An Optimal Algorithm for Finding Champions in Tournament Graphs

Lorenzo Beretta, Franco Maria Nardini, Roberto Trani, and Rossano Venturini

Abstract—A tournament graph is a complete directed graph, which can be used to model a round-robin tournament between n players. In this paper, we address the problem of finding a champion of the tournament, also known as Copeland winner, which is a player that wins the highest number of matches. In detail, we aim to investigate algorithms that find the champion by playing a low number of matches. Solving this problem allows us to speed up several Information Retrieval and Recommender System applications, including question answering, conversational search, etc. Indeed, these applications often search for the champion inducing a round-robin tournament among the players by employing a machine learning model to estimate who wins each pairwise comparison. Our contribution, thus, allows finding the champion by performing a low number of model inferences. We prove that any deterministic or randomized algorithm finding a champion with constant success probability requires $\Omega(\ell n)$ comparisons, where ℓ is the number of matches lost by the champion. We then present an asymptotically-optimal deterministic algorithm matching this lower bound without knowing ℓ, and we extend our analysis to three variants of the problem. Lastly, we conduct a comprehensive experimental assessment of the proposed algorithms on a question answering task on public data. Results show that our proposed algorithms speed up the retrieval of the champion up to $13\times$ with respect to the state-of-the-art algorithm that perform the full tournament.

Index Terms—Tournament Graph, Round-Robin Tournament, Copeland Winner, Minimum Selection, Pairwise Ranking.

◆

1 INTRODUCTION

A *tournament graph* is a complete directed graph $T = (V, E)$, where V and E are the sets of nodes and arcs, respectively [29]. The tournament graph can be used to model a round-robin tournament between n players, where each player plays a match with any other player. The orientation of an arc tells the winner of the match, i.e., we have the arc $(u, v) \in E$ iff u beats v in their match. In the following, we call *arc lookup* or *arc unfold* the operation of looking at the direction of an arc between two nodes.

We address the problem of finding a champion of the tournament, also known as *Copeland winner* [12], which is a vertex in V with the maximum out-degree, i.e., a player that wins the highest number of matches. Our goal is to find a champion by minimizing the number of arc lookups, i.e., the number of matches played. Note that a tournament graph may have more than one champion. In this case, we aim at finding any of them, even if all the proposed algorithms are able to find all of them without increasing the complexity.

- Lorenzo Beretta is with the Basic Algorithms Research Copenhagen (BARC), University of Copenhagen. E-mail:beretta@di.ku.dk
- Franco Maria Nardini and Roberto Trani are with the National Research Council of Italy. E-mail: {francomaria.nardini, roberto.trani}@isti.cnr.it
- Rossano Venturini is with the Department of Computer Science, University of Pisa. E-mail:rossano.venturini@unipi.it

If the tournament is transitive—whenever u wins against v and v wins against w, then u wins against w—we can trivially identify the unique tournament champion with $\Theta(n)$ arc lookups. Indeed, the champion is the only vertex that wins all its matches and, thus, we can perform a *knockout tournament* where the loser of any match is immediately eliminated. However, finding the champion of general tournament graphs requires $\Omega(n^2)$ arc lookups [16], and thus, there is nothing better to do than to play all the matches. This means that the structure of the underlying tournament graph heavily impacts the complexity of the problem.

In this article, we parametrize the problem with the number ℓ of matches lost by the champion and we investigate efficient algorithms that find the champion by performing a number of arc lookups proportional to ℓ. This parametrization is motivated by many applications in Information Retrieval and Recommender Systems that exploit pairwise machine learning (ML) models. These models compare a pair of candidate players at a time to estimate who wins the match. The final champion of the tournament is the player winning the highest number of pairwise comparisons of the all-vs-all tournament induced by the machine-learned model [1], [27]. The parametrization we introduce is motivated by the fact that, nowadays, it is possible to design accurate pairwise models that achieve a low error rate in the estimation of the matches played by the champion. For this reason, we expect a low number of matches ℓ lost by the champion, hence a quasi-linear number of arc lookups is required by our algorithms to find it. This compares with the quadratic number of lookups needed by the previously known algorithms [16]. For this reason, this paper proposes efficient algorithms to find the tournament champion by performing the (asymptotically) minimum number of calls to the machine learning model, i.e., arc lookups, needed

to solve this problem. A more detailed description of the application scenarios is reported at the end of this section.

Our Contributions

The novel contributions of this article are the following:

- we introduce an asymptotically-optimal deterministic algorithm that finds the champion by employing $O(\ell n)$ vertex comparisons, where ℓ is the minimum number of matches lost by any player. Moreover, we prove that $\Omega(\ell n)$ comparisons are necessary, even for randomized algorithms, to obtain a correct answer with any constant probability. It is worth noticing that we match a randomized lower bound with a deterministic algorithm, showing that randomization does not give any advantage to this problem.
- We extend our result to three strictly-related problems. First, we show how to retrieve all top-k players in time $O(\ell_k n)$, where ℓ_k is the number of matches lost by the k-th best player. Second, we consider a model of computation in which we are allowed to play a batch of B matches in parallel, and we design an algorithm that achieves optimal speedup with respect to the sequential version and it finds the champion by performing $O(\frac{\ell n}{B} + \ell \log B)$ arc lookups. This is useful in practice because pairwise comparisons can be batched when the inference is done on novel computing platforms like, for example, GPUs. Third, we generalize the tournament problem in a probabilistic framework, where each arc $(u, v) \in E$ is labeled with the likelihood that u wins against v. These probabilities can be interpreted as the confidence of the machine learning model about the outcome of the comparison. In this setting, we define the champion as the player that minimizes the expected number of matches lost and we introduce an algorithm to find all champions in time $\Theta(\ell n)$, where ℓ is the expected number of matches lost by the champion.
- We provide a comprehensive experimental assessment of the proposed algorithms. We evaluate their performance in terms of running time and number of comparisons against a baseline that perform all the possible pairwise comparisons between players. We focus our attention on a Question Answering task that asks to find the most relevant textual answer to a given question provided by a user [28]. Results show that our proposed algorithms allow us to speed up the identification of the correct answer of up to $13\times$ with respect to methods that play the full tournament.

Application Scenarios

Our investigation is motivated by many application scenarios involving the efficient selection of the most relevant result from a pool of candidates, also known as top-1 retrieval. It is a crucial task in many Information Retrieval and Recommender System applications including Web ad-hoc search [5], question answering [17], conversational search [24], etc. A recent example in this line is conversational assistants. These devices, such as Siri, Google Assistant, and Alexa, are becoming very popular nowadays. They work by exploiting a new way of interaction with the user, where the latter interacts by asking a question and the former provides her the answer with the highest relevance with respect to the question. Conversational assistants introduce a *paradigm shift* in information retrieval as they change the way users submit their information needs to the information retrieval system, i.e., using spoken words and not textual queries. Moreover, since the new paradigm employs a conversation as a means of interaction, only one result is provided to the user as an answer to her question. As a consequence, the precision in the identification of the only answer to return is now of paramount importance to build an effective conversational system.

State-of-the-art solutions for solving the top-1 retrieval task rely on machine learning techniques [23], to select the answer with the highest relevance. The selection of the most relevant result can be addressed in two different ways: *i)* by exploiting machine-learned techniques such as λMART [32], which are based on univariate scoring functions that individually estimate one candidate result at a time, to select the candidate achieving the highest relevance score; *ii)* by employing pairwise Learning-to-Rank techniques such as DUOBERT [27], which are based on bivariate scoring functions that estimate a pair of candidate results at a time, e.g., a binary judgment stating which of the results is more relevant, to select the candidate achieving the highest sum of pairwise scores of an all-vs-all tournament. While the former approach exploits only the information of a single result at a time for computing the ranking score, the latter approach is potentially more powerful because it exploits the information of two candidates at a time for computing the outcome of the tournament. However, the latter approach, although effective, is more expensive than the former one as it performs a quadratic number of comparisons to score all pairs of candidate results, thus making pairwise approaches unappealing in scenarios with tight time constraints. Here is where our research is beneficial as we define algorithmic approaches that allow reducing the number of comparisons performed by the pairwise model to select the most relevant results thus speeding up the whole selection process.

The rest of the article is structured as follows: Section 2 discusses the related work while Section 3 provides a detailed analysis of the problem complexity, and Section 4 presents an efficient algorithm to solve it. Moreover, Section 5 discusses three variants of the algorithm that solve three extensions of the original problem. Finally, Section 6 presents a comprehensive analysis of proposed algorithms in a information retrieval (ad-hoc search) scenario, and Section 7 concludes the work.

2 RELATED WORK

Tournament graphs are a well-known model that has been applied to several different areas such as sociology, psychology, statistics, and computer science. Examples of applications are round-robin tournaments, paired-comparison experiments, majority voting, communication networks, etc. [9], [19], [21], [25], [29]. In this area, we identify two different research lines. The first one aims at finding *the tournament winner*, while the second one aims at *ranking the list of candidates* using pairwise approaches. Given a ranking

of candidates, we can easily define the champion as the top-1 element of a the global ranking, therefore the two tasks are related with each other. In this section, we describe the most important results concerning these two problems.

According to previous works [9], [21], [25], there is no unique definition of the notion of a tournament winner. Nevertheless, all of them agree on defining the winner whenever there is a candidate, called *Condorcet winner*, which beats all the others. Different definitions of winner require different complexities of the algorithms used to identify it. The easiest case to consider appears when T is a *transitive tournament graph*, i.e., a directed acyclic graph, since it is trivial to find the Condorcet winner in linear time by performing a knock-out tournament where the loser of any match is immediately eliminated. Instead, for a general tournament T, the complexity of finding a winner is much higher and strictly depends on the definition of winner.

A winner as defined by Banks [6] is the Condorcet winner of a maximal transitive sub-tournament of T. As there may be several of these sub-tournaments, the *Banks solution* is the set of all these winners. The problem of finding just one winner can be computed in $\Theta(n^2)$ arc lookups, while finding all of them is a $\mathcal{NP}$-hard problem [19].

Slater [31] defined the winner starting from a ranking of candidates. He defined a *Slater solution* to be a total order $\prec$ on vertices that minimizes the number of mis-ordered pairs of vertices, where a pair (u, v) is mis-ordered if u beats v and $u \prec v$. The champion is then defined as the maximum element with respect to $\prec$. However, the computation of the Slater solution is $\mathcal{NP}$-hard as it reduces from the *Feedback Arc Set Problem* [11].

Ailon *et al.* [2], [3] provide a bound to the error achieved by the Quicksort algorithm when used to approximate a Slater solution. The error is defined as the number of misordered pairs of vertices. Ailon *et al.* show that the expected error is at most two times the best possible error. It is apparent that the proposed algorithm requires $\Omega(n \log n)$ arc lookups with high probability. Even though the overall approximation is good, this algorithm fails in finding a champion w every time one of the Quicksort pivots beats w, hence it is not suitable for our purposes.

The results by Shen *et al.* [30] and Ajtai *et al.* [4] provide a ranking based on the definition of *king*. The vertex u is a king if for every vertex v there is a directed path from u to v of length at most 2 in T. The ranking algorithm by Jian *et al.* [30] finds a sorted sequences of vertices $u_1, u_2, \ldots, u_n$ such that for every i 1) u_i beast u_{i+1}, and 2) u_i is a king in the sub-tournament induced by the items $u_i, u_{i+1}, \ldots, u_n$. The authors provide a $O(n^{3/2})$ deterministic algorithm to compute this sequence. On the flip side, a $\Omega(n^{4/3})$ deterministic lower bound for the retrieval of a single king holds. In addition, quicksort produces such a sequence in $O(n \log n)$ comparisons w.h.p. and quickselect retrieves a king in expected linear time. To date the deterministic complexity of finding a king in a tournament is still unknown, however attempts at understanding the problem proceed relentless [8]. Unfortunately, the definition of king is weaker than the one of Copeland winner. Indeed, the latter implies the former [29], and it is possible to construct tournaments in which every vertex is a king. Thus the definition of king does not help us in the identification of the best candidate.

A prolific research line studies the ranking problem under *persistent comparison errors* [10], [13], [14], [20]. This task deals with queries affected by random noise in a scenario where comparison errors are persistent. In this setting, we consider the set of vertices as equipped with a transitive order $\prec$, and every arc of the tournament as the result of a noisy comparison between two items. The answer associated to the comparison (u, v) is consistent with the transitive order $\prec$ with probability $p \approx 1$ and inconsistent with probability $1 - p \approx 0$. All comparisons are independent. By defining the dislocation of u as the difference between its real rank and the rank assigned by an algorithm, Geissmann *et al.* [13] proved that every algorithm produces a ranking with maximum dislocation $\Omega(\log n)$ and total dislocation $\Omega(n)$. A recent work by Geissmann *et al.* [14] settles the problem, matching both lower bounds in $O(n \log n)$ time. Unfortunately, this model does not produce a strong enough guarantee on the quality of the champion, that is only known to be within the top $O(\log n)$ candidates of the original ranking.

A line of work on *non-persistent comparison errors* studies noisy comparisons under the assumption that every comparison can be queried more than once and the results are all independent. Recently, progress has been made on approximate selection [18], and more notably on minimum-selection [22] that is exactly the problem we tackle in this paper, with a different model for noise. In fact, Leucci and Liu [22] just settled the complexity of minimum-selection in the non-persistent comparison error model.

There are several other notions of winner, and most of them can be computed in polynomial time. We refer to Hudry [19] for a complete survey on this topic. The definition used in this paper is the one given by Copeland [12], called *Copeland solution*, where we rank vertices according to the number of matches they win, and a champion is the candidate winning the most matches. As we already mentioned, the Copeland solution requires $\Omega(n^2)$ arc lookups and there is a trivial algorithm to match it [16]. However, Geissmann *et al.* [15] considered a model, similar to the aforementioned persistent comparison errors model, in which errors are no longer stochastic but their total number is bounded. They fix an upper-bound e to the total number of errors and they propose an algorithm to find the Copeland winner of the resulting tournament in $O(n\sqrt{e})$ comparisons and time.

Advancements over Previous Work

In this article, we advance the state of the art by reporting improvements over the result by Geissmann *et al.* [15]. In particular, we propose an algorithm that finds the Copeland winner in $\Theta(\ell n)$ time and comparisons, where ℓ is the minimum number of matches lost by any player, hence $\ell \leq \sqrt{e}$ meaning that our algorithm is at least asymptotically as fast as Geissmann *et al.* [15]. It is worth noting that in our use case ℓ is very small, and so this parameterization is particularly insightful. Moreover, our novel algorithm presented in Section 4 is oblivious with respect to ℓ, while the algorithm by Geissmann *et al.* [15] assume to know e in advance. Finally, we provide a *randomized* lower bound that matches the complexity of our *deterministic* and simple algorithm (Section 3.2). One last remarkable contribution is

the extension of our algorithm to work when comparisons can be performed in batches and we achieve virtually no asymptotic overhead with respect to perfect parallelism (Section 5.3).

3 LOWER BOUNDS

In this section, we prove the lower bound of the Copeland winner problem. An adversarial argument is used by Gutin *et al.* [16] to prove that finding a champion requires $\Omega\left(n^2\right)$ arc lookups. Therefore, the trivial algorithm that finds a champion by performing all the possible matches is optimal in general. The problem is indeed much more interesting if we parameterize it with ℓ, the number of matches lost by the champion. Note that ℓ is unknown to the algorithm. The goal of this section is to prove that $\Omega(\ell n)$ arc lookups are necessary to find a champion. We first show that this bound applies to deterministic algorithms. Then we generalize it to the class of "Monte Carlo" randomized algorithms that are allowed to return an incorrect answer with a fixed positive probability. The latter result clearly implies the former. However, for pedagogical reasons we report them in increasing order of difficulty.

3.1 Deterministic Lower Bound

The following theorem shows that any deterministic algorithm employs $\Omega(\ell n)$ arc lookups to find a champion.

Theorem 3.1. *Any deterministic algorithm that finds a champion in a tournament graph T with n vertices and with ℓ matches lost by the champion requires $\Omega(\ell n)$ arc lookups.*

Proof. The lower bound is proved by using an adversarial argument. Assume that an algorithm claims that a vertex u, losing ℓ matches, is a champion by performing $\frac{1}{2}\ell(n-1)$ arc lookups. There must exist a node v such that the algorithm has performed less than ℓ lookups of arcs incident to v. We thus can let the algorithm be incorrect by adversarially setting v as the winner of those matches, so that v wins more matches than u. In other words any correct algorithm, claiming that a vertex u is a champion with ℓ matches lost, must be able to certificate its answer by showing: 1) a list of $n - 1 - \ell$ matches won by u and 2) a list of ℓ matches lost by any other vertex v. □

3.2 Randomized Lower Bound

We just proved that no deterministic algorithm can perform $o(\ell n)$ arc lookups and output a correctness certificate. Now we extend such a non-existence result to any randomized algorithm, which is allowed to be wrong with a fixed probability. This section is devoted to prove the following theorem stating that it does not exist a Monte Carlo algorithm that finds the Copeland winner with $o(\ell n)$ arc lookups.

Theorem 3.2. *Given a tournament T with n vertices and with ℓ matches lost by the champion, it does not exist a randomized algorithm that performs $o(\ell n)$ arc lookups and outputs the Copeland winner of T with fixed positive probability.*

To prove the theorem above, we need to define the auxiliary problem below and operate a reduction.

Definition 3.3 (Anomalous Row Problem). *Given a matrix $M \in \mathbb{F}_2^{k \times m}$ such that every row but one presents $k + 1$ zeroes and the remaining one presents k zeroes, find the k-zeroes row.*

We will see that the anomalous row problem is not harder than the problem of finding the Copeland winner: technically we will show a reduction between these two problems. Moreover, proving a randomized lower bound for the anomalous row problem turns out to be easier.

The next lemma bounds from below the number of M's entries that must be probed in order to solve the anomalous row problem. This bound is strictly related to Theorem 3.2, as we will see shortly.

Lemma 3.4. *It does not exist a randomized algorithm that solves the anomalous row problem (Definition 3.3) by probing $o(km)$ cells of the input matrix M and returns the correct answer with fixed positive probability.*

To ease the discussion, we defer the proof of Lemma 3.4 to the end of this section. First, we show that if there exists an algorithm violating Theorem 3.2 then we can design an algorithm that violates Lemma 3.4. Thus, proving Lemma 3.4 is sufficient to prove Theorem 3.2.

Given an instance of the anomalous row problem, $M \in \mathbb{F}_2^{k \times m}$, we can assume that k and m are odd and $m > 3k$. Indeed, if this is not the case, it is sufficient to add a dummy row containing $k + 1$ zeroes and several dummy columns containing only ones. It is apparent that this modification preserves both the k-zeroes row and the asymptotic complexity. Then, we construct a tournament having $n = k + m$ players and adjacency matrix

$$A = \left[\begin{array}{cc} B & M \\ \widetilde{M}^T & C \end{array} \right] \in \mathbb{F}_2^{n \times n}$$

where $B \in \mathbb{F}_2^{k \times k}$ and $C \in \mathbb{F}_2^{m \times m}$ are the adjacency matrices of regular tournaments[1] and $\widetilde{M}$ is the *complementary matrix* of M, meaning that $\widetilde{M}_{i,j} = 1 - M_{i,j}$.

We can easily prove that the champion is among the first k players and loses exactly $\ell = (3k - 1)/2$ matches. In fact, due to regularity, every row in B contains exactly $(k - 1)/2$ zeroes and M satisfies the hypotheses of Definition 3.3. Thus, every player among the first k ones loses either ℓ or $\ell + 1$ matches. On the other hand, any player in the last m rows, loses at least $(m - 1)/2$ matches, and $m > 3k$ guarantees that $(m - 1)/2 > \ell$. Therefore, if we find a champion of the constructed tournament then we automatically solve the anomalous row problem.

We are now left to prove Lemma 3.4. First we enunciate a game-theoretic lemma by Yao [33] declined within the terms of our problem.

Lemma 3.5 (Yao's Lemma). *Let $\mathcal{A}$ be the family of deterministic algorithms that output a, possibly wrong, solution to the anomalous row problem and probe $o(km)$ cells. Consider $\mathcal{A}$ equipped with a probability distribution. Then consider the function $\mathcal{C}(A, x)$ that returns 1 if the algorithm A is correct on input x and 0 otherwise. Finally, consider a probability distribution $\mathcal{D}$ over $\mathbb{F}_2^{k \times m}$. We have*

---

1. A $(2j + 1)$-vertices tournament is said to be *regular* if every vertex has out-degree j.

$$\min_{x \in \mathcal{D}} \mathbb{E}_{\mathcal{A}} \left[\mathcal{C}(A, x) \right] \leq \mathbb{E}_{\mathcal{A} \otimes \mathcal{D}} \left[\mathcal{C}(A, x) \right] \leq \max_{A \in \mathcal{A}} \mathbb{E}_{\mathcal{D}} \left[\mathcal{C}(A, x) \right].$$

We know that a Monte Carlo algorithm that proves $o(kn)$ cells can be represented as a probability distribution $\mathcal{A}$, in fact it just tosses some coins at run-time and it decides which algorithm to branch into. Therefore $\min_{x \in \mathcal{D}} \mathbb{E}_{\mathcal{A}} \left[\mathcal{C}(A, x) \right]$ is the probability of the Monte Carlo algorithm defined by $\mathcal{A}$ of being right in the worst case, and $\max_{A \in \mathcal{A}} \mathbb{E}_{\mathcal{D}} \left[\mathcal{C}(A, x) \right]$ is the average case of the best deterministic algorithm against a random input with distribution $\mathcal{D}$. Finally, we prove Lemma 3.4.

Proof of Lemma 3.4. It is sufficient to show an input distribution $\mathcal{D}$ such that any deterministic algorithm with running time $o(km)$ succeeds with arbitrarily small probability, for $k, m \to \infty$. We choose the permutation ϕ of $\{1 \dots k\}$ and k permutations $\sigma_1 \dots \sigma_k$ of $\{1 \dots m\}$ uniformly at random. Consider the random matrix $X \in \mathbb{F}_2^{k \times m}$ such that

$$X[i, j] = M[\phi(i), \sigma_i(j)]$$

where M is a deterministic input matrix as in Definition 3.3. Let $\mathcal{D}$ be the distribution of X, and $A \in \mathcal{A}$ be the algorithm such that $\mathbb{E}_{\mathcal{D}} \left[\mathcal{C}(A, x) \right]$ is maximum. It is sufficient to show that $\mathbb{E}_{\mathcal{D}} \left[\mathcal{C}(A, x) \right] \to 0$ to prove that no Monte Carlo algorithm can perform less than $\Omega(km)$ cell probes. Consider the maximum number P of cells probed by A and define

$$\Gamma_{k,m} = \min \left(\sqrt{\frac{km}{P}}, k \right).$$

We now color $\Gamma_{k,m}$ cells in the input matrix. We first color a 1-valued cell in the k-zeroes row, then we choose $\Gamma_{k,m} - 1$ rows containing $k + 1$ zeroes and color a 0-valued cell drawn from each of those. To this end, we assume to perform such coloring before randomizing the input. We want to estimate the probability that the algorithm probes any colorful cell. Define the event E_i "the algorithm picks a colorful cell during the i-th probe". The probability of E_i is $\Gamma_{k,m}/km$ since the chosen cell's row contains a colorful cell with probability $\Gamma_{k,m}/k$ and, given that , the probability of picking the colorful cell is $1/m$.

Therefore,

$$\mathbb{P}_{\mathcal{D}} \left(\bigcup_{i=1}^{P} E_i \right) \leq \sum_{i=0}^{P} \mathbb{P}_{\mathcal{D}} (E_i) \leq \frac{P \Gamma_{k,m}}{km} \leq \sqrt{\frac{P}{km}}.$$

Finally, we notice that, in case none of the colorful cells is probed, the algorithm "sees" a perfectly symmetric distribution over the $\Gamma_{k,n}$ rows containing a colorful cell. Therefore, the best it can do is to produce a random output, which is right with probability $1/\Gamma_{k,n}$, at most. To conclude, consider $E = \bigcup_{i=1}^{P} E_i$, we have

$$\mathbb{E}_{\mathcal{D}} \left[\mathcal{C}(A, x) \right] \leq \mathbb{P}_{\mathcal{D}} (E) + \mathbb{P}_{\mathcal{D}} \left(\mathcal{C}(A, x) = 1 \mid E^c \right)$$
$$\leq \frac{P \Gamma_{k,n}}{kn} + \frac{1}{\Gamma_{k,n}} \to 0$$

where the last limit holds for k and n that goes to infinity simultaneously. □

4 OPTIMAL DETERMINISTIC ALGORITHM

In this section, we present a simple, deterministic, and asymptotically optimal algorithm that finds every champion in $\Theta(\ell n)$ arc lookups and time. We first introduce the algorithm. Then, we prove its correctness and we bound the number of arc it lookups. Finally, we discuss some implementation details to show that the number of operations performed by the algorithm is $\Theta(\ell n)$ and the space required is linear.

4.1 Algorithm Description

We detail our algorithm in Algorithm 1. The number ℓ of matches lost by the champion is unknown to the algorithm. Thus, it performs an exponential search to find the suitable value of α such that $\alpha/2 \leq \ell < \alpha$ (line 2) so to solve the problem by assuming that the champion loses less than α matches.

Algorithm 1

1: **procedure** FINDCHAMPION($T = (V, E)$)
2:     **for** ($\alpha = 1$; true; $\alpha = 2\alpha$) **do**
3:         $A = V$
4:         $S = \{(u, u) \mid u \in V\}$
5:         $\forall u \in V \ \ lost[u] = 0$
6:         **while** $|A| > 2\alpha$ **do**
7:             choose a pair of vertices u, v in $A^2 \setminus S$
8:             $S = S \cup \{(u, v), (v, u)\}$
9:             $loser = $ **if** $(u, v) \in E$ **then** v **else** u
10:            $++lost[loser]$
11:            **if** $lost[loser] \geq \alpha$ **then**
12:               $A = A \setminus \{loser\}$
13:            **end if**
14:         **end while**
15:         $c, lost_c = $ FINDCHAMPIONBRUTEFORCE(A, E)
16:         **if** $lost_c < \alpha$ **then return** c
17:         **end if**
18:     **end for**
19: **end procedure**

At each iteration, the algorithm maintains a set A of "alive" vertices that is initially equal to V. Then, it performs an elimination tournament among the vertices in A by eliminating a player each time it loses α matches (line 12) until only 2α vertices remain alive (line 6). This stop condition guarantees the convergence of the algorithm. The matches are selected arbitrarily to avoid to play the same match multiple times (line 7). When the elimination tournament ends, a candidate champion is found via the FINDCHAMPIONBRUTEFORCE procedure, which exhaustively finds the vertex c of A with the maximum out-degree in T. Whenever the candidate c loses at least α matches (line 16), the value of α is not the correct one and the champion may have been erroneously eliminated before. Thus, c could not be a champion and the algorithm continues with the next value of α (line 2).

In the reminder of this section, we prove the following theorem stating that Algorithm 1 matches the number of arc lookups indicated by the lower bound (Theorem 3.1) and requires linear space.

Theorem 4.1. *Given a tournament graph T with n vertices and with ℓ matches lost by the champion, Algorithm 1 finds every champion with $\Theta(\ell n)$ arc lookups and time. It also requires linear space.*

4.2 Correctness

Let us first assume that the value of α is such that $\alpha/2 \leq \ell < \alpha$. We now prove that, under this assumption, the algorithm correctly identifies a champion. First, we observe that the algorithm cannot eliminate the champions as each of them loses less than α matches. Thus, if we prove that the algorithm terminates, the set A contains all the champions and the FINDCHAMPIONBRUTEFORCE procedure will identify any (potentially all) of them. Note that a champion of T may not be a champion of the sub-tournament restricted to only the vertices in A. This is why FINDCHAMPIONBRUTEFORCE procedure computes the out-degrees of all vertices in A by looking at the edges of the original tournament T. We use the following lemma to prove that, eventually, the condition $|A| = 2\alpha$ is met and the algorithm terminates.

Lemma 4.2. *In any tournament T of n vertices there is at least one vertex having in-degree $(n-1)/2$.*

Proof. The sum of the in-degrees of all vertices of T is exactly $\binom{n}{2} = \frac{n(n-1)}{2}$. Since there are n vertices, there must be at least one vertex with in-degree $\frac{n-1}{2}$. □

Thus, each tournament of $2\alpha + 1$ vertices, or more, has at least one vertex losing at least α matches. This means that the algorithm has always the opportunity to eliminate a vertex from A until there are 2α vertices left. Notice that the above discussion is valid for any value of α smaller than the target one. Thus, any iterations of the exponential search will terminate and it eventually finds a suitable value of α, i.e., $\alpha/2 \leq \ell < \alpha$, where a champion will be identified.

4.3 Complexity

We now present an analysis of the complexity of the algorithm. Let us first consider the cost of an iteration of the exponential search. We observe that each arc lookup increases one entry of $lost$ by one and that none of these entries is ever greater than α. Thus, the elimination tournament takes no more than $n\alpha$ arc lookups. Moreover, the FINDCHAMPIONBRUTEFORCE procedure takes less than $2n\alpha$ arc lookups since it just considers every arc of the remaining 2α alive nodes. Thus, an iteration of the exponential search takes less than $3n\alpha$ arc lookups.

We get the complexity of the overall algorithm by summing up over all the possible values of α, which are all the powers of 2 from 1 up to 2ℓ. Thus, we have at most $3n\sum_{i=0}^{\lceil \log_2(2\ell)\rceil} 2^i = O(\ell n)$ arc lookups.

4.4 Implementation Details

We now prove that Algorithm 1 can be implemented in $\Theta(\ell n)$ time and linear space. We do this by exploiting the fact that Algorithm 1 allows us to choose any arc as soon as its vertices are alive and it has never looked up before. An efficient implementation is achieved by maintaining two arrays of n elements each: an array A storing the alive

vertices and an array $lost$ storing the number of matches lost by each vertex. A counter $numAlive$ stores the number of alive vertices. Our implementation maintains the invariant that the prefix $A[1, numAlive]$ contains only alive vertices. We use two cursors p_1 and p_2 to iterate over the elements in A. At the beginning $p_1 = 1$, $p_2 = 2$ and $numAlive = n$. Our implementation performs a series of matches involving vertex $A[p_1]$ and all other vertices in $A[p_1 + 1, numAlive]$, thus, advancing the cursor p_2. Then, it moves p_1 to the next position. After every match between $A[p_1]$ and $A[p_2]$, we increment $lost$ of the loser, say vertex v. Whenever $lost[v]$ equals α, we eliminate v according to the following two cases, then we decrease $numAlive$ by one. The first case occurs when v is $A[p_1]$. We swap $A[p_1]$ and $A[numAlive]$, we end the current series of matches, and we start a new one. The second case occurs when v is $A[p_2]$. Here, we swap $A[p_2]$ and $A[numAlive]$, and we continue the current series of matches. In both cases, we decrease $numAlive$ by 1 so that we preserve the invariant.

A similar, slightly less efficient, implementation employs a linked list A to store the alive vertices. In this implementation, the removal of an element from the list is trivial, and p_1 and p_2 are pointers that always advance in the list. When p_2 reaches the end of the list, we advance p_1 by one position in the list and we set p_2 to point to the element just after p_1. This implementation allows us to process the vertices according to the input order (as we never swap elements), which may be desirable in practice if we can somehow predict the strongest of the vertices and sort them according.

As each step of the exponential search ignores the arc lookups of the previous steps, i.e., certain arcs may be considered more than once. Therefore, to reduce the number of arc lookups preserving the time complexity at the cost of using $\Theta(\ell n)$ space instead of $O(n)$, an hash table can be employed to store the results of all arc lookups across the exponential search steps so to avoid unnecessary repeated computations. In detail, each time Algorithm 1 wants the result of a match, it checks the hash table first and, only if this is a new arc lookup, the algorithm compute the result of the match and stores the result in the hash table for the next exponential search steps.

5 GENERALIZATIONS OF THE PROBLEM

We now discuss some generalizations of the Copeland winner problem and we modify Algorithm 1 to solve these problems efficiently. First, we show how to retrieve the top k items, i.e., not only the top-1, by maintaining the complexity proportional to the number of matches lost by the k-th player. Then, we consider the case of a binary machine learned classifier returning a pair of probabilities instead of a binary outcome and redefine the problem in a probabilistic fashion. Finally, we consider the case in which we are able to process batches of arc lookups in parallel, so to exploit parallel processing units, e.g., GPUs.

5.1 Top-k retrieval Version

A simple and useful generalization of the Copeland winner problem is to find the top-k results, i.e., the k vertices with the highest out-degrees. In this setting, the exponential

search of Algorithm 1 can be modified to find the minimum value of α such that the number ℓ_k of matches lost by the k-th result is between $\alpha/2$ and α. To this end, the exponential search must end whenever it finds k vertices with less than α comparisons lost. To accomplish this task, the FIND-CHAMPIONBRUTEFORCE(A, E) procedure must be modified to return the indices of the top-k results of A along with number of matches lost by them. Since $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_n$, the higher the value of k, the higher the time complexity $O(n\ell_k)$ of the algorithm.

5.2 Probabilistic Version

Typically, the outcome of a pairwise classifiers is not a binary response, instead it is a pair of complementary probabilities that can be interpreted as the algorithm's confidence about the comparison's outcome. Thus, a natural generalization of the Copeland winner problem emerges if we associate to each arc (u, v) the probability $p_{u,v}$ of u beating v. Since the probabilities are complementary, we also know that $p_{v,u} = 1 - p_{u,v}$. We refer to this graph as *probabilistic tournament graph*. In this setting, the arcs are Bernoulli random variables, and we define the champion as the player u minimizing the expected number of matches lost, i.e., $\sum_{v \in V} p_{v,u}$ by linearity of the expectation. Since we want our complexity to be parameterized with the expected number of matches lost by the champion, we coherently call this quantity ℓ. In this section, we show that Algorithm 1 needs only little adaptation to work in this setting.

Consider the pseudocode of Algorithm 1, we treat *lost* counters as real-valued and substitute line 10 with two commands incrementing $lost[u]$ by $p_{v,u}$ and $lost[v]$ by $p_{u,v}$. Once operated these slight modifications we are ready to prove the following theorem (analogous of Theorem 4.1).

Theorem 5.1. *Let T be a probabilistic tournament graph with n vertices and with ℓ the expected number of matches lost by the champion. The modified version of Algorithm 1 described above finds every champion by requiring $\Theta(\ell n)$ arc lookups and time. The algorithm requires linear space.*

Correctness

The correctness proof is almost identical to the one we have detailed in Section 4. We are not repeating the whole proof, in fact it is sufficient to substitute occurrences of "losses" with "expected losses" and reformulate the Lemma 4.2 as follows to obtain the desired proof.

Lemma 5.2. *In any probabilistic tournament T of n vertices there is at least one vertex u such that $\sum_{v \in V} p_{v,u} \geq (n-1)/2$. In other words, there exists a player whose expected number of matches lost is at least $(n-1)/2$.*

Proof. The sum of the "expected losses" of all vertices of T is exactly $\binom{n}{2} = \frac{n(n-1)}{2}$. Since there are n vertices, there must be at least one vertex losing $\frac{n-1}{2}$ matches, on average. $\square$

Complexity

The complexity analysis is again akin to the one of Section 4, but we dig in a deeper details here. Each unfolded arc increases $\sum_{u \in V} lost[u]$ by one; since $lost[u]$ of any $u \in V$ is incremented until it surpasses α of at most one unit

at a time, then $lost[u]$ cannot be greater than $\alpha + 1$ and $\sum_{u \in V} lost[u] < (\alpha + 1)n$. Therefore no more than $(\alpha + 1)n$ arcs are unfolded during the elimination step of a single iteration of the exponential search. Moreover, as in Section 4, FINDCHAMPIONBRUTEFORCE procedure takes less than $2n\alpha$ arc lookups. Thus an iteration of the exponential search takes less than $3n(\alpha + 1)$ arc lookups, and summing up all these arc lookups we get the desired $O(\ell n)$ complexity.

5.3 Parallel (Batched) Version

In modern architectures, e.g., GPUs, it is possible to perform multiple arc lookup operations in parallel. A natural question is whether we are able to take full advantage of this parallelism to cut down the complexity of Algorithm 1. In this subsection, we propose Algorithm 2 under the assumption to be able to unfold a batch of B arcs in parallel.

In particular, Algorithm 2 processes $O\left(\frac{\ell n}{B} + \ell \log B\right)$ batches, so the overhead is asymptotically negligible if $B = O\left(n/\log n\right)$, which is a condition that often holds in practice.

Algorithm 2 is a slight modification of Algorithm 1. As the previous algorithm, it performs an exponential search of ℓ repeatedly doubling the parameter α. For each α it assumes that the champion belongs to the set of *alive vertices* A and performs an elimination tournament among the vertices of A eliminating any player that loses α matches. The elimination step is now performed in batches (line 12) and terminates when the alive players are few enough (line 7). The method FINDCHAMPIONBRUTEFORCE$_{PAR}$ (line 18) can be parallelized with no efforts by unfolding all $O\left(6\alpha n\right)$ arcs in batches of B arcs at a time, hence we focus on the elimination step. The main difference with respect to Algorithm 1 resides in the procedure BUILDBATCH, which decides what are the B arcs to lookup in parallel. It creates local copies A_{loc} and $lost_{loc}$ of the set A and the vector $lost$, then the procedure selects matches in $A_{loc} \times A_{loc}$ that have not been played yet and, for each of them, assigns a loss to both opponents. Now suppose that the batched games were played sequentially (namely, played at line 31) and $lost$ and A were updated accordingly: we would have that $lost_{loc}$ provides an upper estimate of $lost$ and $A_{loc} \subseteq A$. Therefore, it is guaranteed that every insertion in a batch will produce a match loss for a player that would be still alive in case we unfolded the batch sequentially. This is a point worth stressing since it guarantees that $lost[u] \leq \alpha$ for each $u \in V^2$. Finally, even though it is not guaranteed that BUILDBATCH produces a B-sized batch, for that to happen it is sufficient that A has at least $2B + 2\alpha$ elements. This can be enforced halving the batch size every time this condition does not hold (line 8) and we will see that this will not spoil the complexity of Algorithm 2. Intuitively, the elimination step consists of two different epochs: the first one unfolds arcs in B-sized batches (where B is the original batch size) until $|A| \geq 2B + 2\alpha$; the second one processes smaller and smaller batches until $|A|$ is small enough (line 7).

---

2. We will employ this property in the complexity analysis.

Algorithm 2

```
 1: procedure FINDCHAMPION_PARALLEL(T = (V, E), B)
 2:     for (α = 1; true; α = 2α) do
 3:         A = V
 4:         S = {(u, u) | u ∈ V}
 5:         ∀u ∈ V  lost[u] = 0
 6:         B' = B
 7:         while |A| > 6α do
 8:             while |A| < 2B' + 2α do
 9:                 B' = B'/2
10:             end while
11:             batch = BUILDBATCH(A, S, B', lost, α)
12:             UNFOLDINPARALLEL(batch)
13:             for (u, v) in batch do
14:                 loser = if (u, v) ∈ E then v else u
15:                 INCREASELOSS(A, lost, α, loser)
16:             end for
17:         end while
18:         c, lost_c = FINDCHAMPIONBRUTEFORCE_PAR(A, E, B)
19:         if lost_c < α then return c
20:         end if
21:     end for
22: end procedure
23:
24: procedure BUILDBATCH(A, S, B', lost, α)
25:     batch = ∅
26:     A_loc = A
27:     lost_loc = lost
28:     while |batch| < B' do
29:         Choose (u, v) ∈ A²_loc \ S
30:         S = S ∪ {(u, v), (v, u)}
31:         batch = batch ∪ {(u, v)}
32:         INCREASELOSS(A_loc, lost_loc, α, u)
33:         INCREASELOSS(A_loc, lost_loc, α, v)
34:     end while
35:     return batch
36: end procedure
37:
38: procedure INCREASELOSS(A, lost, α, v)
39:     ++lost[v]
40:     if lost[v] ≥ α then
41:         A = A \ {v}
42:     end if
43: end procedure
```

Correctness

The correctness can be proven exactly in the same way as the sequential case, the only detail to take care about is that the function BUILDBATCH terminates by producing a B-sized batch. It is sufficient to notice that as long as $|A_{loc}| > 2\alpha$ there is an arc to unfold in $A_{loc}^2 \setminus S$ (using the same argument of the sequential case), and that since we call INCREASELOSS at most $2B'$ times at each iteration, then $|A| \geq 2B' + 2\alpha$ (line 8) is sufficient to ensure the termination.

Complexity

Theorem 5.3. *Given a tournament graph T with n vertices and with ℓ matches lost by the champion, Algorithm 2 finds every*

champion *by requiring $O\left(\frac{\ell n}{B} + \ell \log B\right)$ calls to UNFOLDIN-PARALLEL and $O(\ell n)$ time and space.*

Proof. Consider the i-th iteration of the cycle at line 7 and denote with A_i the number of alive vertices $|A|$ and with B_i the value of B', evaluated immediately before calling the BUILDBATCH function at line 11. In particular we have $A_1 = |V|$ and $B_1 = B$. First, we prove the following lemmas.

Lemma 5.4. *For each $i \geq 0$, $A_i - B_i \leq A_{i+1} \leq A_i$ holds.*

Proof. The first inequality holds since no more than B_i games are played during the i-th iteration and thus no more than B_i players are removed from the alive set. The second inequality holds since the set A of alive vertices decreases over time. □

Lemma 5.5. *Let j be the first iteration in which the conditional statement at line 8 is true, that is $j = \min\{i \mid A_i < 2B + 2\alpha\}$. For each $i \geq j$, $2B_i + 2\alpha \leq A_i \leq 4B_i + 2\alpha$ holds.*

Proof. We prove it by induction.
  Base Case, $i = 1$: it is sufficient to notice that $B_{j-1} = 2B_j$ and $A_j < 2B_{j-1} + 2\alpha \leq A_{j-1}$ hold thanks to the definition of j, and combine those equations with Lemma 5.5.
  Inductive Case, $i > 1$: during the i-th iteration we have two cases depending on whether we update the value of B' or not. If we do not update B', that is $A_i \geq 2B_{i-1} + 2\alpha$, then we have $B_i = B_{i-1}$ and $A_i \leq A_{i-1} \leq 4B_{i-1} + 2\alpha$ by inductive hypotheses. Otherwise, if we update B', we have $A_i < 2B_{i-1} + 2\alpha$ and $B_{i-1} = 2B_i$, then $A_i < 4B_i + 2\alpha$ and $A_i \geq A_{i-1} - B_{i-1} \geq B_{i-1} + 2\alpha = 2B_i + 2\alpha$. □

We now fix α and upper-bound the number of arcs unfolded for each batch size B_i. First we deal with the case $B_i = B$ in which we employ the original batch size; in that case, we can safely upper-bound the number of arcs with αn since every *lost* counter is never greater than α and every arc unfolded increases a *lost* counter by one. Then, consider the case $B_i = B/2^k$, for a specific value of k; we have that $|A_i| \leq 4B/2^k + 2\alpha$ and, by applying the same argument as above on the elements of A_i, that at most $\alpha\left(4B/2^k + 2\alpha\right)$ arcs are unfolded using a batch of size $B/2^k$. Thanks to the clauses at lines 7 and 8, we have $6\alpha \leq A_i \leq 4B_i + 2\alpha$, which implies $B_i \geq \alpha$. To compute the total number of calls to UNFOLDINPARALLEL, it is sufficient to divide the maximum number of arc lookups ($\alpha|A_i|$) by the appropriate batch size (B_i) and sum them up

$$
\begin{aligned}
\sum_i \frac{\alpha|A_i|}{B_i} &\leq \frac{\alpha n}{B} + \sum_{i=1}^{\lceil \log_2(B/\alpha) \rceil} \frac{\alpha|A_i|}{B_i} \\
&\leq \frac{\alpha n}{B} + \sum_{i=1}^{\lceil \log_2(B/\alpha) \rceil} \frac{\alpha\left(4(B/2^i) + 2\alpha\right)}{B/2^i} \\
&< \frac{\alpha n}{B} + 4\alpha \log_2 B + \frac{2\alpha^2}{B} \sum_{i=1}^{\lceil \log_2(B/\alpha) \rceil} 2^i \\
&< \frac{\alpha n}{B} + 4\alpha \log_2 B + 2\alpha
\end{aligned}
$$

where the first addendum refers to the batches processed unfolding B arcs at a time, while the other addenda refers to the case of smaller batch sizes. Finally, to get the number

of parallel unfoldings during the entire execution, it suffices to sum the quantity above for $\alpha = 1, 2, \ldots, 2^{\lceil \log \ell \rceil}$ and we get the desired $O\left(\frac{\ell n}{B} + \ell \log B\right)$.

Now it remains to prove that Algorithm 2 uses $O(\ell n)$ operations and space. The proof is the same as for Algorithm 1, we mainly need to pay attention to lines 26 and 27 since creating local copies would increase the complexity. Fortunately, it is sufficient to use the global versions of A and $lost$, store in a list the changes performed on them, then restore their state before terminating BUILDBATCH. In fact, we adopted local copies only to make the pseudocode clearer. Moreover, since the BUILDBATCH can temporarily skip some vertices (according to the local copy of $lost$) that may be re-included later after the parallel unfold, we cannot employ the linear-space selection described in Section 4.4. In this case, we further need to associate to each node u the set (hash table) of all arcs $(u, \cdot) \in E$ unfolded by the algorithm, so to skip the ones already unfolded, in constant time. The solution proposed in Section 4.4, which employs the cursors p_1 and p_2 to decide the arcs to unfold, properly extended with this check, guarantees $O(\ell n)$ time and space. □

Implementation Details

Algorithm 2 could not use all the comparisons that are available in a single batch, because of the batch size halving (row 8) or because the brute force call (row 18) involves a number of arcs that is not divisible for the batch size. For this reason, we employ a simple heuristic to exploit each batch the most, which applies when employing the hash table to store the results of the arc lookups (Section 4.4). In detail, we add new arcs to the batch, deterministically, each time Algorithm 2 asks to unfold a partially filled batch. We use an heap data structure to get the node with the smallest number of comparisons lost that still has unfolded arcs, then we add to the batch the remaining unfolded arcs, in the order they appear, until the batch becomes full. If all node's arcs are added and the batch is still non-full, then the previous operation is repeated until either the batch becomes full or all arcs have been unfolded.

6 EXPERIMENTS

In this section, we present a comprehensive experimental assessment of the proposed algorithms on a Question Answering task. In detail, we focus on passage ranking that aims at selecting, given a question, the most relevant among a set of textual passages answering the question. To this end, we employ an existing state-of-the-art pairwise model that works by comparing two results at a time and by selecting the winners of the induced round-robin tournament. In this scenario, the proposed algorithms aim to find the tournament champions by reducing the number of pairwise comparisons, i.e., arc lookup, performed using the ML model. In the following, we first describe the experimental setting, then we evaluate the proposed algorithms in terms of number of comparisons and speedup of the ranking process.

Dataset

For the the assessment we employ the Microsoft MAchine Reading COmprehension dataset (MS MARCO) [26]. It is a large scale dataset for Question Answering and consists of approximately 1 million anonymized questions sampled from the Bing search query logs and about 9 million passages extracted from web pages. The development set used for the assessment contains $6,980$ queries having one relevant passage each, on average.

Pairwise Model

Nogueira *et al.* recently tackled the task of ranking passages by using a three-stage ranking architecture [28]. The duoBERT models recently scored among the top-10 solutions of the MS MARCO Passage Ranking Leaderboard[3] and as the first solution whose public code is publicly available[4]. The first stage selects the top-1000 results using the fast BM25 algorithm. The second stage re-ranks these results using a monoBERT neural model [27], which ingests the text of a document at a time to classify it as relevant or not. Lastly, the third stage re-ranks the top-30 results of the previous stage by using a duoBERT pairwise model [28] that classifies all pairs of document's texts to induce a round-robin tournament among the results. In particular, the two most promising configurations presented in Nogueira *et al.* [27] have been tested: duoBERT_{PROBABILISTIC} and duoBERT_{BINARY}. The former works by assigning to each document the sum of the probabilities of all comparisons, while the latter rounds these probabilities in $\{0, 1\}$ before summing them.

Experimental Methodology

In our experiments, we replicate the full multi-stage pipeline proposed by Nogueira *et al.* and we apply the proposed algorithms in the last stage of the pipeline, i.e., top-30 re-ranking. In particular, given a query and the set of its top-30 passages, each algorithm drives the identification of the champions deciding which pairs of passages to compare using the ML model. The objective is to retrieve the top passages by reducing the number of pairwise inferences, i.e., arc lookups, performed by the duoBERT models.

We assess the proposed algorithms by measuring the number of comparisons and the time spent by the ML model to perform all inferences. For fairness, even if our contribution does not regard the effectiveness of the model, we also report the Recall@k metric assessing the fraction of relevant documents captured within the top-k results.

Testing Details

The tests were performed on a machine with sixteen Intel Xeon E5-2630 cores clocked at 2.40GHz, 192GiB RAM, equipped with a NVIDIA TITAN Xp GPU. The GPU has been used to run the monoBERT and duoBERT models.

6.1 Experimental Results

We now present the results of our experimental evaluation. To ease the discussion, we start by discussing the evaluation in the binary setting for the retrieval of the top-1 result (Algorithm 1 and its possible implementations). We then present the results of the proposed algorithms on the problem generalizations, i.e., top-k retrieval, probabilistic setting, and parallel setting (Algorithm 2).

---

3. https://microsoft.github.io/msmarco/
4. https://github.com/castorini/duobert

TABLE 1
Average number of inferences of different implementations of
Algorithm 1 when applied to duoBERT to retrieve the top-1 result on the
MS MARCO dataset. Columns identify whether the implementation
exploits the input order, while rows identify whether it exploits the past
lookups to avoid multiple unfolds of a same arc.

	Ignore input order	Exploit input order
Ignore past lookups	126.09	125.81
Exploit past lookups	76.58	64.62

TABLE 2
Efficiency-Effectiveness performance achieved by monoBERT,
duoBERT, and duoBERT & Alg. 1 when retrieving the top-1 result on
the MS MARCO dataset.

Method	Recall@1	Inferences	Time (s)
BM25 + monoBERT	0.251	1000	65.91
+ duoBERT$_{\text{BINARY}}$	0.269	870	57.34
+ duoBERT$_{\text{BINARY}}$ & Alg. 1	0.269	65	4.26

6.1.1 Asymptotically-optimal Deterministic Algorithm

Section 4.4 discusses some implementation details to take into account when implementing Algorithm 1. In particular, there are two orthogonal aspects to consider in the implementation that we want to assess: exploitation of the input order and exploitation of the past arc lookups. The first aspect exploits the order of the input list when deciding the order of the arc lookups. Since our inputs consists of 30 passages that have already been sorted by the second ranking stage, we expect to have more relevant passages in the first positions of the input. Therefore, it could be desirable to start by performing the comparisons among the more relevant passages coming from the second stage. The second aspect avoids multiple unfolds of a same arc by storing the arc lookups performed during the tournament. Therefore, we can easily save time using a little extra space.

We now assess the impact of the two orthogonal implementation aspects described above, which lead to four implementations. Table 1 reports the average number of inferences of the different implementations of Algorithm 1 when applied to duoBERT to retrieve the top-1 result on the MS MARCO dataset. As expected, the two aspects contribute to reduce the average number of inferences. In particular, we notice that the implementation exploiting the input order is more efficient when used together with the hash table, and that their combination nearly halves the number of inferences of the implementation ignoring both aspects.

Table 2 reports the performance of the best implementation above, i.e., the one exploiting the input order and the past lookups, within the ranking pipeline proposed by Nogueira et al. We report Recall@1, number of inferences and inference time of all ranking stages. The first row shows the performance of the first two stages of the ranking pipeline, i.e., BM25 + monoBERT, used here to retrieve the top-30 results to re-rank. It retrieves the correct answer

for about 25% of the queries but it requires, on average, about 66 seconds when applied to the top-1,000 results returned by BM25. The second row shows the performance of duoBERT$_{\text{BINARY}}$ when employed as third stage of the ranking pipeline. As this model does not guarantee symmetric predictions, each comparison needs two inferences, i.e., u versus v and v versus u; it thus requires $30 \times 29 = 870$ inferences. duoBERT$_{\text{BINARY}}$ improves the quality of the returned list with respect to the previous stage as it retrieves the correct answer for about 27% of the queries. However, we want to highlight that this third stage almost doubles the running time as it require about 57 seconds that must be added to the 66 seconds required by the first two stages, i.e., BM25 + monoBERT. The third row of Table 2 shows the performance of the third stage when employing Algorithm 1 to decide which pairs of passages to compare using the duoBERT$_{\text{BINARY}}$ model. The recall metric is the same as duoBERT$_{\text{BINARY}}$. This result is expected as we proved the algorithm correctness. On average, this configuration requires about 4 seconds per query and it speeds up the ranking process of the third stage of about $13\times$ with respect to the previous configuration. Moreover, the time cost of the third stage is now negligible with respect to the one of the first two stages.

The average number of inferences required by our approach is about 65, which is very close to the minimum number of inferences required to solve this problem when the Champion wins all comparisons, i.e., $29 \times 2 = 58$ inferences. In particular, 95% of the queries are solved with only 50 comparisons or less, i.e., solved with less than 100 model inferences. In addition, we want to highlight that if we apply the algorithm to a symmetric model, we would not need to perform two inferences per comparison, and the algorithm would perform just a few inferences per item.

6.1.2 Top-k Retrieval and Probabilistic Version

Table 3 reports the performance of Algorithm 1 in the top-k retrieval task, both in the binary and the probabilistic settings. As before, we report Recall@k, for k in $\{1, 2, 3, 4, 5\}$, number of inferences, and inference time of all ranking stages. The first row shows the performance of the first two stages of the ranking pipeline introduced by Nogueira et al., i.e., BM25 + monoBERT. The second and fourth rows show the performance of duoBERT$_{\text{BINARY}}$ and duoBERT$_{\text{PROBABILISTIC}}$ when employed as third stage of the ranking pipeline. The two configurations require the same number of inferences, i.e., $30 \times 29 = 870$, and the same inference time, as the underlying model is the same. The binary configuration shows a slightly higher recall than the probabilistic one. Both the versions improve the recall of the previous ranking stage, thus confirming that tournaments are a good modeling of this problem. The third and fifth rows show the performance of these models when employing Algorithm 1 to perform the tournament among the top-30 results of each query. In both cases the recall is preserved, as the algorithm is correct. The proposed algorithm speeds up the ranking process from $13\times$ to $2\times$ in the binary setting and from $6\times$ to $2\times$ in the probabilistic setting, for k ranging from 1 to 5. Remark that Algorithm 1 obtains excellent results in the top-1 retrieval task of both settings.

TABLE 3
Efficiency-Effectiveness performance achieved by monoBERT, duoBERT, and duoBERT & Alg 1 when retrieving the top-k results on the MS MARCO dataset. The number of inferences of monoBERT and duoBERT is independent of the value of k.

Method	Metric	k					
		1	2	3	4	5	10
BM25 + monoBERT	Recall	0.251	0.361	0.436	0.492	0.531	0.638
	# Inference			1,000			
	Time (sec.)			65.91			
+ duoBERT$_{\text{BINARY}}$	Recall	0.269	0.385	0.459	0.516	0.552	0.654
	# Inference			870			
	Time (sec.)			57.34			
+ duoBERT$_{\text{BINARY}}$ & Alg. 1	Recall	0.269	0.385	0.459	0.516	0.552	0.654
	# Inference	65	130	234	266	427	711
	Time (sec.)	4.26	8.58	15.42	17.53	28.14	46.83
	Speedup	(13.5×)	(6.7×)	(3.7×)	(3.2×)	(2.0×)	(1.2×)
+ duoBERT$_{\text{PROBABILISTIC}}$	Recall	0.266	0.385	0.460	0.514	0.550	0.653
	# Inference			870			
	Time (sec.)			57.34			
+ duoBERT$_{\text{PROBABILISTIC}}$ & Alg. 1	Recall	0.266	0.385	0.460	0.514	0.550	0.653
	# Inference	134	209	291	355	445	732
	Time (sec.)	8.86	13.759	19.201	23.397	29.307	48.267
	Speedup	(6.5×)	(4.2×)	(3.0×)	(2.5×)	(2.0×)	(1.2×)

TABLE 4
Average values of ℓ_k when varying k and the tournament type.

Tournament Type	k					
	1	2	3	4	5	10
Binary	0.05	1.09	2.13	3.15	4.18	9.19
Probabilistic	0.78	1.77	2.78	3.78	4.78	9.58

Taking into account that ℓ_k, i.e., the number of matches lost by the k-th result, drives the time complexity of our algorithm, we report in Table 4 the different values of ℓ_k when varying k and the tournament type, i.e., binary or probabilistic. The table shows that, on this dataset, ℓ_k rapidly increases as k grows and that ℓ_k is always higher in the probabilistic setting than in the binary setting. Indeed, in practice, the number of inferences performed by our algorithm rapidly increases as k grows and that the speedups achieved in the probabilistic setting are always smaller than the ones achieved in the binary setting (Table 3).

6.1.3 Parallel (Batched) Version

Table 5 reports the performance of Algorithm 2 in the parallel setting where the algorithm can unfold a batch of multiple arcs in parallel. The table reports the number of inferences and the inference time of all ranking stages, for values of batch size between 2 and 256 when retrieving the top-1 result on the MS MARCO dataset. The Recall@1 metric is not reported as the correctness of the algorithm

guarantees that the effectiveness does not change with the batch size. Indeed, Recall@1 is always close to 27% as in the non-parallel setting. The first row shows the performance of the first two stages of the ranking pipeline, i.e., BM25 + monoBERT, while the second row shows the performance of the third stage, i.e., duoBERT$_{\text{BINARY}}$. The number of batch inferences linearly decreases when increasing the batch size for both configurations, as we can unfold more arcs in parallel per batch. For instance, with a batch size of 64, we can perform 64 inferences at a time and the full round-robin tournament requires only $\lceil 870/64 \rceil = 14$ rounds to perform all inferences. The third row shows the performance of duoBERT$_{\text{BINARY}}$ used as third stage when employing Algorithm 2 to perform the (batched) tournament among the top-30 results of each query. Our algorithm speeds up the ranking from $13\times$ to $3\times$ for batch size ranging from 2 to 64. As expected, the speedup decreases when increasing the batch size as the number of results involved in the tournament is very limited. Indeed, the algorithm can accurately unfold only one arc for each alive vertex (Algorithm 2, set A); it then fills the batch with a simple heuristic that explores all arcs of just a few promising vertices (as described in the "Implementation Details" subsection of Section 5.3). Therefore, as the batch size becomes bigger than the number of results, i.e., 30 in our setting, the choices of the algorithm become less oriented. Nevertheless, Algorithm 2 speeds up the ranking of duoBERT$_{\text{BINARY}}$ for all the values of batch size tested.

TABLE 5
Efficiency of parallel (batched) implementations of monoBERT, duoBERT, and duoBERT & Alg 2 when retrieving the top-1 result on the MS MARCO dataset.

Method	Metric	Batch Size							
		2	4	8	16	32	64	128	256
BM25 + monoBERT	Inferences	500	250	125	63	32	16	8	4
	Time (s)	32.95	16.48	8.24	4.15	2.11	1.05	0.53	0.26
+ duoBERT$_{\text{BINARY}}$	Inferences	435	218	109	55	28	14	7	4
	Time (s)	28.67	14.37	7.18	3.62	1.85	0.92	0.46	0.26
+ duoBERT$_{\text{BINARY}}$ & Alg. 2	Inferences	33	23	14	8	5	4	4	4
	Time (s)	2.14	1.54	0.93	0.55	0.31	0.28	0.26	0.25
	Speedup	(13.4×)	(9.3×)	(7.7×)	(6.6×)	(5.9×)	(3.3×)	(1.7×)	(1.0×)

7 CONCLUSION

We addressed the problem of how to efficiently solve the retrieval of the top-1 result when employing pairwise machine learning classifiers. We mapped it to the problem of finding champions in tournament graphs by minimizing the number of arc lookups, i.e., the number of comparison done through the classifier. We showed that, given the number ℓ of matches lost by the champion, $\Omega(\ell n)$ arc lookups are required to find a champion, and generalized this statement for randomized algorithms that are only correct with some constant probability. Then, we presented an asymptotically optimal deterministic algorithm that solves the problem and matches the lower bound without knowing ℓ. We also turned our attention to three natural variants of the original problem, and showed algorithms that solve them. First, we solved the problem of finding all the top-k players simultaneously. Second, we considered a probabilistic tournament in which any cell of the adjacency matrix contains a probability, and achieved the same performance in that more general case. Third, we supposed we were able to probe B adjacency matrix cells in parallel and achieved a linear (and thus asymptotically optimal) speedup. Finally, we experimentally evaluated the proposed algorithms to speed-up a state-of-the-art solution for ranking on public data. Results show that we are able to speed up the retrieval of the top-1 result of up to $13\times$ in the classic binary setting. We also evaluated the three variants of the original problem and we showed that our proposals speeds-up the retrieval from $13\times$ to $2\times$ for k ranging from 1 to 5 in the binary setting (first variant) and from $6\times$ to $2\times$ for the same range of k in the probabilistic setting (second variant). In the parallel setting (third variant), our proposal consistently speeds up the retrieval of the top-1 result for all the values of batch size tested.

As future work, we intend to investigate three main research directions. On the theoretical side, it would be interesting to characterize the leading constant in the complexity of finding the Copeland winner to better compare the lower bounds and the proposed algorithms. On a more applied side, it is worth investigating heuristics to increase the speed up of our algorithms while retaining their theoretical performance. Lastly, it would be also interesting to investigate the dependency between the number of arc lookups performed by our algorithms and the probability distribution of the graph arcs, so to link the complexity to the data at hand.

REFERENCES

[1] Qingyao Ai, Xuanhui Wang, Nadav Golbandi, Mike Bendersky, and Marc Najork. Learning groupwise scoring functions using deep neural networks. 2019.

[2] Nir Ailon and Mehryar Mohri. An efficient reduction of ranking to classification. In *Proc. COLT*, pages 87–98. Omnipress, 2008.

[3] Nir Ailon and Mehryar Mohri. Preference-based learning to rank. *Machine Learning*, 80(2-3):189–211, 2010.

[4] Miklós Ajtai, Vitaly Feldman, Avinatan Hassidim, and Jelani Nelson. Sorting and selection with imprecise comparisons. *ACM Trans. Algorithms*, 12(2):19:1–19:19, 2016.

[5] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[6] Jeffrey S Banks. Sophisticated voting outcomes and agenda control. *Social Choice and Welfare*, 1(4):295–306, 1985.

[7] Lorenzo Beretta, Franco Maria Nardini, Roberto Trani, and Rossano Venturini. An optimal algorithm to find champions of tournament graphs. In *Proc. SPIRE*, volume 11811 of *Lecture Notes in Computer Science*, pages 267–273. Springer, 2019.

[8] Arindam Biswas, Varunkumar Jayapaul, Venkatesh Raman, and Srinivasa Rao Satti. Finding kings in tournaments. *Discrete Applied Mathematics*, 322:240–252, 2022.

[9] Felix Brandt, Markus Brill, and Paul Harrenstein. Tournament solutions. In *Handbook of Computational Social Choice*, pages 57–84. Cambridge University Press, 2016.

[10] Mark Braverman and Elchanan Mossel. Noisy sorting without resampling. In Shang-Hua Teng, editor, *Proc. SODA*, pages 268–276. SIAM, 2008.

[11] Pierre Charbit, Stéphan Thomassé, and Anders Yeo. The minimum feedback arc set problem is NP-hard for tournaments. *Combinatorics, Probability & Computing*, 16(1):1–4, 2007.

[12] Arthur H Copeland. A reasonable social welfare function. Technical report, mimeo, 1951. University of Michigan, 1951.

[13] Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Sorting with recurrent comparison errors. In *Proc. ISAAC*, volume 92 of *LIPIcs*, pages 38:1–38:12, 2017.

[14] Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Optimal sorting with persistent comparison errors. In *Proc. ESA*, volume 144 of *LIPIcs*, pages 49:1–49:14, 2019.

[15] Barbara Geissmann, Matús Mihalák, and Peter Widmayer. Recurring comparison faults: Sorting and finding the minimum. In *Proc. FCT*, volume 9210 of *Lecture Notes in Computer Science*, pages 227–239. Springer, 2015.

[16] Gregory Z. Gutin, George B. Mertzios, and Felix Reidl. Searching for maximum out-degree vertices in tournaments. *CoRR*, abs/1801.04702, 2018.

[17] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22(1):5–53, 2004.

[18] Shengyu Huang, Chih-Hung Liu, and Daniel Rutschman. Approximate selection with unreliable comparisons in optimal expected time, 2022.

[19] Olivier Hudry. A survey on the complexity of tournament solutions. *Mathematical Social Sciences*, 57(3):292–303, 2009.

[20] Rolf Klein, Rainer Penninger, Christian Sohler, and David P. Woodruff. Tolerant algorithms. In *Algorithms - ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 736–747. Springer, 2011.

[21] Jean-Francois Laslier. *Tournament solutions and majority voting*. Number 7. Springer Verlag, 1997.

[22] Stefano Leucci and Chih-Hung Liu. Approximate minimum selection with unreliable comparisons. *Algorithmica*, 84, 01 2022.

[23] Tie-Yan Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[24] Yoelle Maarek. Alexa and her shopping journey. In *Proc. CIKM, 2018*. ACM, 2018.

[25] John W. Moon. *Topics on tournaments*. Holt, Rinehart, and Winston, 1968.

[26] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. MS MARCO: A human generated machine reading comprehension dataset. In *Proc. NIPS 2016*, volume 1773 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[27] Rodrigo Nogueira and Kyunghyun Cho. Passage re-ranking with BERT. *CoRR*, abs/1901.04085, 2019.

[28] Rodrigo Nogueira, Wei Yang, Kyunghyun Cho, and Jimmy Lin. Multi-stage document ranking with BERT. *CoRR*, abs/1910.14424, 2019.

[29] Kenneth B. Reid. Tournaments. In *Handbook of Graph Theory, 2nd edition*. Chapman and Hall/CRC, 2013.

[30] Jian Shen, Li Sheng, and Jie Wu. Searching for sorted sequences of kings in tournaments. *SIAM J. Comput.*, 32(5):1201–1209, 2003.

[31] Patrick Slater. Inconsistencies in a schedule of paired comparisons. *Biometrika*, 48(3/4):303–312, 1961.

[32] Qiang Wu, Christopher J. C. Burges, Krysta Marie Svore, and Jianfeng Gao. Adapting boosting for information retrieval measures. *Inf. Retr.*, 13(3):254–270, 2010.

[33] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *Proc. FOCS*, pages 222–227. IEEE Computer Society, 1977.

Roberto Trani received the PhD degree in computer science from the University of Pisa, in 2020. He is a postdoctoral research fellow of the National Research Council of Italy, within the High Performance Computing Laboratory, and his research interests are machine learning, algorithms, information retrieval, and high performance computing. For more information: http://hpc.isti.cnr.it/roberto-trani/.

Rossano Venturini received the PhD degree from the University of Pisa in 2010. He is an associated professor at the Computer Science Department of the University of Pisa. His research interests are mainly focused on the design and the analysis of algorithms and data structures for indexing and searching large collections. He received two Best Paper Awards at ACM SIGIR in 2014 and 2015. For more information: http://pages.di.unipi.it/rossano.

Lorenzo Beretta received his master degree from University of Pisa and Scuola Normale Superiore in 2020. He is a PhD student at the University of Copenhagen, within the BARC research centre. His research interest is in discrete algorithms: in particular, he is interested in geometrical optimization problems, sublinear algorithms and hashing.

Franco Maria Nardini received the PhD degree from the University of Pisa in 2011. He is a senior researcher with the National Research Council of Italy. His research interests focus on web information retrieval and machine learning. He authored more than 70 papers in peer-reviewed international journal and conferences. He received the ACM SIGIR 2015 Best Paper Award, ECIR 2014 Best Demo Paper Award, and the ECIR 2022 Industry Impact Award. For more information: http://hpc.isti.cnr.it/ nardini.