, the individuals of the population are represented as trees. Since in this application the individuals are queries, the trees correspond to standard relational algebra expressions. In these trees, leaf nodes correspond to selection operators and internal nodes correspond to set operations.

There are four operations in total:

1.  $Selection(R, \alpha) = \{x \mid x \in R \wedge \alpha(x)\}$
2.  $Union(R_1, R_2) = \{x \mid x \in R_1 \vee x \in R_2\}$

$$3. \text{ Intersection}(R_1, R_2) = \{x \mid x \in R_1 \wedge x \in R_2\}$$

$$4. \text{ Difference}(R_1, R_2) = \{x \mid x \in R_1 \wedge x \notin R_2\}$$

$R$ ,  $R_1$  and  $R_2$  are sets of records, and  $\alpha$  is a predicate formula that evaluates to *true* or *false* for each record  $x$ . In general,  $\alpha$  compares a particular position (*attribute*) in the record  $x$  to a prespecified value. The particular type of comparison that can be used depends on the *type* of the attribute. We distinguish three types of database attributes:

1. **Nominal:** Attributes whose values are (short) character strings. Ordinal relationships among the values are not assumed. Examples include names, telephone numbers, model numbers.
2. **Numerical:** Attributes whose values are numbers. Examples include distances, weights, times, amounts of money.
3. **Textual:** Attributes whose values are free text (long character strings). Examples includes comments, descriptions, exceptions.

We assume that each attribute of the given table is associated with one of these three types. When a selection operation needs to be introduced, its specific formula  $\alpha$  will depend on the type of the selection attribute. Correspondingly, we define three types of selection formulas  $\alpha$ :

1. **Equality:**  $A = a$
2. **Range:**  $a_1 \leq A < a_2$
3. **Substring:**  $A = '*a*'$

In each formula,  $A$  indicates an attribute of the table, and  $a$ ,  $a_1$  and  $a_2$  indicate a value of that attribute. In other words, for nominal attributes, we consider selection formulas that are equality comparisons, for numerical attributes we consider range comparisons, and for text attributes we consider substring comparisons.

The creation of a new selection operation (either in the generation of the initial population, or in subsequent evolutionary cycles) involves two separate decisions: First, an attribute  $A$  is chosen, then appropriate limiting values (either a single value  $a$  or a pair of values  $a_1$  and  $a_2$ ) are adopted. The choice of  $A$  assumes simply that the attributes of  $R$  are distributed uniformly (i.e., all the attributes have equal probability of being chosen). Once  $A$  has been chosen, the limiting values are adopted in accordance with the type of  $A$ . The choice of limiting values is important, as judicious choices will promote good initial fitness and will increase the chance of early convergence. We begin by describing the choice of the limiting value  $a$  when the attribute  $A$  is nominal.

For each attribute  $A$  of  $R$  we define a random variable  $X_A$  whose range is the domain of  $A$ .<sup>1</sup> The limiting values are chosen from the domain of  $A$  according to the distribution of  $X_A$ . Intuitively, if a value  $v$  occurs frequently in the attribute  $A$  in the target set  $T$ , then it would be wise to begin with a selection  $A = v$ , as it may be expected that the records thus selected will have a good fit with  $T$ . The distribution of  $X_A$  is defined with a Bayesian approach. We observe that

$$p(r \in T \mid A = v) = \frac{|\sigma_{A=v}(T)|}{|\sigma_{A=v}(R)|}$$

Namely, the probability that an arbitrary record  $r$  is in  $T$ , given the property that its attribute  $A$  has the value  $v$ , is the proportion of records with this property that are included in  $T$ . This value is normalized to define the probability distribution of  $X_A$ :

$$p(X_A = v) = \frac{P(T \mid v)}{\sum_k P(T \mid V_k)}$$

We handle numerical and textual attributes in a manner similar to nominal attributes, by reducing these types to nominal attributes.

Assume first that  $A$  is a textual attribute; i.e., each value  $v$  of  $A$  is a text string. Initially, we convert all the strings in the domain of  $A$  to a set of *tokens*. A token is defined as a substring delimited by whitespace or punctuation. For each token, we calculate the number of  $R$  records and the number of  $T$  records in which the token appears. These numbers are used to define the probability of a record being in  $T$ , given that it contains a given token. These values are then normalized to a probability distribution for the tokens of  $A$ . When a textual selection  $A = '*a*'$  is required,  $a$  is chosen in accordance with this distribution.

When  $A$  is a numerical attribute, we partition its entire range to a set of *intervals*. Intervals are defined so that the occurrences they encompass have sufficiently small variations (i.e., the variance is below a predefined threshold). For each interval, we calculate the number of  $R$  records and the number of  $T$  records in which values from the interval appear. These numbers are used to define the probability of a record being in  $T$ , given that it contains a value from one of the intervals. These values are then normalized into a probability distribution for the intervals of  $A$ . When a numerical selection  $a_1 \leq A < a_2$  is required, one of the intervals is chosen in accordance with this distribution.

It is worth pointing out that the tokens and intervals are defined only once and do not need to be updated unless  $R$  changes.

### 3.3 Evaluation of Individuals

Once a population is created, each individual must be evaluated to determine its fitness. As already indicated, the fitness of an individual  $Q_i$  is the similarity of its extension  $Q_i(R) = P_i$

---

<sup>1</sup>If the domain of  $A$  is not available, we use the *active* domain; i.e., the set of values of attribute  $A$  in the present instance of  $R$ .

to the target set  $T$ .

To define set similarity, we note that the identity of two sets  $A$  and  $B$  is defined  $A = B$  if and only if  $A \subseteq B$  and  $B \subseteq A$ . Consequently, a reasonable approach to set similarity is to measure the *extent* to which each set is contained in the other. The extent to which  $A$  is contained in  $B$  may be taken as the fraction  $\frac{|A \cap B|}{|B|}$ . Similarly, the extent to which  $B$  is contained in  $A$  is  $\frac{|B \cap A|}{|A|}$ . These fractions range between 0 (total disjointedness) and 1 (total containment). When both fractions are 1, the sets are identical; when either one is 0 (the other is then 0 as well), the sets are disjoint.<sup>2</sup>

For the purpose of fitness, these two measures must be combined into one. Note that when  $P_i$  is compared with  $T$ ,  $\frac{|P_i \cap T|}{|T|}$  measures the extent to which the generated expression covers the given set (i.e., its ability to avoid “false negatives”). Similarly,  $\frac{|P_i \cap T|}{|P_i|}$  measures the extent to which the generated expression is covered by the given set (i.e., its ability to avoid “false positives”). As we have no preference of one error type over the other, we shall fuse the two measures symmetrically. A well-accepted symmetric fusion of these two measures is their *harmonic mean*. The harmonic mean of two numbers  $x_1$  and  $x_2$  is  $2\frac{x_1 \cdot x_2}{x_1 + x_2}$ . Substituting  $\frac{|P_i \cap T|}{|T|}$  and  $\frac{|P_i \cap T|}{|P_i|}$  for  $x_1$  and  $x_2$ , our measure of fitness is:

$$2\frac{|P_i \cap T|}{|P_i| + |T|}$$

This measure preserves the properties that (1) it is between 0 and 1, (2) it is 0 if and only if the two sets are disjoint, and (3) it is 1 if and only if the two sets are identical. Since the target set  $T$  is assumed non-empty, it is well-defined.

### 3.4 Selection of Parents

In genetic programming the two most common methods of selecting individuals with higher fitness are *fitness-proportional* selection and *tournament* selection [7]. In the former, individuals are selected with probability proportional to their fitnesses. In the latter, groups of 3 or 7 individuals are randomly selected and the individual with the highest fitness in each group is selected. In either case, the selection process repeats a number of times equal to the population size to obtain the parents for the new generation. Notice that several copies of the individuals with highest fitness are likely to be added to the parent pool whereas the lowest ranking individuals will have a lower chance at breeding.

Of these two methods we shall use tournament selection with groups of 7. The choice in this case is neither easy nor absolute. However, the fact that tournament selection is easier to implement and easier to parallelize is a major advantage. Also, as with all local search methods, genetic programming tends to converge as it proceeds. This is seen as diminished variation between individuals in the later generations. In such situations, tournament selection is more likely to select for breeding the individuals with the highest fitness values.

---

<sup>2</sup>In information retrieval these measures are known as precision and recall.

### 3.5 Breeding

Once a pool of parent individuals has been selected, the next generation of individuals is created using three operations: direct combination, mutation and crossover. Each operation requires two individuals and in turn creates two new children. Brief descriptions of these operations follow, and Figure 2 illustrates them with examples. The next generation does not necessarily comprise new individuals only. A random portion of the parents may be injected into the new mix without any alteration. In our experiments, 10% of the individuals of each generation were comprised of unmodified parents from the preceding generation. The remaining 90% were modified with these three operations, selected at random.

**Mutation.** Two individuals from the parent pool are selected and modified separately. Recall that each individual is a tree with leaf nodes corresponding to selections and internal nodes corresponding to set operations. In each of the two individuals, a node is selected at random and the tree is pruned at that node, and a newly generated selection is added at to that node. The resulting individuals are added to the new generation.

**Crossover.** Two individuals from the parent pool are selected, and in each, a node is selected at random and the tree is pruned at that node. The removed subtrees are switched: the subtree removed from one parent is grafted to the other tree at the point where it was pruned. The resulting individuals are added to the new generation.

**Direct Combination.** Two individuals from the parent pool are selected and replicated, obtaining two identical pairs. Each pair is connected with a new root node. The operations to be associated with the new roots are selected at random from the three set operations (union, intersection and difference). The resulting individuals are added to the new generation.

We mentioned in the introduction the benefit of concise intensional encapsulations. Conciseness is not an intrinsic consideration in our methodology; namely, our genetic programming process does not include conciseness in its measure of fitness. Of the three breeding operations, mutation tends to create shorter expressions, direct combination tends to create longer expressions, and crossover tends to keep lengths unchanged. Overall, however, since the increase due to direct combination is on the average larger than the reduction due to mutation, the complexity of expressions tends to increase with generations. In an effort to control this increase, we begin with concise individuals; indeed, the initial generation comprises individuals that are single node each (simple selections).

### 3.6 Termination and Optimization

Once the new generation has been generated, the cycle is repeated as shown in Figure 1. The process terminates when “the best of the new generation” exceeds a prespecified threshold fitness, or a prespecified number of generations pass without an improvement in the best fitness attained. In either case, the individual with the best fitness is adopted.

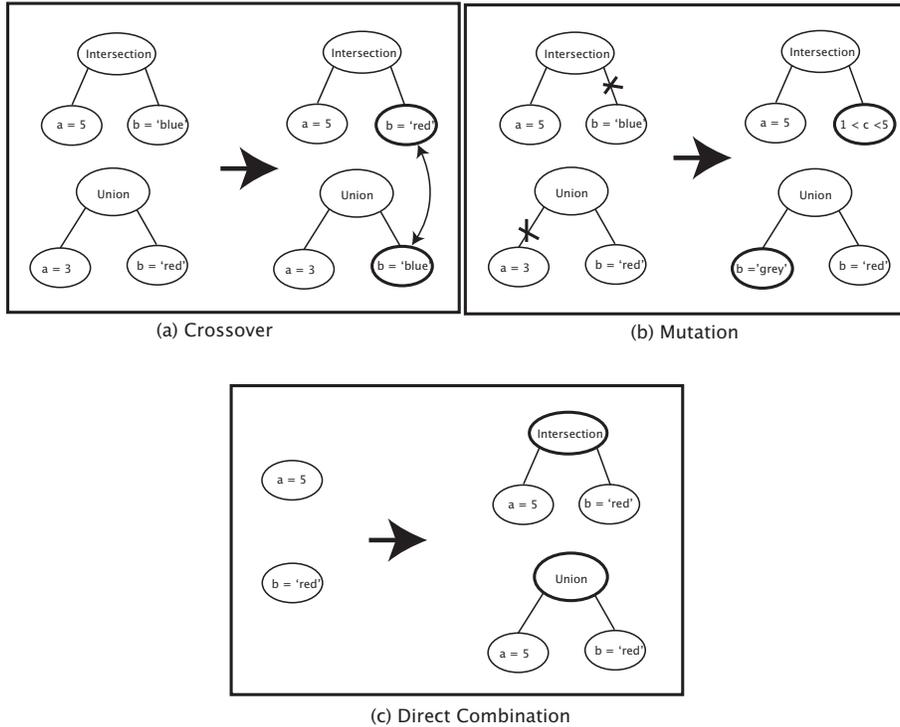


Figure 2: Recombination Methods

As we shall observe in the next section, our search process is dominated by the time required for database access. To alleviate this problem, we *cache* intermediate database results. Recall that the leaves of each query tree are selection operators. These are executed in the database, and the results are stored in memory as efficient bit vectors. Thereafter, all subsequent set operations in a particular tree (all the way to its root) are done in memory, thus avoiding any additional database access. This optimization resulted in substantial improvement.

## 4 Experimentation

### 4.1 Setup

Our methodology was implemented as a prototype using Common Lisp and PostgreSQL for database management. The system was tested on a variant of the TPC-H [13] benchmark database. The original TPC-H database is a synthetic 9-relation database of the inventory control and sales of a manufacturing operation. Since our methodology assumes a single relation, 7 of the 9 relations were joined over their foreign keys (the remaining two relations were considered somewhat redundant). The resulting join was projected to remove attributes that

are not useful for intensional encapsulations (such as employee IDs or telephone numbers). The resulting relation had 24 attributes, 12 of which were nominal, 9 were numerical and 3 were textual. The TPC-H benchmark can generate databases of arbitrary size. Relations of size 10 MB and 100 MB were synthesized to study the scaling of our methodology.

The target sets of records (the sets of records for which encapsulations were sought) were generated by means of queries. Each query was composed by selecting random attributes and random values for those attributes assuming equal probabilities. The query trees were generated with the Probabilistic Tree Creation (PTC2) algorithm [9]. The PTC2 algorithm can generate random queries of precisely defined size. One of the factors examined in the experiments was the effect of the complexity of the generating query on the accuracy of the encapsulation, and queries ranging from single selection predicates to 5-selection predicates were used. These queries were evaluated on the database, and the extensions retrieved were given to our system as targets. Throughout the experiment, the size of the population was kept at 40. This size was determined after some experimentation. A lower population size of 20 required far too many generations to converge; a higher population size of 60 required more time per generation, without giving substantially better results. Table 1 shows the number of experiments performed for each factor. Altogether, the experiment was repeated 2,000 times. Termination was controlled by setting the fitness threshold to 0.99, and the maximal number of generations to 3.

DB Size (MB)	Population Size	Number of Targets				
		Query Complexity				
		1s	2s	3s	4s	5s
10	40	200	200	200	200	200
100	40	200	200	200	200	200

Table 1: Breakdown of Experiments

As explained, the target record sets were generated by random queries. Another acceptable approach would have been to generate random record sets directly. There are three reasons for our choice. First, in many applications (e.g, cooperative answering, or security), the target sets are indeed generated by queries. Second, as we shall see, these *a priori* intensional expressions are used in the evaluation of the effectiveness of the system with respect to conciseness. Finally, using *a priori* intensional expressions guarantees the existence of at least one encapsulation with perfect accuracy for each given target set (i.e., with targets that are random sets of tuples, when the system returns an intensional encapsulation of imperfect accuracy, it is impossible to tell if this is because the system is imperfect or because there simply is no perfect intensional encapsulation of the given target set).

## 4.2 Accuracy

The first measure of performance that we consider is accuracy. Accuracy measures the similarity of the given target set and the extension of the encapsulation obtained at the end

of the search process; i.e., it is the fitness of the final result. Recall that fitness values are between 0 and 1.

The mean accuracy achieved in the entire set of experiments was 94.6%. Moreover, in over half of the experiments the system found the perfect encapsulation. The complexity of the query that generated the target set seems to have a significant effect on the success of the system. More complex targets resulted in a noticeable decrease in the accuracy of the encapsulations. The mean accuracies for targets of different complexity is shown in Table 2; the individual distributions of accuracy are shown in Figure 3.

Complexity (Selections)	Mean Accuracy
1	0.998
2	0.969
3	0.950
4	0.928
5	0.877

Table 2: Accuracy by Target Complexity

### 4.3 Conciseness

Our measure for conciseness of an intensional expression is the number of nodes in the tree that represents it. For example, an expression with a single selection has conciseness measure 1, and an expression which is the difference of two subexpressions, each constituting a conjunction of two selections, has conciseness measure 7. Determining the effectiveness of the system with respect to conciseness is not straightforward. Our approach has been to *compare* the conciseness of the discovered encapsulation to that of the generating intensional expression (which, of course, is unknown to the system).

Overall, in 90.4% of the experiments the system generated intensional encapsulations that were as least as concise as the *a priori* expressions (in 73.1% of the experiments conciseness remained the same, in 17.3% it actually improved). Only in 9.6% of the experiments, the discovered encapsulations were less concise. Like accuracy, conciseness too declined as the complexity of targets increased. Table 3 breaks down the comparative conciseness rates according to the complexity of the targets.

### 4.4 Time

The third measure of performance that we consider is time. Predictably, the primary bottleneck for the system is database management; in particular, the disk input and output activity.

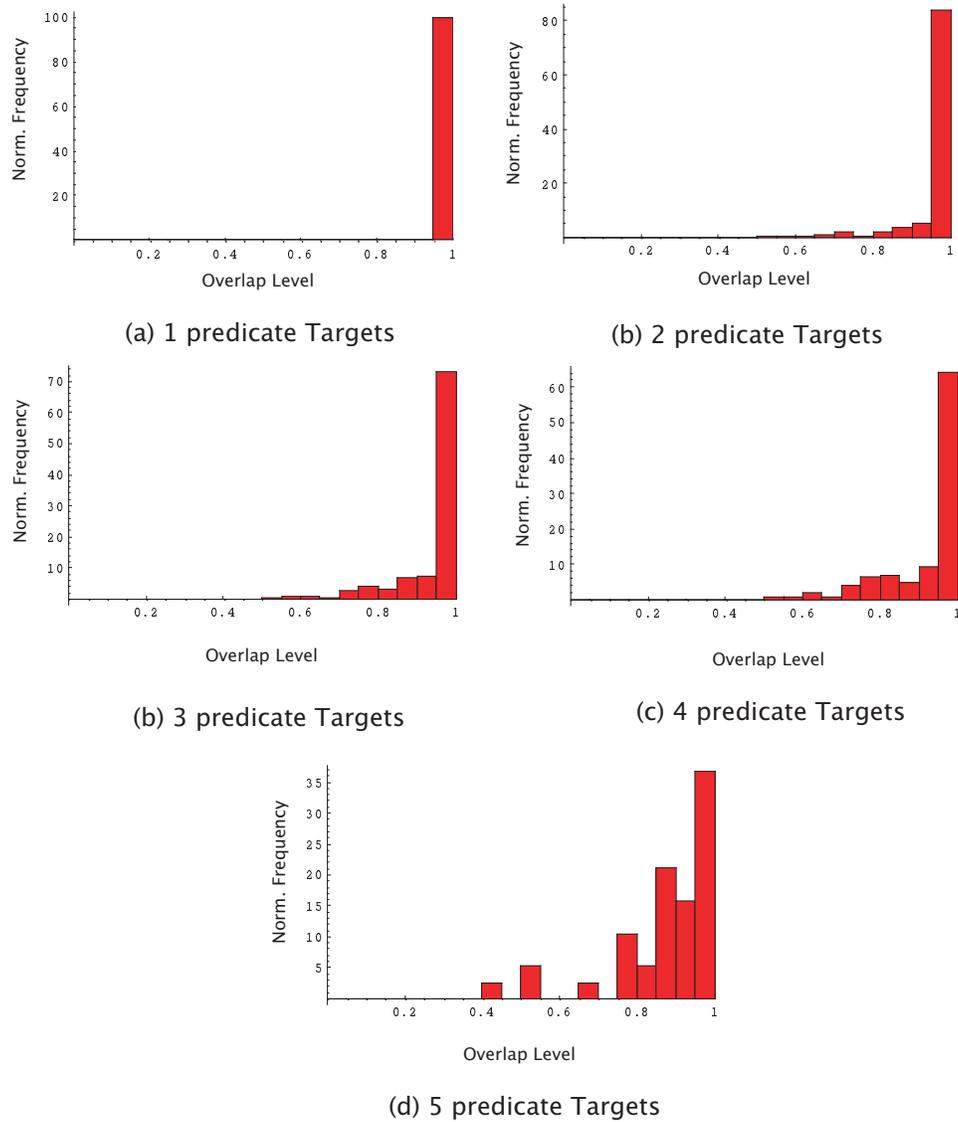


Figure 3: The Distribution of Accuracy for each Target Complexity

Complexity (Selections)	% Less Concise	% Equally Concise	% More Concise
1	1.21	98.79	N/A
2	2.34	84.21	13.45
3	9.03	78.61	12.36
4	17.84	56.48	25.68
5	17.56	47.52	34.92

Table 3: Conciseness by Target Complexity

Figure 4 plots the number of blocks retrieved by the DBMS during each experiment *vs.* the overall duration of that experiment. The relationship is definitely linear. Moreover, since we know that the size of each disk block is 4 KB, the slope of the line is 23.7 MB/s, which roughly corresponds to the data transfer rate of the commodity PC used for the experiment.<sup>3</sup>

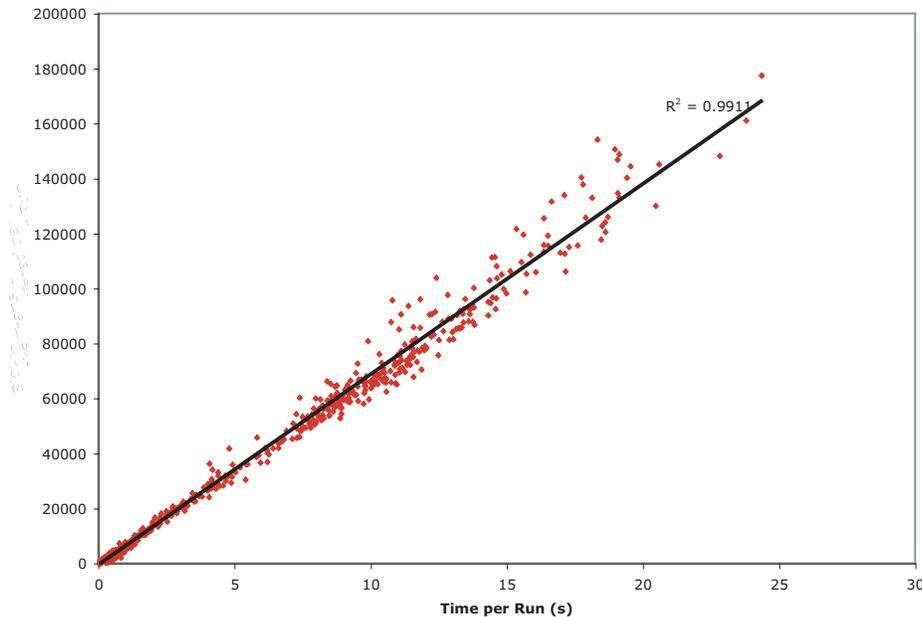


Figure 4: I/O Activity *vs.* Time

As discussed earlier, genetic programming processes are measured by the number of generations required for the process to converge. Figure 5 shows the distribution of the number of generations in our experiments. The mean number of generations required was 2.339. This information, along with the time required per generation, gives the average time required for an experiment. The average time for the 10 MB database was 3.76 seconds, and the average for the 100 MB database was 61.32 seconds. The relationship of the average experiment time with respect to number of generations is shown in Figure 6.

Note that processing time as a function of number of generations is actually a weak quadratic,<sup>4</sup> where one would expect the relationship to be linear. However, as already mentioned, the individuals grow larger as generations advance. Therefore, although the number of individuals processed in each generation is constant, processing individuals in later generations requires more time. Theoretically, the square term would start to dominate after about 20 generations. However, since in practice we never get that far, the behavior

<sup>3</sup>The computer used a 1.8 GHz AthlonXP CPU with 256 MB RAM running FreeBSD on an 80 GB IDE disk drive with Reiser File System.

<sup>4</sup>The coefficient of the square term was 0.025 for the 10 MB database and 0.767 for the 100 MB database.

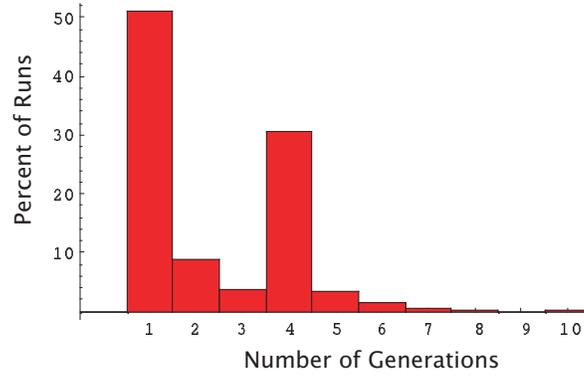


Figure 5: Distribution of Number of Required Generations

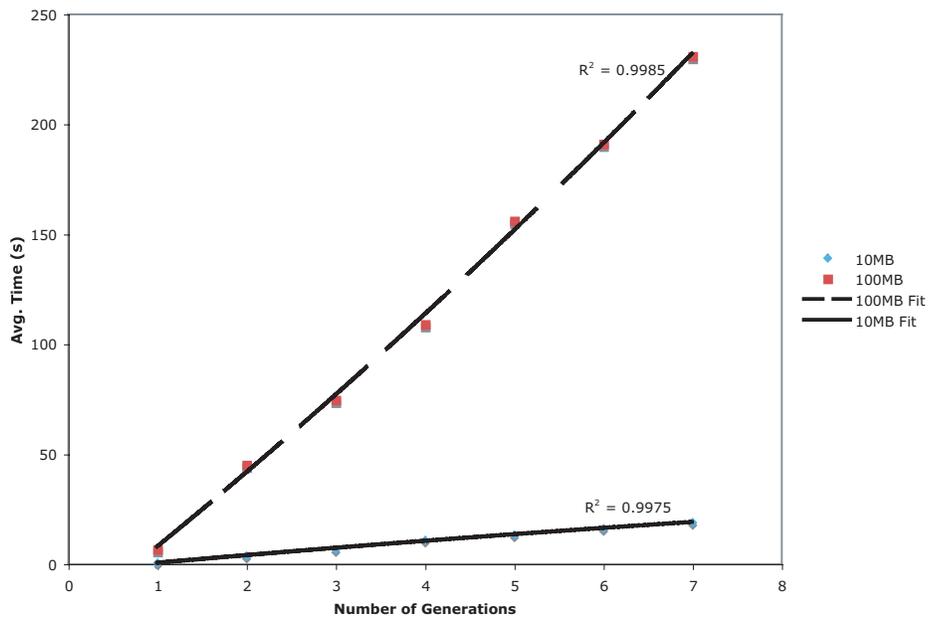


Figure 6: Time Requirement *vs.* Number of Generations

may be regarded linear. It should be noted that the primary reason for weakness of the square term is the caching of intermediate results as explained in Section 3.6. This simple optimization removes a significant load off the DBMS, especially for larger queries.

## 5 Conclusion

We described a novel approach to the well-known problem of finding intensional encapsulations of database subsets, based on the paradigm of genetic programming. In experiments, our method performed very well *qualitatively*; i.e., with respect to both accuracy and conciseness. Its *time* performance may be considered acceptable for all applications that do not require instantaneous results (its performance for moderate size databases may be considered acceptable even for real-time applications).

Indeed, the time performance achieved may be considered surprisingly good, given the general opinion of genetic programming as a solution method of low efficiency. This accomplishment is largely due to the fact that the programs we handle, namely queries, are very specialized, with restricted contexts and relatively small search spaces. In addition, our judicious choice of selection predicates promoted more rapid convergence of the evolutionary process. Furthermore, by caching solutions in memory, we were able to reduce database access substantially and thus contain the time required for each generation. Combined, the reduction in the number of generations required and efficient processing of each generation, resulted in this reasonably good performance. We observe that our method lends itself to parallelization, and we estimate that by increasing the database power (e.g., by using a cluster of database servers), further and considerable improvements may be achieved.

## References

- [1] Acar, A. C. and A. Motro. Why is this User Asking so Many Questions? Explaining Sequences of Queries. In *Proceedings of the 18th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 159–176, Kluwer, 2004.
- [2] Berlin, J. and A. Motro. Autoplex: Automated Discovery of Contents for Virtual Databases. In *Proceedings of COOPIS-01, Sixth IFCIS International Conference on Cooperative Information Systems, Lecture Notes in Computer Science No. 2172*, pages 108–122, Springer, 2001.
- [3] Carvalho, D. R. and, A. A. Freitas. A Hybrid Decision Tree/Genetic Algorithm for Coping with the Problem of Small Disjuncts in Data Mining. In *Proceedings of the 2000 Genetic and Evolutionary Computation Conference*, pages 1061–1068, Morgan Kaufmann, 2000.
- [4] Carvalho, D. R. and A. A. Freitas. A Genetic Algorithm-based Solution for the Problem of Small Disjuncts. In *Proceedings of PKDD 2000, the 4th European Conference*

- on Principles of Data Mining and Knowledge Discovery, Lecture Notes in Artificial Intelligence No.1910*, pages 345–352, Springer, 2000.
- [5] Flockhart, I. W. and N. J. Radcliffe. GA-MINER: Parallel Data Mining with Hierarchical Genetic Algorithms — Final Report. *EPCC-AIKMS-GA-MINER Report 1.0*, University of Edinburgh, 1995.
  - [6] Freitas, A. A. A Survey of Evolutionary Algorithms for Data Mining and Knowledge Discovery. In *Advances in Evolutionary Computing: Theory and Applications*, pages 819-845, Springer, 2003.
  - [7] Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
  - [8] Koza, J. R. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.
  - [9] Luke, S. Two Fast Tree-creation Algorithms for Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, IEEE, 2000.
  - [10] Motro, A. Intensional Answers to Database Queries. *IEEE Transactions on Knowledge and Data Engineering*, 6(3):444-454, IEEE, 1994.
  - [11] Motro, A. Cooperative Database Systems. *International Journal of Intelligent Systems*, 11(10):717–732, Wiley, 1996.
  - [12] Shum, C. D. and R. Muntz. Implicit Representation for Extensional Answers. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 497–522, Benjamin Cummings, 1988.
  - [13] Transaction Processing Performance Council. TPC Benchmark H Rev. 2.1.0. *Technical Report*, TPC, 2002.