A Cooperative Coevolution Framework for Parallel Learning to Rank

Shuaiqiang Wang, Member, IEEE, Yun Wu, Byron J. Gao, Ke Wang, Senior Member, IEEE, Hady W. Lauw, Member, IEEE, and Jun Ma, Member, IEEE

Abstract—We propose CCRank, the first parallel framework for learning to rank based on evolutionary algorithms (EA), aiming to significantly improve learning efficiency while maintaining accuracy. CCRank is based on cooperative coevolution (CC), a divide-and-conquer framework that has demonstrated high promise in function optimization for problems with large search space and complex structures. Moreover, CC naturally allows parallelization of sub-solutions to the decomposed sub-problems, which can substantially boost learning efficiency. With CCRank, we investigate parallel CC in the context of learning to rank. We implement CCRank with three EA-based learning to rank algorithms for demonstration. Extensive experiments on benchmark datasets in comparison with the state-of-the-art algorithms show the performance gains of CCRank in efficiency and accuracy.

Index Terms—Cooperative Coevolution, Learning to Rank, Information Retrieval, Genetic Programming, Immune Programming.

1 INTRODUCTION

Ranking schemes are critical for information retrieval (IR) systems and Web search engines. Traditional ranking methods include the Boolean model, vector space model, probabilistic model and language model. Recently, learning to rank has received increasing attention [1], [2], [3], [4], [5], [6], [7]. Given training data, a set of queries each associated with a list of search results labeled with relevance degree, learning to rank returns a ranking function that can be used to order search results for future queries.

With learning accuracy being the primary concern, learning efficiency can be a crucial issue [1], [8]. Due to the diversity of queries and documents, learning to rank involves larger and larger training data with many features. For example, the CSearch dataset contains ∼25 million instances with 600 features, and the MSLR-WEBS30K dataset collects ∼19 million instances with 136 features. Recently, utilization of click-through data [9] bypasses manual labeling and enables collection with even larger training data. In addition, due to the rapid growth of the Web, ranking functions need to be relearned repeatedly. Therefore, it is important to achieve high efficiency through parallelization while maintaining accuracy.

In light of this, we propose CCRank, a parallel learning to rank framework based on cooperative coevolution (CC), aiming to significantly improve learning efficiency while maintaining accuracy.

Evolutionary algorithms (EAs) are derived from Darwinian evolutionary principles and widely applied in computationally difficult optimization and classification problems [10]. An evolutionary algorithm maintains a population of individuals (solutions) that evolve from generation to generation. Each individual is associated with a fitness score. In standard EAs, fitness scores are static (do not vary over time) and absolute (independent of other individuals). However, selecting an adequate fitness function for a particular optimization problem can be as difficult as solving the problem itself [11].

Coevolutionary algorithms offer an alternative, where the fitness of an individual is relative and determined in relation to other individuals. In addition, such algorithms maintain a collection of EAs that co-evolve simultaneously, where the EAs can interact in either a competitive or a cooperative manner.

In competitive coevolution, individuals represent complete solutions that are gradually refined. Cooperative coevolution (CC) is used in situations where a problem can be naturally decomposed into sub-components. In CC, individuals represent such sub-components and are assessed in collaboration with other individuals so as to form complete solutions [12], [13], [14].

CC is advantageous in solving problems with exceptionally large search space and complex structures [15]. They have been successfully applied to a variety of domains, e.g., function optimization [16], manufacturing scheduling [17] and neural network design [18]. CC follows a divide-and-conquer strategy, decomposing
a complex problem into sub-problems and combining sub-solutions (individuals) in the end to form the final complete solution.

Recently, new advances in multicore processor technology have enabled a wave of parallelization in achieving high performance computing. In CC, EAs evolve almost separately. Thus, the evolving process can be naturally parallelized, allowing significant improvement in learning efficiency. Unfortunately, this privilege has not been explored previously for learning to rank. In this paper, we investigate parallel CC in the context of learning to rank.

Early CC-based algorithms (e.g., JACC-G [19], CCEA [16], CCPSO [20]) solve optimization problems. Adapting CC to learning to rank is non-trivial, which requires completely different treatments for solution representation, problem decomposition, evolution, and combination.

Instead of a vector representation as in optimization algorithms, in our proposed CCRank, a solution is a ranking function represented by a binary tree for easy parsing, implementation, and interpretation [21], [22], [23]. Accordingly, individuals represented by sub-trees form a set of populations which evolve in parallel.

For problem decomposition, optimization algorithms divide the feature space into subspaces of features, each corresponding to a sub-problem. Sub-problems and individual sub-solutions are within their predetermined subspaces. This decomposition is appropriate because their search space is a Cartesian product of features. In CCRank, the search space is non-linear. A similar decomposition would lead to significant loss of information and promising search space. In this study, the decomposition procedure is as follows: Firstly, we initialize $L$ solutions randomly, represented by trees, from the full feature space. Then, we decompose each tree into $N$ sub-trees, resulting in $N$ populations each with $L$ individuals. All populations co-evolve from generation to generation in parallel.

Based on the same argument, the evolution process differs in optimization algorithms and CCRank. In the former, individuals evolve within the same predetermined subspace. In the latter, an open approach is adopted in the sense that any feature from the full feature space can be selected into the subspace. Although the features in a subspace change dynamically, the size of the subspace is upper-bounded, to be precise, $2^{d-1}$ where $d$ is the depth of the sub-tree.

The combination process assembles individuals into a complete solution. For optimization algorithms, combination is straightforward as individuals are represented as vectors, where combination is done at the very end of the whole iterative evolution process. In CCRank, individuals are sub-trees and we need to assemble them properly to form a complete solution. In addition, since learning to rank requires validation of candidate ranking models, each being a solution, we need to collect a pool of diverse candidates produced by different generations. Thus, in CCRank combination is done after each generation during the iterative evolution process.

In CCRank, parallelization of EAs is enabled within generations. Although EAs for sub-solutions can evolve almost independently, we cannot allow them to execute continuously until the end. This is because, as discussed above, in CCRank we need to perform combination to produce candidates after each generation. Thus, EAs must suspend their execution at the end of each generation. After the combination is done, the EAs will continue to execute in parallel again in the next generation.

To demonstrate the improvements achieved by CCRank, we conduct two series of experiments with benchmark datasets in comparison with the state-of-the-art algorithms. Experimental results show the performance gains of CCRank in efficiency and accuracy.

Contributions. Our main contributions are as follows:
1. We investigate parallel cooperative coevolutionary algorithms in the context of learning to rank.
2. We propose CCRank, the first parallel framework for learning to rank, aiming to significantly improve learning efficiency while maintaining accuracy.
3. We implement CCRank with three EA-based learning to rank algorithms for demonstration.

Organization. The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 introduces the background. Section 4 proposes the CCRank framework. Section 5 implements CCRank with three EA-based learning to rank algorithms. Section 6 reports the experimental results. Section 7 concludes the paper.

2 RELATED WORK

2.1 Learning to Rank for Information Retrieval

Learning to rank. Learning to rank has received increasing attention recently from both machine learning and Information Retrieval (IR) communities. Now we review several representative algorithms. RankSVM [3], [28] optimizes a concave lower bound of mean average precision ($MAP$), an IR evaluation measure) with structured SVM. In [29], the same idea is extended to optimize other IR evaluation measures of normalized discounted cumulative gain ($NDCG$) and mean reciprocal rank ($MRR$). RankBoost [2] adopts a Boosting approach for learning to rank. It minimizes the weighted number of disordered pairs. AdaRank [4] is also a Boosting approach, but it minimizes a loss function directly defined on the performance measures. ListNet [1] introduces a probabilistic list-wise loss function, and uses neural
network and gradient descent to train a list prediction model. iGBRT [5] proposes an initialized gradient boosted regression trees (GBRT) algorithm, which uses GBRT to further refine the results of Random Forests for ranking. LambdaMART [6] is derived from the tree-boosting optimization MART [30] and the listwise LambdaRank [31], which can combine their strengths and demonstrate promising performance for widely used information retrieval measures. Top-k learning to rank [7] presents an efficient labeling strategy to generate the ground-truth of the top-k ordering items, and proposes FocusedRank, a new ranking model to fully capture the characteristics of the top-k ground-truth.

**Evolutionary algorithms-based approaches.** Discovery of ranking functions based on evolutionary algorithms (EAs) has been extensively studied in the past few years. Fan et al. [22] proposed a genetic programming (GP) [24] based approach to automatically generate specific weighting schemes for different contexts, and demonstrated that GP was effective in improving the performance of the document ranking problem in information retrieval. Besides, Fan et al. [32] investigated the effects of the fitness functions on genetic programming-based learning to rank for information retrieval. Trotman [33] added 4 baseline ranking functions as individuals in the initial population to guarantee that the ranking precision in training was no worse than the baselines, including the inner product [34] and cosine similarity [35] between query and document vectors, the probability model [35], and Okapi BM25 [36]. Almeida et al. [21] used parts of well-known, significant, and proven effective ranking formulas as basic components for constructing ranking functions. Wang et al. [23] proposed RankIP, a ranking function discovery approach based on immune programming [25].

Other algorithms are surveyed in [37]. Different from all of these algorithms, CCRank is a parallel framework for EA-based learning to rank algorithms.

### 2.2 Parallel Machine Learning

**Parallel machine learning.** Many sophisticated machine learning algorithms cannot process large data sets. Parallelization is an effective way of achieving speed-up. IBM Parallel Machine Learning Toolbox (PML)\(^1\) contains the parallel version of many commonly-used machine learning algorithms (e.g., SVM), and includes an API for incorporating additional algorithms. The toolbox can work on various types of architecture, e.g., multicore machines. By distributing the required computation to computing nodes in a parallel fashion, training can be expedited by several orders of magnitude.

With the advances of multicore technology, parallel machine learning is emerging as an active research discipline. For example, Collobert et al. [38] proposed a parallel algorithm for SVMs for very large-scale problems. Instead of processing whole training dataset in one step, it splitted the dataset into subsets and optimized the subsets with multiple SVMs in parallel. Experimental results showed that the time complexity of the parallel algorithm growed linearly with the number of instances. Graf et al. [39] presented an effective combination strategy for the parallel SVM algorithm, where the partial results of SVMs were combined and filtered again in a “Cascade” of SVMs, until the global optimum was reached. Chu et al. [40] introduced a “summation form” for machine learning algorithms fitting the statistical query model, and adapted the map-reduce paradigm [41] to parallelize variety of learning algorithms, including locally weighted linear regression, k-means, logistic regression, naive Bayes, SVM, ICA, PCA, gaussian discriminant analysis, EM, and back propagation neural network.

Several efforts have been made on parallelizing learning to rank algorithms for significant efficiency gains. For example, Shukla et al. [42] implemented the ListNet [1] algorithm with the parallel Spark framework, and De Soursa et al. [43] proposed a GPU-based parallel learning to rank algorithm PLRAR to accelerate their active learning-based ranking algorithm LRAR. However, these algorithms can only parallelize certain ranking algorithm, and the efforts on a parallel framework to parallelizing a group of learning to rank algorithms are still very limited.

**Cooperative coevolution.** Cooperative coevolution (CC) [16] is a divide-and-conquer coevolutionary architecture for solving problems with exceptionally large search space and complex structures.

Early CC-based algorithms are successfully applied to the function optimization problems. For example, CCDE [44] adopts the differential evolution (DE) [45], [46] to implement the cooperative co-evolutionary framework, demonstrating significant improvement in accuracy over the conventional DE and the cooperative co-evolutionary genetic algorithm (CCGA). JACC-G [19] uses JADE [47], a recent and efficient variant of DE, to evolve the subsolutions and generate a weighting vector for combination of the subsolutions. Experiments demonstrates high efficiency on optimizing 1000-dimension benchmark functions. DECC [48] provides an effective problem decomposition method in the CC framework for large scale non-separable problems, where interacting variables were captured and grouped into subcomponents based on their delta values. CCPSO [20] presents a new cooperative coevolving particle swarm optimization (CCPSO) algorithm for scaling up particle swarm optimization (PSO) algorithms in solving large-scale optimization problems.

CCRank is the first framework adapting CC to the learning to rank problem.

---

3 The Learning to Rank Problem

Let \( D \) be a collection of documents, each represented by a vector of feature values. In an information retrieval system, for a query \( q \), a list of documents from \( D \) are returned as search results, where the documents are ranked according to their relevance to \( q \).

For a given query \( q \), the ground truth relevance of the documents with respect to \( q \) (judged by human experts) is defined as a function \( rel : D \rightarrow \mathbb{N} \), where \( \mathbb{N} \) is the natural number set indicating different relevance levels. In some cases, \( rel \) is a binary function, mapping a document to either 0 (irrelevant) or 1 (relevant). In our experiments, we considered 3 relevance levels of 0 (irrelevant), 1 (partially relevant), and 2 (relevant).

Let \( f : D \rightarrow \mathbb{R} \) be a ranking function assigning real number relevance scores to documents, where \( \mathbb{R} \) denotes the real number set. The effectiveness of ranking functions can be evaluated by a given measure \( s \), such as precision at \( k \) (P@\( k \)), mean average precision (MAP), and normalized discount cumulative gain (NDCG@\( k \)).

Given a training data set \( T \) and an evaluation measure \( s \), the learning to rank problem is to learn a ranking function \( f \) based on \( T \) such that \( s(f) \) is maximized.

4 The CCRank Framework

CCRank parallelizes evolutionary algorithms (EA)-based learning to rank algorithms.

4.1 EA-based Learning to Rank Algorithms

EA-based learning to rank has become one of the most important branches in the learning to rank field. It can directly optimize the non-continuous and non-differentiable IR evaluation measures such as P@\( k \), MAP and NDCG@\( k \) for achieving high accuracy [37]. Besides, it can obtain non-linear or even non-polynomial ranking functions for improvement in accuracy based on basic function operators such as \( \sqrt{\cdot} \), log, sin and cos [37], resulting from much larger search space.

In recent years, many evolutionary algorithms and their variants have been proposed, which share substantial similarities in structure. Based on these structural similarities, in this study we provide a general algorithm for EA-based learning to rank.

The pseudocode of EA-based learning to rank is shown in Algorithm 1, where \( P_i^{(g)} \) denotes population \( i \) in the \( g^{th} \) generation, \( \mathcal{C} \) denotes the candidate set, and \( s \) denotes the fitness function of individuals.

Line 1-4 performs initializations, including randomly generation of \( N \) individuals for the initial population \( P_0 \) (line 1), evaluation of each individual in \( P_0 \) (line 2), selection of the best individual \( f^{(0)} \) from \( P_0 \) (line 3), and collection of \( f^{(0)} \) into the candidate set \( \mathcal{C} \) (line 4).

Lines 5-10 show the whole evolution process from generation to generation. In particular, line 6 evolves current population of individuals \( P_i^{(g)} \) based on the previous generation \( P_i^{(g-1)} \), line 7 evaluates each individual in \( P_i^{(g)} \), line 8 selects the best individual \( f^{(g)} \) from \( P_i^{(g)} \), and line 9 puts \( f^{(g)} \) into the candidate set \( \mathcal{C} \).

Lines 11-12 calculate the performance measures \( s_T(C) \) and \( s_V(C) \) for the candidates using the training set \( T \) and validation set \( V \), based on which line 13 selects the ranking function \( f \). Let \( f_i \in \mathcal{C} \) be the \( i \)th candidate ranking function. The following formula [23] is used for selection:

\[
\arg\max_i ( (\alpha \times s_T(f_i) + \beta \times s_V(f_i)) - \gamma \times \sigma_i ),
\]

where \( \gamma \) is a constant, \( \sigma_i \) is the standard deviation of \( s_T(f_i) \) and \( s_V(f_i) \), and the values of \( \alpha \) and \( \beta \) are based on the sizes of the training set and validation set, i.e.,

\[
\alpha = \frac{|T|}{|T| + k \times |V|}, \beta = \frac{k \times |V|}{|T| + k \times |V|}.
\]

4.2 Overview of CCRank

CCRank adapts parallel cooperative coevolution (CC) to EA-based learning to rank algorithms. It learns a ranking function from training data in two phases.

CCRank starts with the problem decomposition phase (detailed in Section 4.4). Firstly \( L \) initial solutions, represented by trees, are generated randomly from the full feature space. Then, each tree is decomposed into \( N \) sub-trees, resulting in \( N \) populations each with \( L \) individuals.

The evolution phase (detailed in Section 4.5) starts after problem decomposition. It is an iterative process, where \( N \) populations co-evolve in parallel from generation to generation. Each population maintains a collection of individuals and a winner, which is the best individual in the population with the biggest fitness value.

At the end of each generation, the parallel execution is suspended and a complete solution \( f \) is produced by a combination operation, which assembles \( N \) winners evolved in \( N \) populations. Then \( f \) is collected into a
solution pool as a candidate ranking function, which is used to be selected with the validation data after the iterative evolution process.

Note that in standard CC, combination is done only once after the evolution ends. In CCRank, combination needs to be applied at the end of each iteration of the evolution phase. That is because learning to rank, as a classification problem, requires validation of candidate classifiers (solutions), for which we need to collect a pool of diverse candidates produced by different generations.

After the evolution process ends, validation data are used to select the best solution among all candidates as the final ranking function to return.

4.3 Solution Representation

EA-based algorithms for optimization problems use vectors to represent solutions. For the learning to rank problem, the ranking function \( f \) can be non-linear. In CCRank, we use the tree structure to represent solutions. Accordingly, individuals are represented as sub-trees. Trees not only have sufficient expressive power to represent non-linear functions \([21], [22]\), they also have the advantage of allowing easy parsing, implementation and interpretation.

In particular, for each tree, the leaf nodes contain features and constants. The features are mainly classified as content features and hyperlink features. The content features can be further classified as low-level features (basic statistical information of the collection, documents and queries, such as term frequency \( tf \) and inverse document frequency \( idf \)) and high-level features (the outputs of some classic approaches such as BM25 \([49]\) and LMIIR \([50]\)). The hyperlink features usually include the numbers of hyperlinks to the documents, output of PageRank algorithm, etc. Constants serve as coefficients of features in \( f \). In CCRank, 19 constants are used, which are 0.1, 0.2, ..., 0.9, 1, 2, ..., 10.

The internal nodes of the trees contain 8 basic function operators:

- 4 basic arithmetic function operators \(+, -, \times \text{ and } ÷\).
- 4 non-polynomial operators \(\sqrt{x}, \log, \sin \text{ and } \cos\).

In some operators, parameters need protection mechanisms. For example, \( x \) in the logarithm function \( \log(x) \) should be greater than zero, \( y \) in the division function \( x ÷ y \) cannot be zero, and \( x \) in the square root function \( \sqrt{x} \) should be no less than zero.

In CCRank, we design two protection mechanisms for the protected parameter \( x \):

- **PM(1)** \( x := |x| \), if \( x < 0 \).
- **PM(2)** \( x := \varepsilon \), if \( x = 0 \), where \( \varepsilon \) is a number close to zero. In our experiments \( \varepsilon = 0.000001 \).

Table 1 presents these protection mechanisms in CCRank, where \( x\text{-PM}(i) \) means \( x \) needs the \( i \)th protection mechanism in the basic function operation.

<table>
<thead>
<tr>
<th>Function Operations</th>
<th>Example</th>
<th>Protected Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+, -, \times)</td>
<td>( x + y )</td>
<td>-PM(1)</td>
</tr>
<tr>
<td>(\div)</td>
<td>( x ÷ y )</td>
<td>y-PM(2)</td>
</tr>
<tr>
<td>(\sin, \cos)</td>
<td>( \sin(x) )</td>
<td>-PM(2)</td>
</tr>
<tr>
<td>(\sqrt{\cdot})</td>
<td>( \sqrt{x} )</td>
<td>x-PM(1)</td>
</tr>
<tr>
<td>(\log)</td>
<td>( \log(x) )</td>
<td>x-PM(1) &amp; y-PM(2)</td>
</tr>
</tbody>
</table>

\[ d = \left\lceil \log_2(n_F + n_C) \right\rceil + 1, \]

where \( \lceil x \rceil \) is the seiling function that returns the smallest integer not less than \( x \). For example, for the LETOR dataset, \( n_F = 46 \). Let \( n_C = 19 \), and thus \( d = \left\lceil \log_2(46 + 19) \right\rceil + 1 = 7 + 1 = 8 \) such that trees has \( 2^8 - 1 = 128 \geq 46+19 \) leaf nodes. For the MSLR-WEB30K dataset, \( n_F = 136 \), and thus \( d = \left\lceil \log_2(136 + 19) \right\rceil + 1 = 8 + 1 = 9 \) such that trees has \( 2^9 - 1 = 256 \geq 136 + 19 \) leaf nodes.

![Fig. 1. Tree representation.](image)

A tree can be parsed into a function. Figure 1 shows the tree representation for an example ranking function \((f_1 - f_2) + (0.5 * f_3)\).

4.4 Decomposition Phase

CC-based optimization algorithms divide the feature space into subspaces of features, each corresponding to a sub-problem. This decomposition is appropriate because their search space is a Cartesian product of features. In CCRank, the search space is non-linear. A similar decomposition would lead to significant loss of information and promising search space.

In CCRank, initially \( L \) solutions, represented by trees of depth \( d \), are generated randomly from the full feature space. Then, each tree is decomposed into \( N \) sub-trees, resulting in \( N \) populations, each with \( L \) individuals. Each population will be assigned an EA to evolve. Figure 2 shows the decomposition of a single tree (left-hand side) into \( N = 4 \) individuals (right-hand side).

The depth of sub-trees (individuals) \( d_I \) upper-bounds the feature space of individuals. This parameter is used in CCRank whenever individuals are generated. The depth of assembler (to be explained in Section 4.5) is determined by \( d_A = \left\lceil \log_2 n_P \right\rceil \) where \( n_P \) is the number of processors used in the parallel evolution process. Thus, \( d_I = d - d_A \), which implies that the size of the feature space of individuals is upper-bounded by \( 2^{d_I-1} \). For...
example, suppose the depth of trees (solutions) $d = 8$. The calculation of $d$ was introduced in Section 4.3. Let the number of processors $n_P = 8$. Then, the depth of assembler $d_A = \lceil \log_2 8 \rceil = 3$, and the depth of individuals $d_I = 8 - 3 = 5$.

### 4.5 Evolution Phase

Resulting from the differences in search spaces, linear vs. non-linear, the evolution process differs in function optimization algorithms and CCRank. In the former, individuals evolve within the same predetermined subspace. In the latter, an open approach is adopted in the sense that any feature from the full feature space can be selected into the subspace.

Evolution in CCRank executes iteratively, and the number of iteration is predetermined by a given parameter. Each iteration contains a generation of evolving populations. $N$ populations evolve in parallel, each maintaining a collection of individuals and a winner, which is the best individual in the population with the biggest fitness value.

Figure 3 illustrates the iterative evolution process for 4 populations in EA1, EA2, EA3 and EA4 respectively. Within each iteration, there is a generation as shown at the top. Within each generation, in the beginning, individuals of each population for the current generation are generated based on individuals from the previous generation. Then, fitness values of individuals are calculated in a cooperative manner, as to be detailed shortly.

After the generation, parallel execution is suspended temporarily and the candidate solution for the current generation will be generated. First, the winner for each population is updated based on the calculated fitness. Then, all the winners are combined, with the help of the assembler, into a candidate solution. If the preset maximum number of iterations is not met, a new iteration will start and the 4 populations will continue to evolve in a new generation in parallel.

**Combination Operation.** The combination process assembles individuals into a complete solution. For function optimization algorithms, since individuals are vectors, combination is straightforward, which is done at the very end of all generations. In CCRank, individuals are sub-trees and we need to assemble them properly to form a complete solution. In addition, since learning to rank requires validation of candidate ranking models, each being a solution, we need to collect a pool of diverse candidates produced by different generations. Thus, in CCRank combination is done after each generation during the evolution process. Precisely, combination occurs in two cases: fitness calculation (right before the end of the generation, as shown in Figure 4) and candidate generation (right after the end of the generation as shown in Figure 3).

Combination is the inverse process of decomposition. In Figure 2, 4 individuals $i_1 \sim i_4$ (right-hand side) are combined into a complete solution (left-hand side) with the help of the assembler. Assemblers are used to assemble individuals into complete solutions, where they form “crowns” of solution trees. Thus, all of their nodes are internal nodes of solution trees and contain function operators only. In particular, we use $+$ so as to generate simple ranking functions.

**Fitness calculation.** Under the CC framework, fitness of individuals is based on how well they cooperate with other populations. Figure 4 illustrates the fitness calculation for individual $i$ in EA2, one of the 4 EAs in Figure 3. First, individual $i$ and winners $w_1, w_3$ and $w_4$ selected in EA1, EA3 and EA4 from the previous generation, with the help of the assembler, are combined into a solution (right-hand side). Then, the evaluation measure, e.g., $NDCG@k$, for the combined solution is calculated using training data, and the resulting score is assigned to the individual $i$ as its fitness.

The fitness values of other individuals are calculated in the same manner. It seems that fitness calculation requires cooperation involving mutual dependency, which would make parallel execution of EAs infeasible. How-
ever, note that the cooperation is between two different generations. As shown in Figure 4, the individual \(i\) is from the current generation, while winners \(w_1, w_3\) and \(w_4\) are the best individuals from the previous generation. Thus, there is no mutual dependency and all EAs can perform fitness calculation in parallel.

4.6 Pseudocode

We have explained the main procedures of CCRank. Now we summarize them and present the pseudocode of CCRank in Algorithm 2.

**Algorithm 2: CCRank**

**Input**: Training set \(\mathcal{T}\), validation set \(\mathcal{V}\), maximum number of generations \(G\), number of populations \(N\)

**Output**: Ranking function \(f\)

1. Initialize \(\) 
2. for \(g \leftarrow 1\) to \(G\) do 
3. for \(i \leftarrow 1\) to \(N\) do 
4. \(\mathcal{P}_i^{(g)} \leftarrow \text{Evolve}(\mathcal{P}_i^{(g-1)})\) 
5. Update \((w_i, \mathcal{P}_i)\) 
6. end 
7. \(f^{(g)} \leftarrow \text{Combine}(w_1, \ldots, w_N, A)\) 
8. \(C \leftarrow C \cup \{f^{(g)}\}\) 
9. end 
10. \(s_\mathcal{T}(C) \leftarrow \text{Evaluate}(C, \mathcal{T})\) 
11. \(s_\mathcal{V}(C) \leftarrow \text{Evaluate}(C, \mathcal{V})\) 
12. \(f \leftarrow \text{Select}(s_\mathcal{T}(C), s_\mathcal{V}(C))\)

In Algorithm 2, \(\mathcal{P}_i^{(g)}\) denotes the \(i\)th population in the \(g\)th generation, \(w_i\) denotes the winner of \(i\)th population, \(A\) denotes an assembler, \(C\) denotes the candidate set, and \(s\) denotes the fitness function of individuals.

Line 1 performs initialization, including random generation of \(N\) populations, evaluation of each individual, and selection of initial winners.

Lines 2-9 show the whole evolution process from generation to generation. Specifically, lines 3-6 show one generation of the evolution process for all populations that execute in parallel. In particular, line 4 evolves individuals based on the previous generation, and line 5 updates the winner for each EA. Lines 7-8 generate candidate solutions.

Lines 10-11 calculate the performance measures \(s_\mathcal{T}(C)\) and \(s_\mathcal{V}(C)\) for the candidates using the training set \(\mathcal{T}\) and validation set \(\mathcal{V}\), based on which line 12 selects the ranking function \(f\) via Equation (1) (see Section 4.1).

4.7 Discussion

**Time complexity of non-parallel CCRank.** As shown in Section 4.5 and Figure 3, in each iteration, the evolution phase of CCRank involves four steps: population evolution, fitness calculation, winner selection, and solution combination.

In population evolution, generation of each individual (subtree) in EAs involves \(2^{d_i} - 1\) nodes, where \(d_i\) is the depth of individuals. In the cases of \(N\) EAs each maintaining \(L\) individuals, the time complexity of this step is \(O(2^{d}NL)\).

In fitness calculation, each individual and winners from other EAs are combined into a solution for evaluation. Decoding a solution into a ranking function involves \(2^d - 1\) nodes. The whole ordering process refers to \(mnk\) comparisons, where \(m\) is the number of queries, each associated with \(n\) documents, and the top-\(k\) (\(k \ll n\)) documents in each query are ordered to calculate the fitness of the solutions like \(NDCG@k\). Thus the time complexity of the fitness calculation step is \(O(2^dNLmnk)\) in the cases of \(N\) EAs.

Obviously, the time complexity of the winner selection and solution combination steps are \(O(NL)\) and \(O(N)\) respectively. Thus the time complexity of CCRank in each iteration is \(O(2^dNL + 2^dNLmn\log n + NL + N)\). Specifically, evaluation of some IR measures such as \(NDCG\) just needs to order top-\(k\) (\(k \ll n\)) documents for each query. In this case, the time complexity of CCRank in each iteration is \(O(2^dNL + 2^dNLmnk + NL + N)\).

**Speed-up of CCRank.** Firstly, CCRank does not always execute in parallel. In CCRank, the first three steps of the population evolution, fitness calculation and winner selection are performed in parallel. After that, it must suspend the parallel execution to perform combination in order to produce the candidate solution. Secondly, parallelization introduces some additional costs resulting from process and memory scheduling problems. For example, although the computation tasks of each process are equal thanks to same size of population in each EA, EAs still spend different amounts of time to evolve for a certain generation. Since the combination can start only after all EAs finish, the time CCRank spends on an generation is equal to the longest time that any EA can possibly spend for the generation. Let \(\epsilon\) be the cost resulting from the scheduling problem. The speed-up
ratio of CCRank can be formulated as follows:

\[ s = \frac{N (2^d L + 2^d Lmn \log n + L) + N}{(2^d L + 2^d Lmn \log n + L) + N + \epsilon}. \]

(3)

Specifically, with some top-k IR evaluation measures, the speed-up ratio of CCRank can be reformulated as follows:

\[ s' = \frac{N (2^d L + 2^d Lmnk + L) + N}{(2^d L + 2^d Lmnk + L) + N + \epsilon}. \]

(4)

5 Implementations of CCRank

We choose RankGP, RankIP and RankGDE, three evolutionary algorithms (EA)-based learning to rank algorithms to implement CCRank, correspondingly named CCRank-GP, CCRank-IP and CCRank-GDE.

5.1 CCRank-GP

CCRank-GP is constructed by directly implementing the function Evolve in Algorithm 2 as the evolution process of RankGP, where genetic programming (GP) is utilized to learn ranking functions with ranking-oriented features, constants and basic function operators (see Section 4.3).

Genetic programming was derived from genetic algorithm, and now becomes an important branch of evolutionary algorithms. The main difference between genetic programming and genetic algorithm is the representation of individuals. In genetic algorithm, an individual is represented as a sequence of numbers, while in genetic programming, an individual is represented as a tree structure.

Inspired by Darwinian evolutionary principles, genetic programming maintains a population of individuals (solutions) that evolve from generation to generation with three evolutionary operations: reproduction, crossover and mutation.

- Reproduction. Reproduction aims to maintain high-fitness individuals during evolution. Specifically, the reproduction operator selects a high-fitness individual \( i_p \) from current generation, and then reproduces an offspring \( i_o \) by cloning \( i_p \) for return.
- Crossover. Crossover aims to create individuals with higher fitness scores for the next generation based on current generation of individuals. Specifically, the crossover operator selects two high-fitness individuals \( i_p_1 \) and \( i_p_2 \) from current generation as parents, and then generates two offsprings \( i_o_1 \) and \( i_o_2 \) by combination of \( i_p_1 \) and \( i_p_2 \).
- Mutation. Mutation aims to create some variations of high-fitness individuals for the next generation based on current generation of individuals. Specifically, firstly the mutation operator selects one high-fitness individual \( i_p \) from the current generation as a parent, and then replaces a randomly selected subtree of \( i_p \) with a randomly generated one.

In evolution process, we utilize the roulette wheel selection strategy, where each individual is selected as a parent with a certain probability proportional to its fitness scores. In particular, the selection probability of individual \( i_s \) is:

\[ Pr(i_s) = \frac{\mathcal{F}(i_s)}{\sum_{i \in \mathcal{C}} \mathcal{F}(i)}, \]

(5)

where \( \mathcal{C} \) is the current generation of individuals, and \( \mathcal{F}(i) \) is the fitness function that returns a fitness score for each individual \( i \in \mathcal{C} \).

5.2 CCRank-IP

CCRank-IP is constructed by implementing the function Evolve in Algorithm 2 as the evolution process of RankIP, where immune programming (IP) is utilized to learn ranking functions.

Immune programming is an extension of immune algorithms, particularly the clonal selection algorithm, inspired by the biological immune systems or their principles and mechanisms.

Biological immune systems. Immune system can protect the organism against pathogens and eliminate malfunctioning cells. For vertebrates, the immune system is composed of a great variety of molecules, cells, and organs spread throughout the body, where various distributed elements perform complementary tasks without any central organs for control [51], [52].

All elements recognizable by the immune system are called antigens, including pathogens, malfunctioning cells, and healthy cells. The native cells, which originally belong to the organism and are harmless to its functions, are termed self or self antigens, while the disease-causing elements are named non-self or non-self antigens [25].

An immune system should be able to distinguish the harmless cells from the pathogens that causing diseases by lymphocytes, which can produce antibodies to recognize and bind to a certain type of antigens. Besides, for effective reaction to new pathogens and improvement in response to existing ones, the immune system should be capable of memory and learn [53]. After successful recognition, antibodies capable of binding with pathogen are cloned. Besides, a subpopulation of antibodies also undergo mutations, that provides the ability in recognition of both the pathogen itself and similar ones. Moreover, some of the mutated clones may have higher affinity. In the immune system, this process is usually called hypermutation due to a high rate of mutation.

Immune programming. Immune programming (IP) is an EA-based machine learning paradigm inspired by biological immune systems. In the IP algorithm, the antigen represents the given problem while the antibodies represent a set of candidate solutions. Similar to individuals in GP, antibodies in IP are encoded as trees. Each antibody is associated with an affinity score, indicating the fitness of the antibody to the antigen. The antibodies are generated at random initially, and evolve from generation to generation with three immune operations: replacement, cloning and hypermutation.
• Replacement. Replacement aims to replace low-affinity antibodies in current generation with randomly generated ones for next generation. In particular, a new antibody is generated randomly for return if a randomly generated number \( r \) is no greater than the replace rate \( p_r \), where \( 0 \leq r, p_r \leq 1 \).

• Cloning. Cloning aims to maintain high-affinity antibodies during evolution. First of all, an antibody \( i \) with affinity score of \( F(i) \) is selected from current generation. Given the clone rate \( p_c \) where \( 0 \leq p_c \leq 1 \), a new antibody is generated by cloning \( i \) for return with probability of \( p_c \) if a randomly generated number \( r \) is no greater than \( F(i) \).

• Hypermutation. Hypermutation aims to create some variations of high-affinity antibodies for the next generation based on current generation, provided the antibody \( i \) is the very antibody selected in the cloning steps. Given the mutation rate \( p_m \) where \( 0 \leq p_m \leq 1 \), the mutation operator walks through all of the nodes of \( i \) in certain order, and replaces each node with a new randomly generated value with a probability of \( \min \left( \frac{p_m}{F(i)}, 1 \right) \).

**Algorithm 3: Evolution in Immune Programming**

**Input:** Replace rate \( p_r \), clone rate \( p_c \), mutation rate \( p_m \), population size \( P \), current population \( C \) with affinity \( \mathcal{F} \)

**Output:** Next generation of population \( \mathcal{N} \)

1. \( \mathcal{N} \leftarrow \emptyset \)
2. while \( \text{size}(\mathcal{N}) < P \) do
3. \( r \leftarrow \text{Replacement}(C, p_r) \)
4. if \( r \neq \text{null} \) then
5. \( \mathcal{N} \leftarrow \mathcal{N} \cup \{r\} \)
6. else
7. \( i \leftarrow \text{GetGoodAntibody}(\mathcal{C}) \)
8. \( c \leftarrow \text{Cloning}(p_c, i, \mathcal{F}(i)) \)
9. if \( c \neq \text{null} \) then
10. \( \mathcal{N} \leftarrow \mathcal{N} \cup \{c\} \)
11. else
12. \( m \leftarrow \text{Hypermutation}(p_c, i, \mathcal{F}(c)) \)
13. \( \mathcal{N} \leftarrow \mathcal{N} \cup \{m\} \)
14. end
15. end
16. end

The pseudocode of immune programming is shown in Algorithm 3. First of all, line 1 initializes the next generation of individuals \( \mathcal{N} \) as \( \emptyset \), and then lines 2-16 show the whole evolution process from the current generation \( \mathcal{C} \) to the next generation \( \mathcal{N} \). In particular, firstly the replacement operator executes with replace rate \( p_r \) (line 3). If a new antibody \( r \) is generated successfully, it will be put into \( \mathcal{N} \) (lines 4-5). Otherwise, if \( r \) cannot be generated due to \( p_r \) (line 6), a high affinity antibody \( i \) is selected and considered for cloning with clone rate \( p_c \) (lines 7-8). If \( i \) is generated successfully, it will be put into \( \mathcal{N} \) (lines 9-10). Otherwise, if \( i \) cannot be cloned due to \( p_c \) (line 11), it is mutated and put into \( \mathcal{N} \) (lines 12-13).

**CCRank-IP.** Based on immune programming, RankIP can learn ranking functions with ranking-oriented features, constants and basic function operators, and CCRank-IP is constructed by parallelizing RankIP with cooperative coevolution.

However, RankIP and CCRank-IP cannot use IR evaluation measures such as MAP and NDCG@k directly as affinity functions. In immune programming, the affinity scores should be scaled between 0 and 1 because they are used as probabilities in cloning and hypermutation. If all of the them are close to 0, most of the antibodies would be generated randomly with replacement, and thus immune programming can be thought of as a randomized algorithm without any heuristics. On the other side, if all of the the affinity scores are close to 1, most of the antibodies would be generated with cloning, resulting in early convergence.

For the learning to rank problem, the evaluation scores of the most learned ranking functions are too small, generally less than 0.5 in MAP and NDCG@k. Thus RankIP and CCRank-IP use a normalized IR evaluation measures as affinity functions, which shown as follows:

\[
f(x) = \log_b \left( 1 + (b - 1) \times \frac{x}{s_e} \right),
\]

where \( x \) is an IR evaluation measure such as MAP or NDCG@k, and \( s_e \) is the expected affinity score where \( 0 \leq s_e \leq 1 \). Wang et al. [23] demonstrates that the performance of RankIP is robust to \( s_e \).

Equation (6) is a convex function that assigns higher affinity scores for antibodies than linear scale-up. In each generation, generally most of antibodies (candidate ranking functions) perform poor and achieve unsatisfactory evaluation scores, e.g., far smaller than 0.5 in NDCG@k. However, they may contain quality components. Assigning relatively higher affinity scores to them can increase their survival probabilities during evolution, avoiding too early convergence to local optiums.

**5.3 CCRank-GDE**

CCRank-GDE is constructed by implementing the function Evolve in Algorithm 2 as the evolution process of RankGDE, where the tree-based geometric differential evolution (GDE) [26], [27] is utilized to learn ranking functions.

Differential evolution (DE) [45], [46] is a population-based stochastic function minimization method, which has robust performance over a variety of continuous optimization problems. A DE algorithm maintains a population of vectors, each representing a solution to the given problem in Euclidean space. In CCRank-GDE, tree-based GDE [27] uses trees to encode the solutions (ranking models) to the learning to rank problem in the generic metric space [54].
Let $T_1$ and $T_2$ be two trees, where $p$ and $q$ are their roots respectively. Firstly, GDE completes them with NULLs to make sure they have a same structure. Then their distance $\text{dis}(T_1, T_2)$ can be evaluated as follows:

$$\text{dis}(T_1, T_2) = \begin{cases} d(p, q), & \text{if } T_1, T_2 \text{ have no offsprings} \\ d(p, q) + \frac{1}{K} \sum_{i=1}^{K} \text{dis}(s_l, t_l), & \text{otherwise} \end{cases}$$

where $s_l$ and $t_l$ are the subtrees of $T_1$ and $T_2$, $K$ is a constant where $K \geq 1$, which is used to $K$ times signify the distance between the nodes at the depth $r$ than at the depth $r+1$, and the distance $d(p, q)$ between two nodes $p$ and $q$ is defined as follows:

$$d(p, q) = \begin{cases} 0, & \text{if } p = q \\ \frac{C(p-q)}{\max_{v \in A_1} |v-w|}, & \text{if } p, q \in A_1 \\ \delta(i, j), & \text{otherwise when } p \in A_i, q \in A_j \end{cases}$$

where $A_0$, $A_1$, $A_2$ and $A_3$ are 4 categories of the nodes in the tree-based representations. In particular, $A_0$ is the set only containing NULL, $A_1$ is the set of constants, $A_2$ is the set of basic functions (see Table 1), and $A_3$ is the set of features. The function $\delta(i, j)$ is defined as follows:

$$\begin{align*}
\delta(0, i) &= \delta(0, 0) = 5, \quad i \in \{1, 2, 3\} \\
\delta(1, 2) &= \delta(2, 1) = 2 \\
\delta(i, i) &= 1, \quad i \in \{2, 3\} \\
\delta(3, i) &= \delta(i, 3) = 3, \quad i \in \{1, 2\}
\end{align*}$$

In tree-based GDE algorithms, the trees evolve iteratively with homologous crossover and recombinations.

- Homologous crossover [27]. Homologous crossover aims to produce a mutant tree $T_M$ based on two parent trees $T_A$ and $T_B$. Given two positive weights $w_{AM}$ and $w_{BM}$ where $w_{AM} + w_{BM} = 1$, the homologous crossover is performed on the common region of the parent trees, i.e., the largest rooted region where two parent trees $T_A$ and $T_B$ share same topology. The mutant tree $T_M$ is generated using a crossover mask on the common region of $T_A$ and $T_B$ such that the nodes of $T_A$ and $T_B$ in the common region appear in the crossover mask with the probability of $w_{AM}$ and $w_{BM}$ respectively.

- Homologous recombination. Homologous recombination aims to produce an offspring tree $T_O$ based on the mutant tree $T_M$ and another parent tree $T_C$. Firstly, GDE assigns the distance between $T_C$ and $T_O$ by $\text{dis}(T_C, T_O) = \frac{w_{CM}}{w_{MO}} \text{dis}(T_C, T_M)$ where $w_{CM}$ and $w_{MO}$ are positive weight parameters and $w_{CM} + w_{MO} = 1$, and assigns the probability $p = \frac{\text{dis}(T_M, T_O)}{1 - \text{dis}(T_C, T_M)}$. Then GDE initializes $T_O = T_M$. Let $S_M$ and $S_C$ are subtrees of $T_M$ and $T_C$ sharing a same structure. For each node $n_M(i)$ in $S_M$ and $n_C(i)$ in $S_C$, GDE assigns a random node for the same position to $T_O$ with a probability $p$ if $n_M(i) = n_C(i)$. Then the generated tree $T_O$ is returned for the next generation.

The pseudocode of GDE is shown in Algorithm 4. In the beginning, line 1 initializes the next generation of individuals $N$ as $\emptyset$, and then lines 2-13 show the whole evolution process from the current generation $C$ to the next generation $N$. In particular, for each tree $T$ in the current generation, line 4 firstly selects two parent trees $T_A$ and $T_B$. Then line 5 constructs the mutant tree $T_M$ by homologous crossover on $T_A$ and $T_B$. With $T_M$ and $T$, line 6 generates an offspring tree $T_O$ by homologous recombination. Finally lines 7-11 put $T_O$ or $T$ into $N$ based on their fitness scores.

### 6 Experiments

We conducted two series of experiments using benchmark datasets to evaluate the efficiency and accuracy of CCRank.

#### 6.1 Methodology

**Datasets.** We used LETOR 4.0 and MSLR-WEB30K benchmark datasets to demonstrate the promise of CCRank. Each data set has been partitioned into five parts in order to conduct 5-fold cross validation. For each fold, three parts are used for training, one part for validation, and the remaining part for test.

LETOR 4.0 uses the Gov2 Web page collection and two query sets from Million Query track of TREC 2007 and TREC 2008, called MQ2007 and MQ2008. There are about 1,700 queries with 69,623 instances in MQ2007 and about 800 queries with 15,211 instances in MQ2008. The relevance degrees of documents with respect to the queries are judged on three levels: 2 (definitely relevant), 1 (partially relevant), and 0 (not relevant). For each query-document pair in LETOR 4.0, there are in total 46 features.

MSLR-WEB30K collects 30,000 queries with ~19 million instances from a retired labeling set of Microsoft Bing (http://www.bing.com/), a commercial web search engine. The relevance degrees of documents with respect...
speed-up ratios are 1.61, 2.92, 4.84 and 9.80 respectively for LETOR 4.0 with 800 queries in the cases of 2, 4, 8, and 16 processors, and these numbers are 1.74, 3.88, 7.87 and 11.12 respectively for MSLR-WEB30K with 30,000 queries. (2) The efficiency gain of CCRank is especially significant with larger training dataset. For example, the speed-up ratio is about 12 in the case of 16 processors for MSLR-WEB30K, while this number is only less than 10 for LETOR. (3) The efficiency gains of CCRank are stable with different numbers of instances from a same dataset. For example, for LETOR 4.0, the speed-up ratios are 9.61, 9.20 and 9.80 when 25%, 50% and 100% data are used respectively in the cases of 16 processor, and their standard deviation is only 0.31. For MSLR-WEB30K, these numbers are 12.17, 11.88 and 11.12 respectively, and their standard deviation is 0.54.

Equations (3) and (4) can be used to explain the three observations. Firstly, since \( s, s' < N \), obviously CCRank cannot achieve a linear speed up via parallelization. Secondly, the executive time for fitness calculation grows sharply with larger training data, resulting from greater depths of individuals \( (d \text{ and } d_i) \) and number of instances \( (mn) \). On the other side, the time for other steps are relatively stable. Since fitness calculation is the most time-consuming step in the evolution phase, the speed-up of CCRank is more significant via parallelization of fitness calculation. Thirdly, with different numbers of instances from a same dataset, the speed-up functions differ slightly. Although larger amount of instances may lead to a higher speed-up ratio, it also increases the complexity of process and memory scheduling, which may reduce the speed-up ratio, resulting from a greater value of \( \epsilon \) in Equations (3) and (4). Thus, in this case, the efficiency gains of CCRank are relatively stable.

### 6.3 Accuracy

In the second series of experiments, we demonstrated the accuracy of CCRank on LETOR 4.0 and MSLR-WEB30K.

**Evaluation measures.** We use three standard ranking accuracy metrics to evaluate the rank functions generated by learning to rank algorithms: precision at \( k \) (\( P@k \)), mean average precision (\( MAP \)) and normalized discount cumulative gain (\( NDCG@k \)). \( P@k \) measures the accuracy within the top \( k \) results of the returned ranked list for a query:

\[
P@k = \frac{\# \text{ of relevant docs in top } k \text{ results}}{k}.
\]

\( MAP \) takes the mean of the average precision values over all queries, where the average precision (\( AP \)) for each query is defined as the average of the \( P@k \) values for all relevant documents:

\[
AP = \frac{\sum_{k=1}^{N} \left( P@k \times \text{rel}(k) \right)}{\# \text{ relevant docs for this query}}.
\]

where \( \text{rel}(k) \) is a binary function mapping a document to either 1 (relevant) or 0 (irrelevant). Note that \( P@k \) and
MAP can only handle cases with binary judgment, relevant or irrelevant. Recently, a new evaluation measure \( NDCG@k \) [55] has been proposed to handle multiple levels of relevance:
\[
NDCG@k = \sum_{j=1}^{k} \frac{2^{rel(j)} - 1}{\log(j)}, \quad \text{if } j = 1
\]
\[
NDCG@k = \sum_{j=1}^{k} \frac{2^{rel(j)} - 1}{\log(j)}, \quad \text{if } j > 1,
\]
where \( rel(j) \) is the integer rating of the \( j^{th} \) document, and the normalization constant \( Z_k \) is chosen such that the perfect list gets a \( NDCG \) score of 1.

**Comparison partners.** We compared CCRank-IP, CCRank-GP and CCRank-GDE with the state-of-the-art learning to rank algorithms, including RankIP [23], AdaRank [4], RankBoost [2], RankSVM [3], ListNet [1] and LambdaMART [6]. In particular, CCRank-IP is the parallelization of RankIP, and a direct comparison of the two can provide valuable and irreplaceable insights.

**Performance.** Table 2 shows the accuracy comparison under the \( MAP \) measure. For MQ2007, CCRank-IP and RankBoost shared the same performance, which is only worse than LambdaMART and the difference is very subtle. For MQ2008, CCRank-IP outperformed all other algorithms. Although CCRank-GDE and CCRank-GP failed to achieve improvement in accuracy, it can still be comparable to the state-of-the-art algorithms. For MSLRWEB30K, CCRank-IP, CCRank-GDE and CCRank-GP can also achieve comparable performance to the comparison partners. For example, the performance of CCRank-GDE is slightly better than ListNet and RankBoost.

Although the performance of CCRank-GP is the worst, it is merely 0.76% worse than that of ListNet.

Figures 7, 8 and 9 show the accuracy comparison under the \( P@1 \sim 5 \) and \( NDCG@1 \sim 5 \) measures. The results are consistent with the ones under \( MAP \). For MQ2007 and MQ2008, CCRank-IP is among the best for both measures, while CCRank-GDE and CCRank-GP are comparable to the comparison partners. For MSLRWEB30K, all of our proposed CCRank algorithms are comparable to the comparison partners. Besides, compared with RankIP, the performance of CCRank-IP is even slightly better than RankIP. That is because the complexity of the models generated by CCRank-IP are lower than those by RankIP resulting from the simple structure of the assembler, which is only composed of the operator +. According to Occam’s razor, simpler models might be less overfitting, leading to higher test performance.

### 7 Conclusion

In this paper we proposed CCRank, a parallel learning to rank framework based on cooperative coevolution, aiming to significantly improve the learning efficiency while maintain accuracy. Furthermore, we implemented CCRank with three EA-based learning to rank algorithms based on genetic programming, immune programming and geometric differential evolution respectively. We experimentally compared CCRank with state-of-the-art algorithms on benchmark datasets, demonstrating the gains of CCRank in efficiency and accuracy.

For future work, we plan to extend CCRank in several directions. One direction is to further explore our...
parallel CC framework by incorporating some recently proposed evolutionary algorithms, such as Differential Evolution [19] and Particle Swarm Optimization [20]. Another direction is to organize other machine learning algorithms, e.g., SVM and Neural Network, to work in a collaborative manner for the learning to rank problem, where we perform each algorithm with a subset of features to train a sub-model, and then assemble them into a complete ranking model for predicting relevance scores. Last but not least, our current parallel CC framework has demonstrated significant speed-up, but not scale-up. We plan to investigate how to further boost efficiency by taking full advantage of parallelization. For this purpose, more economic and sophisticated cooperation schemes need to be considered.

ACKNOWLEDGMENTS

The authors would like to thank the editor and the anonymous reviewers for their constructive comments and suggestions. This work is supported in part by the Academy of Finland (268078), the National Science Foundation of China (71402083, 61272240 and 71171122), and the Academy of Finland (268078), the National Science Foundation of Shandong Province of China (BS2012DX012).

REFERENCES


Yun Wu is currently an Engineer at Software Development Center, Agricultural Bank of China. He received M.Eng. and B.Sc. in Computer Science from Shandong University, China, in 2014 and 2011 respectively. He was an Intern at Baidu.com. His research interests include information retrieval, data mining, and machine learning.

Byron J. Gao received Ph.D. and B.Sc. in Computer Science from Simon Fraser University, Canada, in 2007 and 2003 respectively. Currently he is an Associate Professor of computer science at Texas State University. He joined Texas State University in 2008. Before that, he was a postdoctoral fellow at the Database Lab of University of Wisconsin-Madison in 2007-2008. His research spans several related fields including data mining, databases, information retrieval, and bioinformatics.

Ke Wang received Ph.D from Georgia Institute of Technology. He is currently a professor at School of Computing Science, Simon Fraser University. Before joining Simon Fraser, he was an associate professor at National University of Singapore. His research interests include database technology, data mining and knowledge discovery, with emphasis on massive datasets, graph and network data, and data privacy. He has published in database, information retrieval, and data mining conferences, including SIGMOD, SIGIR, PODS, VLDB, ICDE, EDBT, SIGKDD, SDM and ICDM.

He is currently an associate editor of the ACM TKDD journal and he was an associate editor of the IEEE TKDE journal, an editorial board member for DMKD, and a PC co-chair for SDM 2008. He is a general co-chair for SDM 2015 and 2016.

Hady W. Lauw received his Ph.D. degree from Nanyang Technological University, Singapore. He is currently an Assistant Professor at School of Information Systems, Singapore Management University (SMU). Before joining SMU, he was a Scientist in the Data Mining department of A*STAR’s Institute for Infocomm Research. Earlier on, he was a postdoctoral researcher with Microsoft Research in United States. His research interests include social network mining and Web mining.

Jun Ma is a full professor of Shandong University. He received B.Sc., M.Sc. and Ph.D. from Shandong University in China, Ibaraki University and Kyushu University in Japan respectively. He was a senior researcher at the department of computer science of Ibaraki University in 1994 and German National Research Center for Information Technology from 2000 to 2003. His research interests include information retrieval, data mining and algorithms.