# Ranking Function Discovery by Genetic Programming for Robust Retrieval

Li Wang[1], Weiguo Fan, Rui Yang, Wensi Xi, Ming Luo, Ye Zhou, Edward A. Fox
Department of Computer Science
Virginia Polytechnic Institute and State University
{liwang5, wfan, yrui, xwensi, lming, yezhou, fox}@vt.edu

## Abstract

Ranking functions are instrumental for the success of an information retrieval (search engine) system. However nearly all existing ranking functions are manually designed based on experience, observations and probabilistic theories. This paper tested a novel ranking function discovery technique proposed in [Fan 2003a, Fan2003b] – **ARRANGER** (**A**utomatic gene**R**ation of **RAN**king functions by **GE**netic p**R**ogramming), which uses Genetic Programming (GP) to automatically learn the "best" ranking function, for the robust retrieval task. Ranking function discovery is essentially an optimization problem. As the search space here *is not a coordinate system*, most of the traditional optimization algorithms could not work. However, this ranking discovery problem could be easily tackled by ARRANGER. In our evaluations on 150 queries from the ad-hoc track of TREC 6, 7, and 8, the performance of our system (in average precision) was improved by nearly 16%, after replacing Okapi BM25 function with a function automatically discovered by ARRANGER. By applying pseudo-relevance feedback and ranking fusion on newly discovered functions, we improved the retrieval performance by up to 30%. The results of our experiments showed that our ranking function discovery technique – ARRANGER – is very effective in discovering high-performing ranking functions.

## 1. Introduction

Text resources in digital format are quickly increasing with the rapid development of the IT industry. This tremendous collection of resources serves as a rich repository for our society in general. However, it also brings challenges to the general public. How to use this repository effectively is one of the biggest challenges. Researchers have developed various information retrieval systems, also known as search engines, to help people quickly and accurately find what they need from this repository. The working process for an information retrieval (search engine) system can be simplified to the process of returning an ordered document list according to a user's information need (expressed as queries). Therefore the most critical part for an IR system is its ranking function, which is used to order documents based on their similarity degrees to a user query. Designing a good ranking function, however, is not an easy task. There are many well-known ranking functions, such as Okapi BM25, TFIDF, and INQUERY. But most of those ranking functions are manually designed by experts based on heuristics, experience, observations, and statistical theories. One novel part of our work is that we use a Genetic Programming (GP) based technique called ARRANGER (**A**utomatic gene**R**ation of **RAN**king functions by **GE**netic p**R**ogramming) to discover ranking functions automatically [Fan 2003a, Fan2003b]. Ranking functions usually could not work consistently well under all situations. Various information retrieval studies have shown that the performance of a ranking function is very context-dependent [Salton & Buckley, 1988; Zobel & Moffat, 1998]. The context may depend on text collections or even properties of queries. Using a static ranking function can not guarantee good performance under all situations. How to find the "optimal" ranking function for a specific context is quite a challenge. The advantage of ARRANGER is that it can learn the "optimal" ranking functions according to different contexts by effectively combining multiple types of evidence in an automatic and systematic way. Using 150 queries from the ad-hoc task of the Robust Track in TREC 6, 7, and 8, we found ranking functions discovered with ARRANGER

[1] Li Wang is now at the University of Michigan (wang@umich.edu).

improved the performance of our baseline system, which uses the Okapi BM25 ranking function, by 8 ~ 16%. Based on those newly discovered ranking functions, we also tried other performance improvement techniques such as pseudo-feedback (query expansion) and information fusion which combines scores from different ranking functions using a regression technique. These techniques altogether help improve our performance by up to 30% in average precision over the baseline Okapi system.

Our paper is organized as follows. Section 2 states our research objectives. Section 3 describes basic data processing steps. Section 4 reviews ARRANGER – a GP-based ranking function discovery technique. Section 5 summarizes other techniques used in our system and gives a detailed description of our final submissions. Section 6 shows the official submission results in comparison with the other TREC teams. We conclude our paper in Section 7.

## 2. Research objectives

We have two objectives in this year's Robust Track:

1) We want to test the ARRANGER framework proposed in [Fan 2003a, Fan 2003b] to see whether it can work well on more heterogeneous collections.
2) We want to test whether the newly discovered ranking functions can work well with other performance improvement techniques such as query expansion through blind feedback, and ranking fusion using logistic regression.

## 3. Data processing

All our experiments were run on a two-2.3GHz processor Dell Server running the Linux operating system. Since our concentration in TREC is to test our GP-based ranking function discovery technique, ARRANGER, we didn't take advantage of the document structure. Past TREC results also showed that structure information didn't help in these data. In the parsing process, we simply removed the non-informative content in the collection and kept only the texts in the TEXT field. These texts were indexed into both forward index and inverted index formats for our experimental purposes after removing stop words and stemming. No phrases were used in our experiments.

For query processing, we indexed three different versions of the topic descriptions. The first version is description queries, which are generated based on the Description field only as required by the Robust Track. The second, short queries, are based on the Title and Description fields. The third, long queries, are extracted based on all fields from the topic description.

## 4. Ranking function discovery based on Genetic Programming

### 4.1 Genetic Programming

Genetic Programming (GP), an extension of Genetic Algorithms (GA), is an artificial intelligence technique, inspired by Darwin's theory of evolution. "Computer programs that evolve in ways that resemble natural selection can solve complex problems even their creators do not fully understand" [Holland, 1975]. Genetic Programming has been widely used and approved to be effective in solving optimization problems, such as financial forecasting, engineering design, data mining, and operations management. GP makes it possible to solve complex problems for which conventional methods can not find an answer easily.

In Genetic Programming, a large number of individuals, called a population, are maintained at each generation. An individual represents a tentative solution for the target problem. All these solutions form a space, say, $\Sigma$. In reality, individuals could be stored using complex data structures, such as a tree, a

linked list, or a stack. A tree is the most popular form to store and represent individuals. Figure 1a shows an example of a tree which represents the expression (X + Y)*Z. Now, as our target in TREC is to find an "optimal" ranking function to sort documents in the collection, individuals should represent tentative ranking functions. Figure 1b shows an individual representing a ranking function. A fitness function (f $(\cdot)$: $\Sigma \rightarrow$ R) is also needed in Genetic Programming. A fitness function takes the solution space, $\Sigma$, as its domain and returns a real number. Hence tentative solutions, represented by individuals, could be measured and ordered according to their return values. The return value of a fitness function must appropriately measure how well an individual, which represents a solution, can solve the target problem.

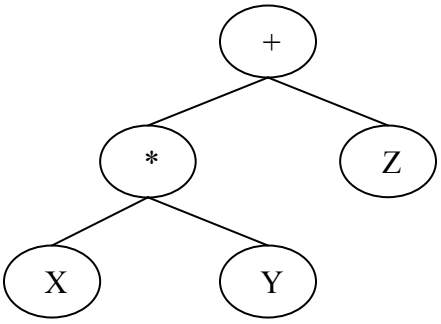


Figure1a. A simple expression represented by a tree

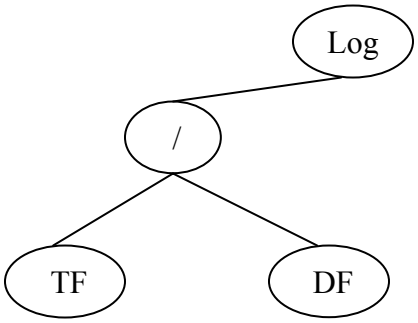


Figure 1b. A simple ranking function

Genetic Programming searches for the "optimal" solution by evolving the population generation after generation. Individuals in the new generation are produced based on those in the current one. Three genetic operators are usually used to produce the new generation. They are Reproduction, Crossover, and Mutation. The reproduction operator directly copies or, in a more appropriate term, clones some individuals into the next generation. The probability for an individual to be selected for Reproduction should be proportional to its fitness. Therefore the better a solution solves the problem, the higher probability it has to enter the next generation. While Reproduction keeps the best individuals in the population, Crossover and Mutation introduce transformation and so provide variations to enter into the new generation. The crossover operator randomly picks two groups of individuals, selects the best individual in each of two groups as parent according to their fitness, exchanges a randomly selected gene fragment of each parent and produces two "children". Thus, a "child" may obtain the good fragments of its excellent parents and may exceed them further, providing a better solution to the problem. Since parents are selected from a "competition", good individuals are more likely to be used to generate offspring. The mutation operator randomly changes a gene code, which could be a function or a parameter in our ranking function discovery task, of an individual. Figure 2 shows how the Crossover operator works. Using these genetic operators, a new generation is produced. The new generation keeps individuals with the best fitness in the last generation and takes in more "fresher air", providing creative solutions to the target problem. Better solutions are obtained either by inheriting and reorganizing old ones or by lucky mutation, simulating Darwinian Evolution. As we can see, Genetic Programming takes a so-called *stochastic search* approach, intelligently, extensively, and "randomly" searching for the optimal point in the entire solution space. It is less likely to be trapped in the local optima, which is the major problem of many other search algorithms. It provides sound solutions to many arduous problems, for which people have not found a theoretical or practical breakthrough.

### 4.2 Motivation for using Genetic Programming in ranking function discovery

A ranking function plays an essential role in an IR system (or search engine). It evaluates the similarity degree of a document to the query, so documents can be ranked according to its returned value. However, many empirical studies have shown inconsistent performance by existed well-known ranking functions on various collections [Salton & Buckley, 1988; Zobel & Moffat, 1998]. The same ranking function may work well on one collection, but poorly on others. They are collection-sensitive, and sometimes even query-sensitive. Given a specific context, how to select the right one from available ranking functions or how to design a new function for a given context has not been fully studied before.

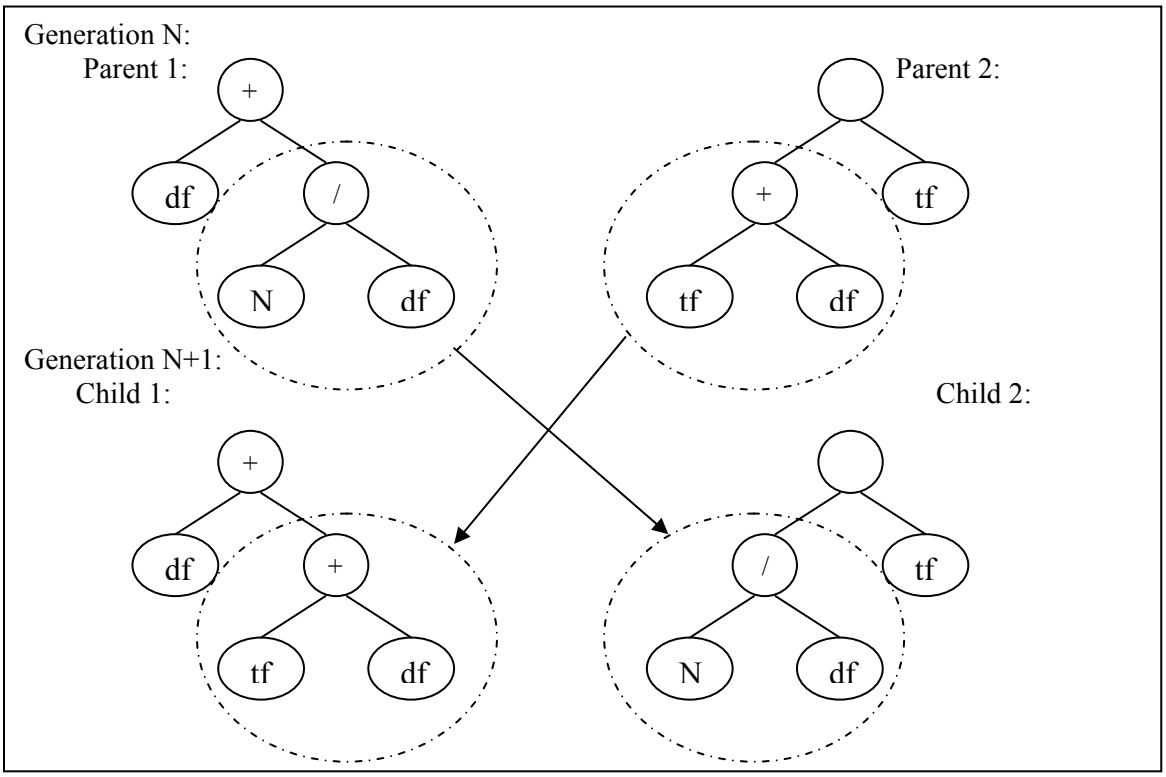


Figure 2. A simple example shows how the Crossover operator works in our ranking function discovery task.

Nearly all the existed ranking functions are manually designed, based on experience, heuristics and probability theory. Some parameters in these functions are usually adjusted to accommodate collection differences. However, these functions should still be categorized as static ranking functions, since the function structure is untouched and the effect of such adjustment is limited. Our GP-based ranking function discovery approach provides a framework which could automatically learn the "optimal" ranking function for the given context. As the structures of discovered ranking functions are not constrained, these customized functions could provide striking performance on the target collection where static ranking functions can not.

Ranking function discovery is essentially an optimization problem. We are looking for the global optimal point in the space, which consists of all the possible ranking functions. However this task is completely different from the traditional high-dimension optimization problem, since the space of ranking functions *is no longer a coordinate system* (As in Abstract, make this clearer. Do you mean a vector space or metric space or measure space?). Conventional approaches for solving optimization problems, such as

conjugate gradient, linear programming, nonlinear programming, and simulated annealing, can hardly work here. Also the ranking function space consists of an infinite number of elements, which makes it impossible to get the "optimal" point for random search and exhaustive search. As we showed before, functions could be expressed by trees. We can actually treat the ranking function space as a space consists of all kinds of tree structures. Genetic Programming shows its sharp edge in solving such kind of problems, since its internal tree structure representation for "individuals" can be perfectly used for describing ranking functions. This is the major motivation to choose GP for the ranking function discovery task.

### 4.3 Outline of our GP-based ranking function discovery system – ARRANGER

In this section, we give a brief introduction to the ARRANGER engine. Please refer to [Fan 2003a, Fan 2003b] for a more detailed introduction and for validation.

Basically a ranking function consists of three parts: variables, constants, and operations (which connect the first two parts). Hence we need to identify all the potential variables that are used in the ranking function by ARRANGER. Some examples for these variables are tf, tf_query, tf_max, length, N, tf_avg, tf_Avg_Col, df_max_Col, df, etc. Table 1 gives the meaning of these variables.

| | |
|---|---|
| tf | Query term frequency in the document (vector) |
| tf_query | Query term frequency in the query (vector) |
| tf_max | The maximum term frequency in a document (scalar) |
| Length | Document length in the number of words (scalar) |
| Length_avg | Average document length in the number of words (scalar) |
| N | Number of documents in the collection (scalar) |
| tf_avg | Average term frequency in the current document (scalar) |
| tf_avg_Col | Average term frequency for all the documents in the collection (scalar) |
| df_max_Col | Maximum document frequency for a word in the collection (scalar) |
| df | Document frequency for the query words (vector) |

Table 1. Definitions for variables

There are two different types of variables, scalar and vector. Some of these predefined variables are summaries calculated for the whole collection or a specific document, such as tf_max, N, tf_Avg_Col, etc. These variables belong to the category of scalar variable. The remaining variables have vector nature, such as tf_doc and tf_query. We defined that when such variables appear in a ranking function, they represent vectors, instead of single numbers. For example, if a query has n words in it, tf_doc could be represented by $(x_1, x_2, \ldots, x_n)$, where $x_i$ ( i = 1,2,…,n ) is the term frequency (tf) of the query's *i*th word in the document. For constants, they are defined to be scalar only. Based on pre-selected variables and constants, we define two types of functions (operations), single-parameter functions (denoted by $\sigma(\cdot)$) and two-parameter functions (denoted by $\circ$). Single-parameter functions include log( ) and sqrt( ). Two-parameter functions include +, -, *, /. Some functions, such as log( ), sqrt( ) and /, need to be protected, since the domain of these functions is not the whole real number space. As a variable could be a scalar or a vector, those functions must take that into consideration. For one-parameter functions, we define $\sigma(x) = y$ and $\sigma((x_1, x_2, \ldots, x_n)) = (\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n))$, where x, y and $x_i$ represent scalar variables and

$(x_1, x_2, \ldots, x_n)$ is used to represent vectors. For two-parameter functions, we define $x \circ y = z$, $x \circ (x_1, x_2, \ldots, x_n) = (x \circ x_1, x \circ x_2, \ldots, x \circ x_n)$ and $(x_1, x_2, \ldots, x_n) \circ (y_1, y_2, \ldots, y_n) = (x_1 \circ y_1, x_2 \circ y_2, \ldots, x_n \circ y_n)$, where x, y, z, $x_i$ and $y_i$ represent scalars and $(x_1, x_2, \ldots, x_n)$ represent a vector. Following our definitions for variables and functions, when a vector variable appears in the ranking function, the final result also is a vector, where a scalar usually is needed to measure the similarity degree between a document and a query. In this case, we further define that the return value of a ranking function is the summation of all the elements when a vector is finally returned by that function. Based on all those rules defined by us, the ARRANGER could work on discovering ranking functions. Also when we plug in the newly-discovered functions into our search engine, the same rules must be followed.

Queries in the TREC 6, 7, and 8 Ad-Hoc task (topic 301- 450) are used to discover ranking functions. According to the procedure described in section 3, the collection is first processed into dictionary and inverted files, such that our search engine can work on them. For each query, the search engine returns the top 5000 document names using an arbitrary function. Any popular ranking functions, which have been proved effective, could be used for this purpose. We used the Okapi BM25 ranking function for this first scan. On average more than half of all the relevant documents are listed in the top 5000 documents for each query. Therefore those documents have included enough relevant documents, whose properties could be learned later. According to the relevance judgments, those documents are separated into two groups, relevant and nonrelevant. Each group needs to be randomly divided into three parts, called training, validation, and testing data set. Then we randomly combine the relevant and nonrelevant documents associated with each data set. Now the training, validation, and testing sets all include relevant and nonrelevant documents in random order. The fitness value for a ranking function is the average precision we could get in our system when using that function.

The framework of ARRANGER works as follows: First, the best ranking functions learned from the training set are stored and the rest are discarded. Then those functions are tested on the validation set. According to their performance, the functions which do not have consistent performance on both data sets are screened out. Finally, "survived" functions are tested again on the test data set. The same screening rule follows. Only the most robust and consistent functions are selected and they form the ranking function candidate pool. Since an appropriate stopping rule is hard to find for the Genetic Programming approach, over-training is inevitable unless protecting rules are set. By running the ranking functions on two other independent data sets, over-trained functions are filtered out once performance inconsistencies appear.

We used ARRANGER to discover "optimal" functions on the Robust Track collection. We tested the automatically learned functions on three types of queries: description query, short query, and long query as described in the Section 3. Table 2 shows the results on the entire collection. From this table, you can see that significant improvement is achieved by replacing the Okapi BM25 function with our newly-discovered functions.

| | Description query (average precision) | Short query (average precision) | Long query (average precision) |
|---|---|---|---|
| Okapi BM25 (baseline) | 0.1880 | 0.2194 | 0.2375 |
| GP func1 | 0.2173 (+15.6%) | 0.2394 (+9.1%) | 0.262 (+10.3%) |
| GP func2 | 0.2079 (+ 10.6%) | 0.2317 (+5.6%) | 0.2607 (+9.8%) |
| GP func3 | 0.2047 (+ 8.9%) | 0.2282 (+4.0%) | 0.259 (+9.1%) |
| GP func4 | 0.2036 (+8.3%) | 0.2245 (+2.3%) | 0.2602 (+9.6%) |

Table 2. Performance comparison of Okapi BM25 and GP functions on 150 queries of Ad-Hoc task at TREC 6, 7, and 8.

## 5. Other performance improvement techniques

### 5.1 Pseudo-relevant feedback

Pseudo-relevance feedback (automatic query expansion) is the process of adding more terms to a user's query to promote performance of search engines. It is a widely-used and effective technique, especially for very short queries. In pseudo-relevance feedback, a small number of documents are first retrieved according to the user's query and these documents are assumed to be relevant. Words in those documents as well as words in the original query are sorted according to a weighting function. An expanded query is generated by selecting some words from this list. There are many variations in using different weighting functions and strategies to select words for the new query.

We apply various pseudo-relevance feedback techniques, based on new functions discovered by our ARRANGER. They are Rocchio, Ide dec-hi, CHI, KLD, RSV, DRC, and a variation of KLD, which we deduced by probability theory. Those techniques are applied on both description queries and long queries. They provide significant performance improvement on both types of queries. As we expected they improve more on description queries than long queries. For each approach, there are several parameters to be adjusted, for example, the number of documents assumed relevant, the number of terms for the expanded query, and parameters in the weighing function. A factorial design was used to look for the "best" parameter settings, which provides at the same time a high performance mean and low performance variation for ad-hoc tasks in TREC 6, 7, and 8. After comparison, we found Rocchio and Ide dec-hi are the best query expansion schemes on our automatically learned functions. Table 3 gives the performance comparisons.

| | Description query (average precision on 150 queries) | Long query (average precision on 150 queries) |
|---|---|---|
| GP function 1 without QE (baseline) | 0.2173 (+15.6%) | 0.2394 (+9.1%) |
| GP function 1 + Rocchio | 0.2422 (+28.9%) | 0.2661 (+ 12.0%) |
| GP function 1 + Ide Dec-Hi | 0.2390 (+27.1%) | 0.2744 (+15.5%) |

Table 3. – The effects of pseudo-relevance feedback on performance

### 5.2 Rank fusion – combine scores from different ranking functions

Since many high quality ranking functions have been learned, an old saying "two heads are better than one" could be used in our system to further improve performance. In our experiment, three GP-based functions and Okapi BM25 are combined to produce a new ranking function. Because the relevance judgment only provides binary relevant (1) and nonrelevant (0) information, logistic regression is an appropriate tool to find such a relationship. Let p denote the probability that a document is relevant to the query and let gp1, gp2, gp3, and okp represent scores returned by our three GP-based functions and the Okapi BM25 function for this document, respectively. Our initial model is

$$\text{logit}(p) = \beta_0 + \beta_1 * \text{gp1} + \beta_2 * \text{gp2} + \beta_3 * \text{gp3} + \beta_4 * \text{okp} + \text{INT}$$

INT includes all the possible two factor, three factor, and four factor interactions. Only after including interaction terms, the similarity degree between a document and query could be appropriately measured when conflict scores are given by different ranking functions. Otherwise a main-effect-only model can not fit the data well.

For each of 150 queries, the search engine generates names and scores of the top 300 documents returned by these four ranking functions. A union operation is applied on all the returned documents, therefore we generate a huge matrix with 5 columns (gp1 score, gp2 score, gp3 score, okp score, and

relevance information). If a document was not listed in the top 1000 list by a function, its score associated with this function is assigned to 0. After model selections, we achieved such a model: logit(p) = $\beta_0$ + $\beta_1$*gp1 + $\beta_2$*gp2+ $\beta_3$*gp3+ $\beta_4$*okp + $\beta_5$*gp1:gp2 + $\beta_6$* gp1:gp3 + $\beta_7$* gp2:okp + $\beta_8$*gp1:gp3:okp + $\beta_9$*gp2:gp3:gp4+ $\beta_{10}$*gp1:gp2:gp3:okp, where X:Y represents the interaction between factor X and Y. All $\beta_i$'s are highly significant (with p-value < $10^{-5}$ ) in this model. The combined ranking function is then tested on the whole collection and the result is shown in Table 4.

| | 150 queries (long) | 50 test old queries (long) |
|---|---|---|
| Okapi BM25 | 0.2375 | 0.1251 |
| GP1 function | 0.2620 | 0.1393 |
| GP2 function | 0.2602 | 0.1334 |
| GP3 function | 0.2607 | 0.1346 |
| Comb function | 0.2666 | 0.1417 |

Table 4. Performance comparison between combined function and other functions

The performance of the combined function is superior to all other functions from which it is generated. Another appealing property we found in experiments is that the combined function produces the smallest performance variation on TREC 6, 7, and 8 among all the ranking functions. Experiments show that our ranking function fusion approach improves not only the performance but also the consistency of the information retrieval system, although the difference is not statistically significant.

**6. Results**

We submitted five independent runs for this year's Robust Track. Our submissions do not involve any human intervention, so they are all automatic runs. The first four runs use all the topic fields and the last one only uses the description field of topics. Table 5 gives the detailed description of our submissions. Table 6 summarizes the final evaluation results from TREC for all 5 runs.

| **Run Number** | **Description** |
|---|---|
| VTcdhgp1 | In this run, we first search long queries (all fields of topics) against Robust collection, using a linearly combined ranking function (combining 3 GP functions we derived from experiments with Okapi). Secondly, we assume the top 6 documents are relevant and use Ide dec-hi approach to "expand" the description field of each query to 22 words. Finally, we search the "expanded query" against the Robust collection again using a GP ranking function, which we derived from previous experiments. |
| VTgpdhgp2 | Same as VTcdhgp1 except that we use GP ranking function for the first search and expand the query to 14 words instead of 22 words. |
| VTcdhgp3 | Same as VTcdhgp1 except that we expand the query to 23 words. |
| VTgpdhgp4 | Same as VTcdhgp2 except that we expand the query to 17 words. |
| VTDokrcgp5 | In this run, we first search description field of queries against Robust collection using Okapi BM25 ranking function. Secondly, we assume the top 8 documents are relevant and use Rocchio method to "expand" the description field of each query to 22 words. Finally, we search the "expanded query" against the Robust collection again using a GP ranking function, which we derived from previous experiments. |

Table 5. Description of our five official submissions

| Run No. | MAP | P10 | #>Median | #Best |
|---|---|---|---|---|
| VTcdhgp1 | 0.2649 | 0.432 | 62 | 3 |
| VTgpdhgp2 | 0.2731 | 0.449 | 69 | 3 |
| VTcdhgp3 | 0.2637 | 0.432 | 61 | 2 |
| VTgpdhgp4 | 0.2696 | 0.448 | 65 | 2 |
| VTDokrcgp5 | 0.2563 | 0.408 | 60 | 4 |
| ***In total, we contribute 14 queries that have the best performance among 100 queries*** | | | | |

Table 6. Official submission results. The last run is based on the description field only.

As can be seen from Table 6, we contribute 14 queries that have the best performance in 100 topics. Our last run based on the description field performs even better than the median submission run (MAP= 0.2387, P10 = 0.3990). Our best run trails by 12% in MAP and 8% in P10 from the best team. We consider the performance results very satisfactory considering the fact that we had a relatively low baseline system. We are currently in the process of improving the parsing and indexing process to improve the baseline performance.

## 7. Conclusion

In this paper, we used ARRANGER, a GP-based discovery engine, to discover several ranking functions for the Robust Track. We observed up to 16% performance improvement over our baseline Okapi system. The experimental results show that the automatically learned ranking functions are capable of outperforming expert-designed functions.

In addition, we also tried some other popular performance improvement techniques, such as pseudo-relevance feedback and a ranking fusion technique. Both of them work well with those new functions and help further improve our system performance. Not only do they increase the average precision, but they also make the system more robust and provide less performance variation on different query sets.

## References


W. Fan, M.D. Gordon, P. Pathak, "A generic ranking function discovery framework by genetic programming for information retrieval", *Information Processing and Management*, in press, 2003a.

W. Fan, M. D. Gordon, P. Pathak, "Discovery of context-specific ranking functions for effective information retrieval by Genetic Programming", *IEEE Transactions on Knowledge and Data Engineering*, in press, 2003b.

J. H. Holland, *Adaptation in Natural and Artificial Systems*, the University of Michigan Press, Ann Arbor, 1975.

S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford, "Okapi at TREC-4", in D. K. Harman, editor, *Proceedings of the Fourth Text Retrieval Conference*, pages 73–97. NIST Special Publication 500-236, 1996.

G. Salton and C. Buckley, "Term weighting approaches in automatic text retrieval", *Information Processing and Management*, 24(5): 513–523, 1988.

J. Zobel and A. Moffat, "Exploring the similarity space", *SIGIR Forum*, 32(1): 18–34, 1998.