Maintaining a Kingdom in a Tournament

Oren Weimann and Raphael Yuster

University of Haifa, Israel
oren@cs.haifa.ac.il, raphy@math.haifa.ac.il

Abstract. A king in an n-vertex tournament graph G is a vertex that can reach any other vertex v with a path of length at most two. A kingdom is a data structure that given any vertex v returns such a path from a king to v in $O(1)$ time. In this paper, we show how to maintain a kingdom while the tournament graph G undergoes updates. We consider both edge updates that flip the direction of an edge, and vertex updates that insert/delete vertices by activating/deactivating rows and columns of the graph's adjacency matrix.
For a *single* edge-flip, we show that after $O(n^{3/2})$ preprocessing time, we can maintain a kingdom in $O(1)$ time following the edge-flip. With $\tilde{O}(n^2)$ preprocessing time, we can support any *constant* number of edge-flips, vertex insertions, and vertex deletions in $O(\log n)$ time per operation. For an *arbitrary* number of edge-flips, we present a randomized algorithm that maintains a kingdom in $O(\log n)$ expected time following every edge-flip, and another algorithm that supports vertex insertions in $O(\sqrt{n})$ amortized time per insertion.

Keywords: Tournament, king, dynamic graphs, fault tolerance

1 Introduction

A tournament is a directed graph $G = (V(G), E(G))$ with exactly one edge between every pair of vertices. Already in 1953, Landau [5] observed that every tournament has a *king* – a vertex that can reach any other vertex with a (directed) path of length at most two. Indeed, it is not difficult to see that any vertex of a tournament with maximum out-degree is a king. Alternatively, the source of every maximal transitive subtournament is a king. See Maurer [8] for many basic mathematical properties of kings.

Both Landau and Maurer used kings in tournaments in order to analyze the mathematical structure of dominance hierarchies in animals. In such applications (as well as others such as sports competitions or elections), not only the identity of the king is important, but also the dominance hierarchy that the king induces (which we call a *kingdom*). A kingdom, given any vertex v, reveals the path (of length at most two) from the king to v. Formally, a kingdom is a directed spanning tree of depth at most 2 rooted at a king x. I.e., for every child u of x there is an edge $(x, u) \in E(G)$, and for every child v of u there is an edge $(u, v) \in E(G)$. Notice that not all out-neighbors of x in G must be children of x in the kingdom (it is possible that some of them are grandchildren).

In this paper, we are interested in maintaining a kingdom in a *dynamic* tournament G. That is, a tournament that undergoes edge-flips, and vertex insertions/deletions. Maintaining a king itself is actually trivial by simply maintaining the out-degree of every vertex (and reporting the maximal one as the king). Maintaining a kingdom however, is much more challenging.

The kingdom structure. There are several combinatorial and algorithmic questions that naturally arise regarding kingdoms. We call a king plus its children a *dominating out-star*. Notice that a dominating out-star D is a dominating set in the tournament. Namely, for every vertex $v \in V(G) \setminus V(D)$, there is a vertex $u \in V(D)$ such that $(u, v) \in E(G)$; we say that u *dominates* v. A natural question is to determine the smallest size of a dominating out-star. Let $f(n)$ denote the maximum over all n-vertex tournaments, of the minimum size of a dominating out-star. It is not difficult to show that $f(n) = (1 + o(1)) \log n$ (see Proposition 1) although an exact formula seems elusive. This problem is naturally related to the well-known problem of Erdős and Moser [4] on determining $h(n)$, the minimum over all n-vertex tournaments, of the maximum order of a transitive subtournament. Since every maximal transitive subtournament contains a dominating out-star, we have $f(n) \leq h(n)$. It is well known that $2 \log n \geq h(n) \geq \log n$ [4, 11]; unlike $f(n)$, the exact asymptotics of $h(n)$ is not known. We mention also that it is known that $g(n)$, the maximum over all n-vertex tournaments, of the minimum size of a dominating set (so clearly $g(n) \leq f(n)$) also satisfies $g(n) = (1 + o(1)) \log n$ [6, 12].

As for the algorithmic side, there are several results on king detection which also compute a corresponding kingdom [2, 3, 7, 10]. In particular, an $O(n^{3/2})$ time algorithm of Shen, Sheng, and Wu [10] computes a king and is easily seen to also compute a dominating out-star and a kingdom. It is also shown in [10] that any *deterministic* algorithm for finding a king requires $\Omega(n^{4/3})$ queries to the tournament adjacency matrix, hence no deterministic algorithm can run in $o(n^{4/3})$ time. Yet, it was observed by Abboud, Grossman, Naor, and Solomon [1] that there is an $O(n)$ randomized Las Vegas algorithm for detecting a king and computing a corresponding kingdom.

The main results in this paper extend these algorithms to the *dynamic* and *fault-tolerant* setting. That is, we are allowed to preprocess the graph, such that we can efficiently (either worst case or amortized) maintain the kingdom after any vertex/edge change. In a tournament, changing an edge means flipping its direction, inserting a vertex v additionally inserts an edge between v and every other vertex, and deleting a vertex v deletes all its incident edges. In a fault tolerant algorithm, we wish to preprocess the graph efficiently, such that after a single change (or perhaps constantly many changes) we can maintain the kingdom efficiently.

It is not difficult to see that a single edge-flip or vertex insertion or vertex deletion might cause a vertex to no longer be a king and can cause significant change to the kingdom. As we do not want to compute the new kingdom explicitly following every change, we will maintain the kingdom implicitly with a data structure that stores all the structural information of the resulting kingdom.

That is, given a vertex v, our data structure supports a constant time query $q(v)$ which outputs a path of length at most two from the newly computed king x to v. Formally, for a tournament G, a *Kingdom Structure* (KS) is a triple (x, Y, S) such that x is a king of G, Y is a list of (not necessarily all) out-neighbors of x so that $\{x\} \cup Y$ forms a dominating out-star, and S is a data structure that can be queried in *constant time* to return $q(v)$. Notice that a KS encodes the entire information of a kingdom.

Our results. Our first result is a fault tolerant algorithm that handles any constant number of changes efficiently. As an edge-flip is a special case of two vertex operations (deletion of a vertex v and reinsertion of v with one incident edge changed compared to v's original adjacencies), it suffices to describe the algorithm following only vertex operations. Formally, we assume that the tournament G is represented by its (binary) adjacency matrix M (indexed by $V(G)$), and that M includes additional rows and columns that are initially marked as *inactive*. A change (A, R) is given by a list of vertices R (active rows/columns of M that should be marked as inactive) and a list of vertices A (inactive rows/columns of M that should be marked as active). We never access an inactive entry in M, and accessing an active entry in M is done in constant time.

Theorem 1. *Given an n-vertex tournament G, for every constant k, we can preprocess G in $\tilde{O}(n^2)$ deterministic time or $\tilde{O}(n)$ expected time, so that following any set (A, R) of changes with $|A \cup R| \leq k$, we can compute a KS for the new tournament deterministically in $O(\log n)$ time. If $A = \emptyset$ (only removals are allowed) then the new KS can be computed in $O(1)$ time.*

Our second result is a dynamic algorithm for handling vertex insertions only. Each insertion is presented by activating a new row/column at the bottom/right of the current adjacency matrix M. Denoting by n the current number of vertices, each insertion is performed in amortized $O(\sqrt{n})$ time (i.e., if we start with a graph that has n vertices and perform $t = O(n)$ subsequent insertions, then the total time of all insertions is worst case $O(t\sqrt{n})$).

Theorem 2. *Given an n-vertex tournament G, there is a dynamic algorithm that, after preprocessing G in $O(n^{3/2})$ deterministic time or $\tilde{O}(n)$ expected time, can compute a KS in $O(\sqrt{n})$ amortized time following every new vertex insertion.*

We then move on to consider a dynamic algorithm for handling edge-flips. We present an algorithm that can handle any number of edge-flips in (nearly optimal) $O(\log n)$ expected time per flip. We assume that the sequence of edge-flips is chosen in advance by an oblivious adversary as he wishes but only according to the structure of the initial tournament graph G. The oblivious-adversary model is standard for measuring the performance of randomized dynamic algorithms. The sequence of edge-flips is oblivious to the coin tosses made by the algorithm, and the algorithm is oblivious to the sequence of edge-flips. Handling an adaptive-adversary is left as an open problem.

Theorem 3. *Given an n-vertex tournament G, there is a dynamic algorithm that, after preprocessing G in $\tilde{O}(n)$ expected time, can compute a KS in $O(\log n)$ expected time following every edge-flip.*

Finally, we show how to improve the above $O(\log n)$ bound to (optimal) $O(1)$ deterministic time in the special case where only a single edge-flip is allowed.

Theorem 4. *Given an n-vertex tournament G, we can preprocess G in $O(n^{3/2})$ time, so that following any single edge-flip, a KS of the resulting tournament can be computed in $O(1)$ time.*

Roadmap. In Section 2 we describe a very simple $O(n^2)$ time deterministic algorithm to obtain a KS for a tournament in which the dominating out-star has $O(\log n)$ vertices. This algorithm is used as an ingredient in the preprocessing part of Theorem 1 (which is presented in Section 3), and in the dynamic algorithm of Theorem 2 (which is presented in Section 4). The dynamic algorithm of Theorem 3 for handling edge-flips is presented in Section 5 and the single edge-flip algorithm of Theorem 4 is presented in Section 6.

2 Computing a KS with a Small Dominating Out-star

Recall that $f(n)$ denotes the maximum over all n-vertex tournaments, of the minimum size of a dominating out-star. It is easy to see that $f(n) \leq \log(n+1)$ by repeatedly finding a maximum out-degree vertex and eliminating its out-neighbors. Formally,

Proposition 1. $f(n) \leq \log(n+1)$.

Proof. It suffices to prove that every tournament with $2^k - 2$ vertices has a dominating out-star of order at most $k-1$. This clearly holds for $k = 2$. Proceeding inductively, let G be a tournament with $n = 2^k - 2$ vertices and let v be a vertex with maximum out-degree. Since the maximum out-degree in a tournament is at least $(n-1)/2$, the out-degree of v is at least $2^{k-1} - 1$. Remove v and its out-neighbors from G to obtain a tournament on at most $2^{k-1} - 2$ vertices which, by induction, has a dominating out-star of order at most $k-2$. Such an out-star, together with v, is a dominating out-star for G. □

Corollary 1. *Given an n-vertex tournament G, we can construct a KS (x, Y, S) for G with $|Y| = O(\log n)$ in $O(n^2)$ deterministic time or $O(n)$ expected time.*

Proof. Suppose that M is the (binary) adjacency matrix of G (so $M(u,v) = 1$ if and only if $(u,v) \in E(G)$). Initially, we set all vertices of G as *unmarked*. Also we set an array P indexed by $V(G)$ where $P(v)$ is either $\emptyset$ or else $P(v) = u$ such that $(u,v) \in E(G)$. Finally, let Y be an (initially empty) list and let x be a variable that at the end will contain a king.

Notice that at any given time we can compute a vertex of maximum out-degree in the subtournament of G induced by the unmarked vertices in $O(nd)$

time where d is the number of unmarked vertices. We repeatedly proceed as in the proof of Proposition 1. We find a vertex of maximum out-degree in the subtournament of the yet unmarked vertices. Suppose this vertex is y. Set y as marked. For each unmarked out-neighbor v of y, set $P(v) = y$ and set v as marked. If all vertices $V(G)$ are now marked, set $x = y$ and halt. Otherwise, add y to Y and repeat.

As at each round, the number of unmarked vertices reduces by a factor of at least 2, so the overall runtime is $O(n^2)$ as claimed. Alternatively, this can be done in $O(n)$ expected time, if instead of the maximum-degree vertex we randomly choose an unmarked vertex (whose degree is expected to be at least half of the maximum-degree vertex). Notice that (x, Y, P) is indeed a KS, as given $v \in V(G)$, either $(x, v) \in E(G)$ and the query $q(v)$ returns the single edge (x, v) or $(v, x) \in E(G)$ and the query $q(v)$ returns the path $(x, P(v), v)$ where $P(v) \in Y$. □

Recall from the introduction that the algorithm of Corollary 1 is not optimal. The algorithm from [10] computes a KS in $O(n^{3/2})$ time. However, that algorithm only guarantees that $|Y| \leq \sqrt{n}$ while the algorithm from Corollary 1 guarantees $|Y| = O(\log n)$; which will be important for us later. The algorithm from [1] computes a KS in expected $O(n)$ time, but it only guarantees that $|Y|$ has expected size $O(\log n)$. We can use the latter as an alternative for Corollary 1 if we settle for a randomized algorithm. However, in our applications, the running time of Corollary 1 will not be a bottleneck as we shall use it as a subroutine for relatively small subtournaments.

Another noteworthy observation is that in all of these algorithms, the computed dominating out-star is a *transitive* subtournament of G, and that the king is the source of that transitive subtournament. One may wonder whether it is always the case that a smallest dominating out-star can be taken to be a transitive subtournament. This, however, is false as the following example demonstrates.

Let G be tournament consisting of n vertices, with four vertices designated a, x_1, x_2, x_3 and the remaining $n - 4$ vertices are partitioned into three (say equal) parts V_1, V_2, V_3. Define the edges to be (a, x_1), (a, x_2), (a, x_3), (x_1, x_2), (x_2, x_3), (x_3, x_1). Further, for $i \in \{1, 2, 3\}$, all vertices of V_i are in-neighbors of a and of x_j for $j \neq i$ and are out-neighbors of x_i. Finally, orient the edges of the subtournament induced by $V_1 \cup V_2 \cup V_3$ at random. Clearly, $\{a, x_1, x_2, x_3\}$ is a dominating out-star where a is the king, and $\{a, x_1, x_2, x_3\}$ is not transitive. It is easily checked that with high probability (as n grows), there is no dominating out-star with fewer than four vertices and that $\{a, x_1, x_2, x_3\}$ is the only dominating out-star with four vertices.

Finally, it is easy to see that $f(3) = f(4) = f(5) = f(6) = 2$, and that $f(n) = 3$ for all $n = 7, \ldots, 14$. To see, say, the latter, notice that $f(14) \leq 3$ by Proposition 1, while $f(7) \geq 3$ by considering the so-called Paley tournament [9] on seven vertices. This is the (unique) regular tournament on seven vertices (i.e., the in-degree and out-degree of each vertex is 3) for which the out-neighbors of every vertex form a directed triangle. Clearly, it cannot be dominated by two vertices. Thus, for all $2 \leq n \leq 14$, we have that $g(n) = f(n)$ (recall that

$g(n) \leq f(n)$ is the maximum over all n-vertex tournaments of the minimum size of a dominating set). A simple probabilistic argument shows that $g(n) = (1 + o(1)) \log n$, hence so does $f(n)$. It seems interesting to determine whether $f(n) = g(n)$ for all n.

3 A Fault Tolerant Algorithm

Proof of Theorem 1. The idea is to repeatedly apply Corollary 1 and store the information in a rooted tree $\mathcal{T}$ of depth k. The nodes of $\mathcal{T}$ correspond to subgraphs obtained from G by removing at most k vertices. The root of $\mathcal{T}$ corresponds to the entire graph G, and stores a KS (x, Y, S) for G obtained using Corollary 1. The children of the root are subgraphs of the form $G \setminus \{v\}$. The crucial observation is that we only need to consider vertices v that belong to Y (other vertices do not affect the KS), and by Corollary 1 we have $|Y| = O(\log n)$. I.e., the root has only $O(\log n)$ children. We repeat this process until depth k.

Formally, the nodes of $\mathcal{T}$ are pairs of the form $(Q, (x_Q, Y_Q, S_Q))$ where Q is a subtournament of G and (x_Q, Y_Q, S_Q) is a KS for Q obtained using Corollary 1. The root of $\mathcal{T}$ is $(G, (x_G, Y_G, S_G))$. For a node $(Q, (x_Q, Y_Q, S_Q))$ of $\mathcal{T}$ its children are defined as follows: For each $v \in \{x_Q\} \cup Y_Q$ there is a child $(Q_v, (x_{Q_v}, Y_{Q_v}, S_{Q_v}))$ where $Q_v = Q \setminus \{v\}$ is the tournament obtained from Q after removing v. We construct $\mathcal{T}$ only until level k (the level of the root is 0). By Corollary 1, the number of children of a vertex of $\mathcal{T}$ at level r is at most $\log(n + 1 - r)$ as every tree vertex at level r corresponds to a tournament with $n - r$ vertices, whose corresponding KS is constructed in $O((n - r)^2) = O(n^2)$ deterministic time or $O(n)$ expected time. Hence, the entire tree $\mathcal{T}$ is of size $O(\log^k n)$ and is constructed in $O(n^2 \cdot \log^k n) = \tilde{O}(n^2)$ deterministic time or $\tilde{O}(n)$ expected time, as required.

We next describe how changes are handled. Suppose (A, R) is a set of changes where A is a set of newly added vertices, $R \subset V(G)$ is a set of removed vertices, and $|A \cup R| \leq k$. We shall first handle the removed vertices R. To this end, we locate in $\mathcal{T}$ a node $(Q, (x_Q, Y_Q, S_Q))$ where $V(G) \setminus V(Q) \subseteq R$ and $(\{x_Q\} \cup Y_Q) \cap R = \emptyset$. This can be done by traversing $\mathcal{T}$ from the root as we next describe: Initialize $R^* = R$ and proceed as follows. If the root $(G, (x_G, Y_G, S_G))$ already has the required property, we halt at the root. Otherwise, let $v \in (\{x_G\} \cup Y_G) \cap R^*$. Set $R^* = R^* \setminus \{v\}$ and traverse to the child $(Q, (x_Q, Y_Q, S_Q))$ where $V(G) \setminus V(Q) = \{v\}$. Now continue in the same manner. Suppose we are now at some tree node $(Q, (x_Q, Y_Q, S_Q))$. If $(\{x_Q\} \cup Y_Q) \cap R^* = \emptyset$ we halt at this tree vertex. Otherwise, let $v \in (\{x_Q\} \cup Y_Q) \cap R^*$. Set $R^* = R^* \setminus \{v\}$ and continue to the child $(Q_v, (x_{Q_v}, Y_{Q_v}, S_{Q_v}))$ where $V(Q) \setminus V(Q_v) = \{v\}$. Notice that since $|R| \leq k$ and since $|R^*|$ decreases by 1 as we traverse to the next child, the procedure must halt at some tree node $(Q, (x_Q, Y_Q, S_Q))$ after k steps as required. Also note that the time required to traverse the tree is $O(\log n)$ as each list Y_Q has $O(\log n)$ elements. In fact, we may further reduce the traversal time to $O(1)$ since recall that in the KS of Corollary 1, S_Q is just an array $P = P_Q$, where $P(v) = \emptyset$ if $v \in x_Q \cup Y_Q$. So we just need to query $P(v)$ for every $v \in R^*$ and see if it is empty.

Once we arrive at our desired tree node $(Q, (x_Q, Y_Q, S_Q))$ we have the property that $V(G) \setminus V(Q) \subseteq R$ and $(\{x_Q\} \cup Y_Q) \cap R = \emptyset$. Recall also that S_Q is just an array $P = P_Q$ such that $P(v) = \emptyset$ for $v \in \{x_Q\} \cup Y_Q$ and otherwise $P(v)$ is a vertex of $x_Q \cup Y_Q$ which dominates v. Let $G^* = G \setminus R$, namely the subtournament obtained from G after removing R. Define $S_{G^*} = (P, R)$ (namely, the data structure has two components, the array P and the set R of removed vertices). We claim that $(G^*, (x_Q, Y_Q, S_{G^*}))$ is a KS for G^*. Indeed, given $v \in V(G)$ we wish to return $q(v)$. If $v \in R$ then we return that the query $q(v)$ is invalid. Otherwise, if $v = x_Q$ we return $q(v) = x_Q$, if $(x_Q, v) \in E(G)$ then $(x_Q, v) \in E(G^*)$ and we return $q(v) = (x_Q, v)$. Otherwise, we return $q(v) = (x_Q, P(v), v)$ (notice that this is a valid path of length 2 in G^*). We have proved that a KS for G^* can be constructed in constant time, as required.

We next need to handle the vertex addition set A. Starting from G^* (which is just G with $|R|$ rows and columns of the adjacency matrix of G marked inactive) we add the vertices of A (recall, this means that we activate a bottom row and a rightmost column of the adjacency matrix) one by one. We will show how this can be done in $O(\log n)$ time for each addition. To simplify notation, suppose that $A = \{a_1, \ldots, a_t\}$ where the row/column corresponding to a_i is the $n + i$ row/column of the adjacency matrix. Recall that the KS for G^* is the $(x_Q, Y_Q, S_{G^*}) = (x_Q, Y_Q, (P, R))$ obtained by the deletion procedure above. Starting with a_1, we check whether (y, a_1) is an edge for some $y \in \{x_Q\} \cup Y_Q$. If so, simply set $P(a_1) = y$. This takes $O(\log n)$ time since $|\{x_Q\} \cup Y_Q| = O(\log n)$. Otherwise, a_1 dominates all vertices of the dominating star $\{x_Q\} \cup Y_Q$ of G^* so it is a king of $G^* \cup \{a_1\}$. Therefore, we just modify the KS by setting $Y_Q := Y_Q \cup \{x_Q\}$ and $x_Q := a_1$. Similarly, when adding a_i, if (y, a_i) is an edge for some $y \in \{x_Q\} \cup Y_Q$ then we set $P(a_i) = y$. This search takes $O(\log n)$ time since $|Y_Q| = O(i + \log n) = O(\log n)$ since $i = O(1)$. Otherwise, a_i dominates all vertices of the dominating star of $G^* \cup \{a_1, \ldots, a_{i-1}\}$, hence it is a king of $G^* \cup \{a_1, \ldots, a_{i-1}\}$, and we modify the KS by setting $Y_Q := Y_Q \cup \{x_Q\}$ and $x_Q := a_i$. As $|A| \leq k$ is constant, the entire procedure takes $O(\log n)$ time and we arrive at a KS for $G^* \cup A$, as required. □

4 A Dynamic Insertion Algorithm

Proof of Theorem 2. Recall that we start from an n-vertex tournament G given by its adjacency matrix, and we want to support every new insertion by exposing a new row at the bottom of the matrix and a new column at the right of the matrix. Our goal is to preprocess G and obtain a KS for it, and then show that following $\Theta(n)$ subsequent insertions, the total number of operations required for maintaining a KS during t insertions is $O(t\sqrt{n})$.

We initially preprocess G in $O(n^{3/2})$ time using the algorithm from [10]. Recall that this computes a KS of the form (x, Y, P) where $|Y| \leq \sqrt{n}$ and P is an array indexed by $V(G)$ such that $P(v) = \emptyset$ if $v \in \{x\} \cup Y$ and otherwise $P(v) \in \{x\} \cup Y$ and $(P(v), v) \in E(G)$.

We insert new vertices one by one. In most cases, an insertion takes only $O(\sqrt{n})$ time, but from time to time it will take longer, yet still not violate the

claimed amortized bound. Let $Y_0 := \{x\} \cup Y$ (as we will modify Y and x, we need to remember the original dominating out-star). We shall also maintain a list Z consisting of "worrisome" newly inserted vertices. Initially, $Z = \emptyset$. We use G to refer to the *current* tournament and its adjacency matrix.

Consider the first vertex v to be inserted. We check in $O(\sqrt{n})$ time whether there is some $y \in \{x\} \cup Y = Y_0$ such that (y, v) is an edge. If so, then set $P(v) = y$. Otherwise, v dominates the dominating out-star of G, so v is a king of $G \cup \{v\}$. We then modify $Y := \{x\} \cup Y$, $x := v$, and $P(v) := \emptyset$ to obtain the new KS for $G \cup \{v\}$. We also add v to Z. We continue in this manner for the first $\sqrt{n}$ insertions. I.e., when inserting v, if there is some $y \in \{x\} \cup Y$ where (y, v) is an edge then we set $P(v) = y$. If $y \in (\{x\} \cup Y) \setminus Y_0$ then we add v to Z. Otherwise, we modify $Y := \{x\} \cup Y$, $x := v$, and $P(v) := \emptyset$ and add v to Z. After doing $\sqrt{n}$ insertions, we have $|Y| \leq 2\sqrt{n}$ and $|Z| \leq \sqrt{n}$, so it takes $O(\sqrt{n})$ time to do each of the first $\sqrt{n}$ insertions.

Once we arrive at insertion number $\sqrt{n}$ we "rearrange" our data structure. Notice that $(\{x\} \cup Y) \setminus Y_0 \subseteq Z$ and that $Y \setminus Y_0$ is a prefix of Y. Using the algorithm of Corollary 1, we compute a KS for the subtournament $G[Z]$ induced by Z. Notice that $G[Z]$ is accessible in the bottom right $\sqrt{n} \times \sqrt{n}$ of the present adjacency matrix (so we have $O(1)$ access to its edges). The time required for computing the KS of $G[Z]$ is therefore $O(\sqrt{n}^2) = O(n)$ and the resulting dominating out-star is of size at most $O(\log \sqrt{n}) = O(\log n)$. Let the obtained KS of $G[Z]$ be (x_Z, Y_Z, P_Z). We merge it with our current KS (x, Y, P) of G as follows. We set $Y := Y_0 \cup Y_z$, $x := x_Z$ and $P(v) := P_Z(v)$ for each $v \in Z$ (for $v \notin Z$, we keep $P(v)$ intact). Observe that the entire rearranging operation requires only $O(n)$ time, which is amortized $O(\sqrt{n})$ time over the previous $\sqrt{n}$ "cheap" insertions. Crucially, however, is that now $|Y| \leq \sqrt{n} + \log n$.

We may now perform $\sqrt{n}/\log n$ iterations of $\sqrt{n}$ insertions each, where at the end of each iteration, we perform the rearrangement procedure just described for the Z vertices inserted during the iteration. Each such iteration extends the current Y by at most $\log n$ vertices, so each insertion still requires $O(\sqrt{n})$ amortized time. Note that we have now already added $n/\log n$ vertices. We could have, in fact, performed $\sqrt{n}$ iterations to account for n insertions. This is perfectly fine, but if we do so, the size of Y could be as large as $\sqrt{n} \log n$, so this yields an insertion algorithm with amortized time $O(\sqrt{n} \log n)$ which is only slightly larger than what we are claiming. Let us see how we can further avoid this $\log n$ factor.

Once we perform the $\sqrt{n}/\log n$ iterations described above (call these "level 1" iterations), we can rearrange the union of the sets Z of all iterations. Notice that such union of Z's has at most $n/\log n$ vertices. We can now use the algorithm of [10] to obtain a KS with dominating out-star of size $(n/\log n)^{1/2}$ and the time to construct it is $(n/\log n)^{3/2}$ which is only $O((n/\log n)^{1/2})$ when amortized over all $n/\log n$ insertions. We then merge it as described earlier with the current KS to obtain a KS for the current tournament whose Y is only of size $\sqrt{n}+(n/\log n)^{1/2}$. We can now perform $\sqrt{\log n}$ "level 2" iterations where each iteration invokes a "level 1 iteration". Once done, we have inserted $n/\sqrt{\log n}$ vertices, and use [10]

to rearrange, in $(n/\sqrt{\log n})^{3/2}$ time for a KS of the current tournament whose Y is of size only $\sqrt{n} + (n/\sqrt{\log n})^{1/2}$. Continuing in this way, we perform "level t" iterations, to obtain a KS whose dominating out-star is of size at most $\sqrt{n} + n^{1/2}/(\log n)^{1/2^t}$ and the time for rearranging using [10] is $(n/(\log n)^{1/2^{t-1}})^{3/2}$. Once we have $t = O(\log\log\log n)$, we have inserted $\Theta(n)$ vertices and the current Y is still of size $O(\sqrt{n})$. This is because $\sum_{t=1}^{\log\log\log n}(1/\log n)^{1/2^t} = O(1)$. The amortized time for each insertion is therefore at most $O(\sqrt{n})$. □

5 A Dynamic Edge-Flip Algorithm

Proof of Theorem 3. We will use the following simple claim.

Claim 1. *Given any n-vertex tournament, we can find a vertex with out-degree at least $n/4$ in expected $O(n)$ time.*

Proof. We claim that in every tournament there are at least $(n-1)/2$ vertices whose out-degree is at least $n/4$. Suppose not, and let X be the set of vertices whose out-degree is less than $n/4$, so we have that $|X| \geq n/2 + 1$. Consider the subtournament induced by X and a vertex with maximum out-degree in it. Already in that subtournament, the out-degree of this vertex is at least $(|X| - 1)/2 \geq n/4$, a contradiction. We therefore conclude that a randomly chosen vertex v has out-degree at least $n/4$ with probability at least $1/2 - 1/2n \geq 1/3$. The lemma follows since checking v's out-degree is done in $O(n)$ time. □

Let $F = \{f_1, \ldots, f_m\}$ be any sequence of flips. Let G^i be the tournament obtained from G after performing the flips $f_1, \ldots, f_i$ (so that $G^0 = G$). I.e., if $f_i = \{x, y\}$ then G^i is obtained from G^{i-1} by flipping the edge between x and y.

We first define the components of our data structure that contains the KS following the i'th flip. Initially, the data structure will contain the KS for $G^0 = G$.

- The adjacency matrix A of the current tournament G^i.
- A variable K holding a king of G^i.
- An array D of vertices forming a dominating out-star of G^i. We call the vertices of D *backbone vertices*. It will always hold that if $D = \{v^i_1, \ldots, v^i_{t_i}\}$ (i.e., $D[j] = v^i_j$), then $t_i = O(\log n)$ and $(v^i_j, v^i_{j'}) \in E(G^i)$ if and only if $j > j'$. Furthermore, the last vertex $v^i_{t_i}$ is K.
- An array B of size n such that if $v = v^i_j$ is a backbone vertex, then $B[v] = -j$. Otherwise, $B[v]$ is the index of a backbone vertex such that $(D[B[v]], v) \in E(G^i)$. We call the set of all vertices v for which $B[v] = j$, the *bag* of v^i_j.
- An array L of size n such that the first segment of L contains all vertices in the first bag, as well as the first backbone vertex. The second segment of L contains all vertices in the second bag, as well as the second backbone vertex and so on. The ordering within each segment is arbitrary.
- An array P such that $P[j]$ is the index in L of the last vertex of the j'th segment. If j is larger than the number of backbone vertices, then $P[j]$ is irrelevant (and will not be used).
- An array U of *unmarked vertices*. These can be in any order. We will only

use U when updating the data structure after a flip. Once the current update is complete, U is empty.
• A variable M holding the number of marked vertices (i.e., $M = n - |U|$).

Let us observe that a query can be performed in $O(1)$ time. Given a vertex v, we check $B[v]$. If $B[v]$ is negative (so v is a backbone vertex), then if $v \neq K$ we return the path (K, v), and otherwise we return the path (K). If $B[v]$ is positive (i.e., it is equal to the index of the bag containing v), then let $D[B[v]]$ be its dominator. If $D[B[v]] \neq K$, then return the path $(K, D[B[v]], v)$. Otherwise, return the path (K, v).

Let us next see how we set our data structure for the initial tournament $G = G^0$ whose adjacency matrix A is given. We use Claim 1 to locate (in expected $O(n)$ time) a vertex v (so that $v = v_1^0$ will be the first backbone vertex of G^0) with out-degree at least $n/4$. For each out-neighbor u of v, we set $B[u] = 1$. We also set $B[v] = -1$ and set $D[1] = v$. All out-neighbors of v and v itself are assigned consecutively to the initial segment of L. We set M to be the number of out-neighbors of v plus one. We set $P[1] = M$ (since this is the index in L of the end of the first segment). We set U to be the list of all in-neighbors of v.

Locating the j'th backbone vertex of G^0. More generally, we describe how to locate the j'th backbone vertex v of G^0. We will later, during updates, use this procedure to determine a j''th backbone vertex of G^i, if necessary. We repeatedly choose an index p at random in $\{1, \ldots, n - M\}$ (as we must choose an unmarked vertex, and the number of unmarked vertices is $n - M$), and let $v = U[p]$ (so v is a randomly-chosen unmarked vertex). For each vertex $u \in U$ we query $A[v, u]$ to find the out-degree of v in the remaining subtournament on the unmarked vertices. If this number is larger than $(n - M)/4$ we are happy with v, otherwise we choose a different v. By Claim 1, the expected time to find v is $O(n - M)$. Once v is found we continue as follows: For each unmarked out-neighbor u of v (i.e., in $O(n - M)$ time), we set $B[u] = j$. We also set $B[v] = -j$ and set $D[j] = v$. All unmarked out-neighbors of v and v itself are assigned consecutively to the next segment of L. Note that we know that this segment starts at index $P[j - 1] + 1$ of L. We increase M by the number of unmarked out-neighbors of v plus one. We set $P[j] = M$ (since this is now the index in L of the last vertex of the j'th segment). We set U to be the list of all unmarked in-neighbors of v (as these now remain unmarked). Note that the total expected time to locate the j'th backbone vertex and perform the data structure modifications is $O(n - M)$. Also observe that the case $j = 1$ described in the previous paragraph can be merged to the general case if we just initialize $U = [n]$, $M = 0$ and $P[0] = 0$.

We repeatedly apply the above procedure increasing j by one each time, halting when $M = n$ and set K to be the backbone vertex v in the final iteration.

Notice that after this preprocessing stage, the data structure satisfies all required properties listed earlier. Furthermore, as in each iteration the number of unmarked vertices decreases by a fraction of at least 3/4, we have that the size of the backbone is at most $O(\log n)$. Also, the expected runtime of the j'th iteration is $O((3/4)^j n)$ and hence expected $O(n)$ in total.

The update procedure. Let B^i_j be the *bag* of v^i_j, and let G^i_j be the subtournament consisting of all backbone vertices $v^i_j, v^i_{j+1}, \ldots, v^i_{t_i}$ and their bags. In particular, $G^i_1 = G^i$. We shall maintain the following additional property for our data structure: if $x \in B^i_j \cup \{v^i_j\}$, then there is an edge from x to every backbone vertex v^i_k with $k < j$. Clearly, this property holds after the preprocessing stage (i.e., for $i = 0$). Let us now describe the update procedure. Suppose we arrive at flip $f_{i+1} = \{x, y\}$ where the current tournament is G^i. We consider three cases:

1. Both x and y are not backbone vertices of G^i. In this case we only need to modify $A[x, y]$ (in constant time) to obtain the adjacency matrix of G^{i+1}. Note that we can tell if a vertex v is backbone or not by just checking if $B[v]$ is positive or negative.
2. One of $\{x, y\}$ (say, w.l.o.g x) is a backbone vertex and the other is not, and it also holds that $x = v^i_j$, $y \in B^i_k$, and $k < j$. Again, we modify $A[x, y]$ to obtain the adjacency matrix of G^{i+1}. Note that we can determine j as $j = -B[x]$ and determine k as $k = B[y]$.
3. Otherwise, let k be the smallest index such that v^i_k is a backbone vertex and $v^i_k \in \{x, y\}$. Again, k is found by examining $B[x]$ and $B[y]$. Notice that both x and y are vertices of G^i_k. We modify $A[x, y]$ to obtain the adjacency matrix of G^{i+1} (in particular, G^i_k becomes G^{i+1}_k). Next, we recompute a KS and a backbone of G^{i+1}_k to obtain a backbone and corresponding bags. Informally, this is done as in the aforementioned preprocessing algorithm, only we start at its k'th iteration (see details below). Finally, we concatenate the obtained backbone and bags to the end of the prefix backbone $\{v^i_1, ..., v^i_{k-1}\}$ and their bags (this is the prefix backbone of G^i which remains intact) to obtain a backbone and bags for G^{i+1}.

It is easy to see that cases 1 and 2 require $O(1)$ time and maintain all the data structure properties. Let us now be more formal regarding case 3. Notice that we need to "reset" our data structure such that it looks as if it now starts the k'th iteration (as the first $k-1$ bags and backbone vertices remain intact). Let $w = P[k-1]+1$ and let U be all the vertices of $L[w, \ldots, n]$ in U (so we effectively "unmark" all vertices of G^i_k). Also set $M = P[k-1]$ to be the number of marked vertices. Now set $j = k$ and use the aforementioned procedure *locating the j'th backbone vertex*, performing it repeatedly until no unmarked vertices are left. This will yield the backbone vertices $v^{i+1}_k, v^{i+1}_{k+1}, \ldots, v^{i+1}_{t_{i+1}}$ and their bags, and modify the data structure accordingly, maintaining its properties.

It remains to analyze the running time of case 3. Note that the running time is (in expectation) linear in the number of vertices of G^i_k (i.e., $O(|V(G^i_k)|)$ time). In particular, the larger k is, the less time the update is expected to take, as the sizes of the bags geometrically decrease (in the worst case, if $k = 1$, we actually run the entire preprocessing algorithm from scratch in $O(n)$ time). We next prove that case 3 is not expected to occur frequently.

For a vertex v, a flip stage i, and an index j, Let $p(v, i, j)$ be the probability that v is the j'th backbone vertex of G^i, i.e., $v = v^i_j$. Recall that $1 \leq j \leq O(\log n)$. Consider the last flip-stage that changed v^i_j (this could be at flip-stage i or

earlier). Then, by Claim 1, v_j^i was chosen at random among a pool of at least $|V(G_j^i)|/4$ vertices and the probability that v was chosen to be v_j^i is therefore at most $4/|V(G_j^i)|$. Thus, $p(v,i,j) \leq 4/|V(G_j^i)|$.

Fix $1 \leq k \leq O(\log n)$. Consider the $(i+1)$'th flip $f_{i+1} = \{x, y\}$. The probability that $x = v_k^i$ or $y = v_k^i$ is therefore at most $4/|V(G_k^i)| + 4/|V(G_k^i)| = 8/|V(G_k^i)|$. Hence, the probability that case 3 occurred and the corresponding value of k in it is our chosen k is at most $8/|V(G_k^i)|$. The expected runtime in this case is $O(|V(G_k^i)|)$. Hence, the a priori expected runtime is only $O(|V(G_k^i)| \cdot 8/|V(G_k^i)|) = O(1)$. As there are $O(\log n)$ possible choices for k, the expected runtime of an update is $O(\log n)$, as required. □

6 A Single Edge-Flip Algorithm

Proof of Theorem 4. Let (x, Y, S) be a KS for G. Recall that x is a king, Y is a list of (some) out-neighbors of x, and S is a data structure such that given $v \in V(G)$ where $(v, x) \in E(G)$, S may be queried in constant time to yield a vertex $y \in Y$ such that (x, y, v) is a path of length 2. We next define the components (and properties) of S. These properties hold after preprocessing and, when specified, also hold after a single edge-flip.

1. A variable m (meaning *marked*). After preprocessing, $m = empty$, but following an edge-flip, m may be assigned a vertex.
2. An array P indexed by $V(G)$, such that $P(v)$ is a list of at most two vertices. After preprocessing, it holds for each $v \in \{x\} \cup Y$ that $P(v)$ is empty, and it holds for $v \notin \{x\} \cup Y$ that $P(v)$ contains at most two vertices, the first of which, say y, is in $\{x\} \cup Y$ such that $(y, v) \in E(G)$. After an edge-flip, it will be the case that if v is not a vertex of the current dominating out-star, then the first *unmarked* vertex of $P(v)$ (there will always be such a vertex) is a vertex of the current dominating out-star.
3. After preprocessing, suppose that $Y = \{y_1, \ldots, y_t\}$. Then, if $v \notin \{x\} \cup Y$ and the first vertex of $P(v)$ is y_j, it shall always be the case that $(v, y_i) \in E(G)$ for $i < j$. Also, if the first vertex of $P(v)$ is x, then $(v, y_i) \in E(G)$ for all $1 \leq i \leq t$.
4. An array W indexed by $V(G)$ such that after preprocessing, $W(v)$ is the list of all vertices in $\{x\} \cup Y$ that are in-neighbors of v. Furthermore, if $v \notin \{x\} \cup Y$ then the first vertex of $W(v)$ is also the first vertex of $P(v)$.
5. After preprocessing, let Z be the set of all vertices v in $V(G) \setminus (\{x\} \cup Y)$ such that the first vertex of $P(v)$ is x (in particular these vertices are directly dominated by the original king x, but notice that there may also be other vertices not in Z which are dominated by the king). Assuming $Z \neq \emptyset$, we compute a king z for $G[Z]$ (the subtournament induced by Z), a list Y_Z such that $\{z\} \cup Y_Z$ is a dominating out-star of $G[Z]$, and an array P_Z such that for each $v \in Z \setminus (\{z\} \cup Y_Z)$, we have that $(P_Z(v), v) \in E(G)$ and $P_Z(v) \in (\{z\} \cup Y_Z)$. In particular, (z, Y_Z, P_Z) is a KS for $G[Z]$. Now, for each $v \in Z \setminus (\{z\} \cup Y_Z)$, we will have that the *second* vertex of $P(v)$ is $P_Z(v)$ (recall that the first vertex of $P(v)$ is x).

We invoke the algorithm of [10] twice to construct the KS specified above on the original G. We first invoke it for G to obtain x, Y, and the array P where $P(v) = \emptyset$ if $v \in \{x\} \cup Y$ and otherwise $P(v)$ is a single vertex in $\{x\} \cup Y$ which dominates v. Recall that Item 2 above is an artifact of the algorithm (and that $\{x\} \cup Y$ is a transitive subtournament). We then gather all vertices of Z as in Item 5 and invoke [10] again on $G[Z]$ to obtain z, Y_Z, P_Z and the modified two-element lists $P(v)$ for $v \in Z \setminus (\{z\} \cup Y_Z)$. Both invocations require $O(n^{3/2})$ time, and the construction of W in Item 4 is done in $O(|Y|n) = O(n^{3/2})$ time, since $|Y| \leq \sqrt{n}$. Altogether, the preprocessing takes $O(n^{3/2})$ time, as claimed.

Let us next be formal about our query procedure that may be used either on the original graph after preprocessing, or after a single edge-flip (where the correctness after an edge-flip is proved below). Here we refer to G, x, Y, S as either the original tournament and its KS or the tournament and its modified KS after a single flip. Given $v \in V$, if $v = x$, then return $q(v) = (x)$. Else, if $(x, v) \in E(G)$ return $q(v) = (x, v)$, else return (x, y, v) where y is the first unmarked vertex of $P(v)$. Clearly this takes $O(1)$ time and is correct for the original tournament.

Now consider an edge-flip. There are several cases:

- If (u, v) is flipped to (v, u) where both $u, v \notin \{x\} \cup Y$. This can be verified in constant time as for such vertices we have that $P(v)$ and $P(u)$ are both not empty. In this case we do nothing. We keep the same KS and this is valid since the dominating out-star has not changed.
- If (v, y) is flipped to (y, v) for $v \notin \{x\} \cup Y$ and $y \in \{x\} \cup Y$. Here again we know that $y \in \{x\} \cup Y$ since $P(y)$ is empty and again we do nothing. This is valid since v is still dominated by the first vertex of $P(v)$ (which can't be y).
- If (y, y') is flipped to (y', y) for $y, y' \in Y$. Once again, we do nothing, as $\{x\} \cup Y$ is still a dominating out-star.
- If (y, v) is flipped to (v, y) for $v \notin \{x\} \cup Y$ and for $y \in \{x\} \cup Y$. If y is not the first vertex of $P(v)$, then we do nothing. If y is the first vertex of $P(v)$, we must search for another vertex to dominate v. Since the first vertex of $W(v)$ is also y, we look for the next vertex. If y' is such, then just set $P(v) = y'$. If, however, there is no next vertex, then v dominates all of $\{x\} \cup Y$. Hence, v is a new king, so we can modify the KS to be $Y := \{x\} \cup Y$, $x := v$, and we do not need to change the array P at all (all vertices except v keep their dominators).
- If (x, y) is flipped for $y \in Y$. Then x is no longer guaranteed to be king. There are now two possibilities. If $Z = \emptyset$ then we can just define $P(x) := y$, set the new king as y_t (the last vertex of Y; recall that Y still induces a transitive subtournament) and modify $Y = Y \setminus \{y_t\}$. If, however, Z is not empty, we mark $m = x$, add $Y := Y \cup Y_Z$ (so that the order of Y is $(y_1, \ldots, y_t, Y_Z)$ and note that this takes $O(1)$ time by concatenating lists), and set the new king to be z. Notice that queries work correctly even for vertices that were originally in Z, as when scanning their list $P(v)$, the first vertex (i.e., x) is marked so is not returned, and their second vertex is unmarked and returned correctly. □

References

1. A. Abboud, T. Grossman, M. Naor, and T. Solomon. From donkeys to kings in tournaments. In *32nd ESA*, volume 308, pages 3:1–3:14, 2024.
2. M. Ajtai, V. Feldman, A. Hassidim, and J. Nelson. Sorting and selection with imprecise comparisons. *ACM Trans. Algorithms*, 12(2):19:1–19:19, 2016.
3. A. Biswas, V. Jayapaul, V. Raman, and S. R. Satti. Finding kings in tournaments. *Discret. Appl. Math.*, 322:240–252, 2022.
4. P. Erdős and L. Moser. On the representation of directed graphs as unions of orderings. *Math. Inst. Hung. Acad. Sci*, 9:125–132, 1964.
5. H. G. Landau. On dominance relations and the structure of animal societies: III The condition for a score structure. *The Bulletin of Mathematical Biophysics*, 15:143–148, 1953.
6. X. Lu, D.-W. Wang, and C. K. Wong. On the bounded domination number of tournaments. *Discrete Mathematics*, 220(1-3):257–261, 2000.
7. N. S. Mande, M. Paraashar, and N. Saurabh. Randomized and quantum query complexities of finding a king in a tournament. In P. Bouyer and S. Srinivasan, editors, *43rd FSTTCS*, pages 30:1–30:19, 2023.
8. S. B. Maurer. The king chicken theorems. *Mathematics Magazine*, 53(2):67–80, 1980.
9. R. E. A. C. Paley. J. math. and phys. *The Bulletin of Mathematical Biophysics*, 12:311–320, 1933.
10. J. Shen, L. Sheng, and J. Wu. Searching for sorted sequences of kings in tournaments. *SIAM Journal on Computing*, 32(5):1201–1209, 2003.
11. R. Stearns. The voting problem. *The American Mathematical Monthly*, 66(9):761–763, 1959.
12. E. Szekeres and G. Szekeres. On a problem of Schütte and Erdös. *The Mathematical Gazette*, pages 290–293, 1965.