Bottleneck Paths and Trees and Deterministic Graphical Games

Shiri Chechik[1], Haim Kaplan[2], Mikkel Thorup[3], Or Zamir[4], and Uri Zwick[5]

1 **Blavatnik School of Computer Science, Tel Aviv University, Israel**
shiri.chechik@gmail.com
2 **Blavatnik School of Computer Science, Tel Aviv University, Israel**
haimk@tau.ac.il
3 **Department of Computer Science, University of Copenhagen, Denmark**
mikkel2thorup@gmail.com
4 **Blavatnik School of Computer Science, Tel Aviv University, Israel**
orzamir@mail.tau.ac.il
5 **Blavatnik School of Computer Science, Tel Aviv University, Israel**
zwick@tau.ac.il

Abstract

Gabow and Tarjan showed that the *Bottleneck Path* (BP) problem, i.e., finding a path between a given source and a given target in a weighted directed graph whose largest edge weight is minimized, as well as the *Bottleneck spanning tree* (BST) problem, i.e., finding a directed spanning tree rooted at a given vertex whose largest edge weight is minimized, can both be solved deterministically in $O(m \log^* n)$ time, where m is the number of edges and n is the number of vertices in the graph. We present a slightly improved randomized algorithm for these problems with an expected running time of $O(m\beta(m,n))$, where $\beta(m,n) = \min\{k \geq 1 \mid \log^{(k)} n \leq \frac{m}{n}\} \leq \log^* n - \log^*(m/n) + 1$. This is the first improvement for these problems in over 25 years. In particular, if $m \geq n \log^{(k)} n$, for some constant k, the expected running time of the new algorithm is $O(m)$. Our algorithm, as that of Gabow and Tarjan, work in the *comparison model*. We also observe that in the word-RAM model, both problems can be solved deterministically in $O(m)$ time. Finally, we solve an open problem of Andersson et al., giving a deterministic $O(m)$-time comparison-based algorithm for solving deterministic 2-player turn-based zero-sum terminal payoff games, also known as *Deterministic Graphical Games* (DGG).

1998 ACM Subject Classification G.2.2 Graph Theory – Graph Algorithms

Keywords and phrases bottleneck paths, comparison model, deterministic graphical games

Digital Object Identifier 10.4230/LIPIcs.STACS.2016.27

1 Introduction

The *Bottleneck Path* (BP) problem, also known as the *min-max path* problem, is the problem of finding a path from a given source s to a given target t in a weighted directed graph $G = (V, E)$ in which the maximum edge weight is minimized. In the closely related *Bottleneck Spanning Tree* (BST) problem, also known as the *min-max arborescence* problem, we are asked to find a directed spanning tree of a given weighted directed graph $G = (V, E)$, rooted at a given root vertex s, such that the maximum edge weight in the tree is minimized.

Deterministic Graphical games (DGG) form a simple and interesting family of 2-player turn-based zero-sum games. A DGG is played by two players, players 0 and 1, on a directed

33rd Symposium on Theoretical Aspects of Computer Science (STACS 2016).
Editors: Nicolas Ollinger and Heribert Vollmer; Article No. 27; pp. 27:1–27:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


graph $G = (V, E)$ whose vertex set V is partitioned into $V = V_0 \uplus V_1 \uplus T$, [1] where V_i is the set of vertices controlled by player i, where $i = 0, 1$, and T is the set of *terminals*. A *payoff* function $p : T \to \mathbb{R}$ assigns a payoff to each terminal. A token is placed at a *start* vertex $s \in V_0 \cup V_1$. If the token is currently at a vertex $u \in V_i$, then player i chooses an edge $(u, v) \in E$ and moves the token to v. Each non-terminal vertex is assumed to have at least one outgoing edge. Terminals have no outgoing edges. If the token reaches a terminal t, the game ends and player 0 pays player 1 the payoff $p(t)$. Player 0, also known as min, wants to minimize this payoff, while player 1, also known as max, wants to maximize it. If the game never ends, no payment is made. Player 0 prefers an infinite play over a positive payoff, while player 1 prefers an infinite play over a negative payoff. A DGG, with start s, is solved by finding its min-max value and optimal strategies for the two players.

BP, BST and DGG share several similar features. First, they are both min-max optimization problems, though under different interpretations. Second, they can all be solved by trivial linear time algorithms if the edge weights, or the terminal payoffs, are given in *sorted* order. Third, they can all be solved using a *threshold method* that goes back to Edmonds and Fulkerson [9] when the edge weights, or the terminal payoffs, are not given in sorted order. In the case of the BP problem, for example, we find the *median* of the edge weights and partition the edges accordingly into *light* and *heavy* edges. We now check whether there is a directed path from s to t that uses only light edges. If so, all heavy edges can be discarded. If there is no such light path, we can set the weight of all light edges to $-\infty$. In the next iteration, we compute the median of all edges whose weight is not $-\infty$. This easily leads on $O(m \log n)$-time algorithms for the BP, BST and DGG problems. We also note that if the graph is undirected, we can *contract* the light edges. This leads to simple $O(m)$-time algorithms for the BP and BST problems in undirected graphs.

Gabow and Tarjan [16] used a more sophisticated version of the threshold method to obtain an $O(m \log^* n)$-time algorithm for the BP and BST problems. We present an improved randomized algorithm for these problems whose running time is $O(m\beta(m, n))$. As mentioned, for $m \geq n \log^{(k)} n$, the expected running time is $O(m)$, i.e., best possible. We also show that BP and BST are equivalent under randomized reductions. As $\log^* n$ and $\beta(m, n)$ are both extremely slowly growing functions, our improved bound has no practical importance. We believe, however, that understanding the exact complexity of fundamental problems such as BP and BST is an important endeavor.

Andersson et al. [1] used the technique of Gabow and Tarjan [16] to obtain an $O(m\beta(m, k))$-time algorithm for solving DGGs, where k is the number of terminals. We show, perhaps surprisingly, that the DGG problem is *easier* than the BP and BST problems. By combining the viewpoints of the two players, we obtain a simple deterministic $O(m)$-time algorithm for solving DGGs.

1.1 Bottleneck Paths and Bottleneck Spanning Trees

Both the BP and BST in *directed* graphs are well-motivated problems that were studied by many researchers. The BP problem, for example, models the problem of finding a route from one city to another minimizing the maximum distance travelled between two consecutive cities. The equivalent problem of finding a max-min path from s to t is the problem of finding a maximum *capacity* path in a flow network. Edmonds and Karp [10] obtained an

[1] Here $A \uplus B$ stands for the *disjoint union* of A and B, i.e., the union $A \cup B$ where is it assumed that $A \cap B = \emptyset$.

efficient, though not strongly polynomial, maximum flow algorithm that repeatedly computes maximum capacity augmenting paths.

Edmonds and Fulkerson [9] introduced the *threshold method* that yields a simple $O(m \log n)$-time algorithm for the BP problem. They also note that the min-max s-t path problem has a max-min *dual*, i.e., the problem of finding a s-t *cut* whose minimal edge weight is maximized.

Dijkstra's [7] single-source shortest paths algorithm can be easily adapted to solve the BP and BST problems. The resulting algorithm solves, in fact, the *Single-Source Bottleneck Paths* (SS-BP) problem in which a min-max path is sought from the source s to any vertex of the graph. It is easy to see that the tree of min-max paths returned by the algorithm is also a min-max spanning tree. If Fibonacci heaps [13] are used, an $O(m + n \log n)$-time algorithm is obtained.

Gabow and Tarjan [16] obtained an improved algorithm for the BP and BST problems that runs in $O(m \log^* n)$ time, where $\log^* n = \min\{k \geq 1 \mid \log^{(k)} n \leq 1\}$, where $\log^{(1)} n = \log n$ and $\log^{(k)} n = \log \log^{(k-1)} n$, for $k > 1$. We improve on this 25 year old result and obtain a randomized algorithm whose running time is $O(m\beta(m,n))$, where $\beta(m,n) = \min\{k \geq 1 \mid \log^{(k)} n \leq \frac{m}{n}\} \leq \log^* n - \log^*(m/n) + 1$. In particular, if $m \geq n \log^{(k)} n$, for any constant k, the expected running time of the new algorithm is $O(m)$. Our algorithm, as that of Gabow and Tarjan, works in the *comparison model*, i.e., the only operations it performs on edge weights are pairwise comparisons. We also observe that in the word-RAM model, where comparisons are not the only operations allowed on edge weights, both problems can be solved deterministically in $O(m)$ time.

The BP problem can be easily reduced to the BST problem. We give the first randomized reduction in the other direction, showing that the BP and BST problems are essentially equivalent.

Punnen [23] shows that for a wide class of bottleneck problems, if the problem can be solved in $O(f(m))$ time when the weights are given in sorted order, then the problem can be solved in $O(f(m) \log^* m)$-time, when the weights are not given in sorted order.

The BP, BST and SS-BP can also be solved easily in $O(m)$ time if the input graph is *acyclic*.

All the results stated above are for directed graphs. For *undirected* graphs, both the BP and BST are much easier. Camerini [5] gave a simple $O(m)$-time algorithm for BP and BST problems in undirected graphs. Furthermore, if T is a *Minimum Spanning Tree* (MST) of an undirected graph $G = (V, E)$, i.e., a spanning tree such that the *sum* of its edge weights is minimal, then for any $s, t \in V$, the unique path in T between s and t is a min-max path between s and t. (See, e.g., Hu [19].) An MST of an undirected graph can be found in $O(m)$ expected time (Karger, Klein and Tarjan [21]), or deterministically in $O(m\alpha(m,n))$ time (Chazelle [6]). [2]

The *All-Pairs Bottleneck Paths* (AP-BP) problem, in which we want to find a bottleneck path for every pair of vertices in a weighted directed graph can be easily solved in $O(mn)$ time by first sorting all edge weights and then running a linear time SS-BP algorithm from each vertex. In dense enough graphs, faster algorithms can be obtained using fast matrix multiplication. Vassilevska, Williams and Yuster [25] showed that the AP-BP problem can be reduced to the problem of computing $(\max, \min)$ products and gave an algorithm whose running time is $O(n^{2+\omega/3})$, which is $O(n^{2.80})$, for computing such products. Here $\omega < 2.38$

[2] Chazelle's algorithm improves on $O(m\beta(m,n))$- and $O(m \log \beta(m,n))$-time algorithms of Fredman and Tarjan [13] Gabow et al. [15]. Is this an indication that similar improvements are also possible for the BP and BST problems?

is the matrix multiplication exponent. Duan and Pettie [8] obtained a faster algorithm for computing $(\max, \min)$ products whose running time is $O(n^{(3+\omega)/2})$, which is $O(n^{2.69})$, matching Matoušek's [22] fastest known algorithm for computing *dominance* products. For vertex weighted graphs, a faster running time of $O(n^{2.58})$ was previously obtained by Shapira, Yuster and Zwick [24].

1.2 Deterministic Graphical Games

Many turn-based 2-player zero-sum games can be modeled using finite *game trees*. The game starts at the root. At even levels, the first player chooses an edge to one of the children of the current vertex, at odd levels the second player chooses the edge. Each leaf has a *payoff* associated with it, which is the amount the first player has to pay the second player. The *value* of such a game can be easily determined by starting at the leaves and alternatingly computing the minimum or the maximum of the values of the nodes at the lower level.

It is natural to generalize game trees into *game graphs*, yielding exactly the *Deterministic Graphical Games* (DGGs) defined above. A game can now return to a position visited before, as may happen, for example, in *chess*. The main difference between game trees and game graphs is that *infinite* plays are now possible. An infinite play is considered to be a *draw*, i.e., no payment, or equivalently a payment of 0, is made. Such game graphs are implicit in Zermelo's [27] classical, but slightly incomplete, proof that each position in chess has a definite value. For more technical and historical details, see Washburn [26] and Andersson et al. [1].

A *strategy* for a player in a DGG is a rule for selecting the next edge to play in each situation. A general strategy may depend on the full *history* of the play and may be *randomized*. It can be shown, however, that in DGGs, both players may restrict themselves without loss to *pure positional strategies*, i.e., deterministic strategies that depend only on the current position. Each vertex v in a DGG has a *value* $val(v)$. Player 0 has a (pure positional) strategy that guarantees that the outcome of the game will be at most $val(v)$, no matter what strategy is used by player 1. Similarly, player 1 has a (pure positional) strategy that guarantees that the outcome of the game will be at least $val(v)$, no matter what strategy is used by player 0. Such strategies are said to be *optimal* from v. Both players actually have pure positional strategies that are optimal from all vertices.

Let $G = (V, E)$ be a DGG and let t be the terminal with the largest payoff. The set of vertices from which player 1 can force the game to end in t can be easily found in linear time using an alternating backward search from t, also known as *retrograde analysis*. (See details in [1] or in Section 6.) Thus, if the payoffs are given in sorted order, it is easy to find the values of all vertices, and optimal strategies for both players, by "peeling" the terminals one by one, in decreasing order.

Andersson et al. [1] used the technique of Gabow and Tarjan [16] to obtain an $O(m\beta(m, k))$-time algorithm for finding the value and optimal strategies for a specific start vertex s in a DGG with m edges and k terminals. We obtain a simple deterministic $O(m)$-time algorithm for the same problem.

The best known algorithm for finding the values, and corresponding optimal strategies, of *all* vertices of a DGG in the comparison model runs in $O(m + k \log k)$ time. The algorithm begins by sorting the payoffs in $O(k \log k)$ time, and then finds all values in $O(m)$ additional time.

When the payoffs are moved from terminals to edges or non-terminal vertices, and the sequence of resulting payoffs is accumulated in some way, we obtain *Mean Payoff Games* (MPGs) and *Discounted Payoff Games* (DPGs) [11, 17, 28, 2, 3] or *Parity Games* (PGs)

[12, 20]. These games are much harder than DGGs. No polynomial time algorithms are known for their solution.

1.3 Organization of the Paper

In the next section we review the classical $O(m \log^* n)$-time algorithm of Gabow and Tarjan [16] for the Bottleneck Path (BP) and Bottleneck Spanning Tree (BST) problems. In Section 3 we present our improved algorithm. In Section 4 we prove the equivalence of the BP and BST problems. In Section 5 we observe that both BP and BST can be solved deterministically in $O(m)$ time in the word-RAM model. In Section 6 we present a deterministic $O(m)$-time algorithm, in the comparison model, for solving Deterministic Graphical Games (DGGs). This result is independent of the results of the previous sections. The main results of the paper are in Sections 3 and 6. We conclude in Section 7 with some open problems.

2 The $O(m \log^* n)$-time Algorithm of Gabow and Tarjan

In this section we sketch the $O(m \log^* n)$-time algorithm of Gabow and Tarjan [16] for the BST problem in weighted directed graphs. The algorithm can be easily modified to solve the BP problem. We also note that the algorithm performs only $O(m)$ comparisons.

Gabow and Tarjan [16] first observe that if the edge weights are small integers, i.e., $w : E \to \{0, 1, \ldots, k\}$, then the BST problem can be easily solved in $O(m + k)$ time. We first use *bucket sort* to sort the outgoing edges of each vertex in non-decreasing order and then use an *incremental search*. The search starts at the source vertex s and finds all vertices reachable from s using edges of weight 0. If all vertices are reached, we are of course done. Otherwise, we resume the search from all vertices reached allowing now edges of weights 0 and 1, and so on. It is not difficult to check that this can be implemented in $O(m + k)$ time.

Assume now that the edge weights are real numbers. If the edge weights are given to us in sorted order, we could easily replace them by integer weights from $\{1, 2, \ldots, m\}$, depending on their *rank*, and use the algorithm above to solve the problem in $O(m)$ time. However, sorting the edge weights in the comparison model requires $\Omega(m \log n)$ comparisons and time. The challenge is solving the problem *without* sorting all the edge weights.

Let $G = (V, E)$ be an instance of the BST problem were $E = E_0 \cup F$ such that all edges of $E_0 \subseteq E$ are known to have weights below the bottleneck weight and such that the weight of each edge of E_0 is smaller than the weight of each edge of F. (Possibly $E_0 = \emptyset$.) For simplicity, assume that all edges of F have distinct weights. By repeatedly finding medians [4], using $O(|F| \log k)$ time and comparisons, we can partition F into k subsets $E_1, \ldots, E_k$ of almost equal size such that the weight of all edges in E_j are smaller than the weights of all edges in E_{j+1}, for $j = 0, 1, \ldots, k-1$. We can now replace the edge weights of all edges in E_j by j, for $j = 0, 1, \ldots, k$, and run the linear time algorithm above. If the answer we get is i, then the bottleneck edge belongs to E_i. Thus, all the edges of $E_{i+1} \cup \cdots \cup E_k$ are not needed, and all edges of $E_0 \cup \cdots \cup E_{i-1}$ are now known to have weights which are below the bottleneck weight. We are thus left with a smaller instance $G = (V, E')$ where $E' = E'_0 \cup F'$, $E'_0 = E_0 \cup \cdots \cup E_{i-1}$ and $F' = E_i$. Note that $|F'| \leq |F|/k + 1$.

We can now iterate the above step. Let $F^{(j)}$ be the edge set known to contain the bottleneck edge after j iterations and let $m_j = |F^{(j)}|$. Initially $F^{(0)} = E$ and $m_0 = m$. When $m_j = 1$, we are done. Let k_j be the number of sets to which $F^{(j)}$ is partitioned. Thus $m_{j+1} \leq m_j/k_j$, and consequently $m_j \leq m/(k_0 k_1 \ldots k_{j-1})$. The number of comparisons used in the j-th iteration is therefore $O(m_j \log k_j) = O((m/(k_{j-2}k_{j-1})) \log k_j)$. (We let $k_{-2} = k_{-1} = 1$.) The time used in the j-th iteration is $O(m)$. We choose $k_0 = 2$ and

$k_j = 2^{k_{j-1}}$, for $j > 0$. After at most $\log^* n$ iterations we get $m_j = 1$. The total time spent is $O(m \log^* n)$. The number of comparisons used in the j-th iteration is $O(m_j \log k_j) = O((m/(k_{j-2}k_{j-1})) \log k_j) = O(m/k_{j-2})$ and the total number of comparisons performed is thus $O(m)$.

3 An $O(m\beta(m,n))$-time Algorithm for Bottleneck Paths and Trees

Let $G = (V, E)$ be the input graph, $w : E \to \mathbb{R}$ be a weight function defined on its edges, and let $s \in V$ be a source vertex. Let $m = |E|$ and $n = |V|$. For simplicity, we assume that all edge weights are distinct. In the previous section we saw that in $O(m \log k)$ time we can partition E into $E = E_1 \uplus E_2 \uplus \cdots \uplus E_k$ such that $E_1, E_2, \ldots, E_k$ have roughly the same size and such that all edges of E_j have weight smaller than all edges of E_{j+1}, for $j = 1, 2, \ldots, k-1$. In $O(m)$ time, we can then find the set E_i that contains the bottleneck edge.

To obtain the improved algorithm, we adopt a slightly different approach. Let $\lambda_1 < \lambda_2 < \cdots < \lambda_k$ be k *thresholds* and let $\lambda_0 = -\infty$ and $\lambda_{k+1} = \infty$. The thresholds naturally partition E into $E = E_0 \uplus E_1 \uplus \cdots \uplus E_k$ such that $E_i = \{e \in E \mid \lambda_i \leq w(e) < \lambda_{i+1}\}$. Explicitly computing this partition requires $\Omega(m \log k)$ time. We show, however, that we can compute the index i of the set E_i that contains the bottleneck edge in $O(m + nk)$ time, or even $O(m + n \log k)$ time, using a simple deterministic algorithm that does not explicitly compute the partition.

To obtain our improved algorithm, we set the k thresholds to the weights of k *randomly* chosen edges of the graph. We then compute the set E_i that contains the bottleneck edge and revert to the standard algorithm. The exact details will follow after describing the simple algorithm for locating the part that contains the bottleneck edge.

3.1 Locating the Bottleneck Weight Among k Thresholds

Let $G = (V, E)$ be a weighted directed graph, $w : E \to \mathbb{R}$ a weight function defined on its edges, $s \in V$ a source vertex, and let $-\infty = \lambda_0 < \lambda_1 < \cdots < \lambda_k < \lambda_{k+1} = \infty$ be arbitrary thresholds. Let $w^*(G)$ be the bottleneck edge weight of G. We begin by describing a simple $O(m + nk)$ time algorithm, called Locate, for computing the index i such that $\lambda_i \leq w^*(G) < \lambda_{i+1}$. For concreteness, we consider the BST problem. The details for the BP problem are almost identical.

The algorithm is composed of $k + 1$ phases. In the i-th phase, where $i = 0, 1, \ldots, k$, the algorithm finds all vertices $u \in V$ for which there is a directed path from s to u in G all whose edges have weights that are strictly smaller than λ_{i+1}. For every vertex u we maintain a value $d[u]$ such that a path from s to u all whose edge weights are at most $d[u]$ was already discovered. Initially $d[s] = -\infty$ while $d[u] = \infty$ for every $u \in V \setminus \{s\}$. We maintain the invariant that at the beginning of the i-th phase, for $i = 0, 1, \ldots, k$, we have $d[u] < \lambda_i$ for every $u \in V$ for which there is a path from s to u all whose edge weights are smaller than λ_i. In the i-th phase itself, we identify all vertices u with $d[u] < \lambda_{i+1}$ and examine all their outgoing edges. If $(u, v) \in E$, we let $\bar{w}(u, v) = \max\{d[u], w(u, v)\}$. If $\bar{w}(u, v) < d[v]$, we let $d[v] \leftarrow \bar{w}(u, v)$. If at the end of the i-th phase $d[u] < \lambda_{i+1}$ for all $u \in V$, we know that $\lambda_i \leq w^*(G) < \lambda_{i+1}$. The complexity of the algorithm is $O(m + nk)$ as we examine each edge at most once and each vertex at most k times. A more precise description follows.

In addition to the phase number i and the values $d[u]$, for every $u \in V$, the algorithm maintains two sets of vertices $A, B \subseteq V$. The set A contains all vertices $u \in V$ with $d[u] < \lambda_{i+1}$ whose outgoing edges were not examined yet. The set B contains all vertices

Algorithm Locate$(G = (V, E), w, s, (\lambda_1, \ldots, \lambda_k))$

```
λ_0 ← −∞ ; λ_{k+1} ← ∞
foreach v ∈ V do d[v] ← ∞
d[s] = −∞ ; A ← ∅ ; B ← V
for i ← 0 to k do
    foreach u ∈ B do
        if d[u] < λ_{i+1} then Move(v, B, A)
    while A ≠ ∅ do
        u ← Extract(A)
        foreach (u, v) ∈ E do
            w̄ ← max{d[u], w(u, v)}
            if w̄ < d[v] then
                d[v] ← w̄
                if d[v] < λ_{i+1} and v ∈ B then
                    Move(v, B, A)
    if B = ∅ then return i
```

Figure 1 Locating the bottleneck weight among k thresholds.

$u \in V$ for which $d[u] \geq \lambda_{i+1}$. Initially $i = -1$, $A = \emptyset$ and $B = V$. At the beginning of the i-th phase, for $i = 0, 1, \ldots, k$, the algorithm examines all vertices of B and moves to A each vertex u for which $d[u] < \lambda_{i+1}$. As long as A is not empty, the algorithm removes an arbitrary vertex u from A and scans all its outgoing edges as above. For every outgoing edge $(u, v) \in E$ it lets $\bar{w}(u, v) = \max\{d[u], w(u, v)\}$. If $\bar{w}(u, v) < d[v]$, it lets $d[v] \leftarrow \bar{w}(u, v)$. If $d[v] < \lambda_{i+1}$ and $v \in B$, then v is moved from B to A. The i-phase ends when A is empty. If B is also empty, the algorithm returns i and terminates. Otherwise, it moves on to the $(i + 1)$-st phase.

Pseudo-code of Locate is given in Figure 1 . Function Move(v, B, A) moves v from B to A while Extract(A) removes and returns an arbitrary item of A. With a simple linked-list implementation, both these operations take constant time.

▶ **Theorem 1.** *Algorithm* Locate *returns an index i such that $\lambda_i \leq w^*(G) < \lambda_{i+1}$. Its running time is $O(m + nk)$.*

Proof. The correctness of the algorithm follows from the invariant stated above: At the end of the i-phase, for every $u \in V$, if there is a path in G from s to u all whose edges have weights strictly smaller than λ_{i+1}, then $d[u] < \lambda_{i+1}$. The invariant holds vacuously for $i = -1$.

Suppose that the invariant holds at the end of the $(i - 1)$-st phase. Suppose, for the sake of contradiction, that the invariant does not hold at the end of the i-th phase. Namely, suppose that there is a path $s = u_0 \to u_1 \to \cdots \to u_k$ in G such that $(u_j, u_{j+1}) \in E$ and $w(u_j, u_{j+1}) < \lambda_{i+1}$, for $j = 0, 1, \ldots, k - 1$, but $d[u_k] \geq \lambda_{i+1}$. Let u_ℓ be the first vertex on the path for which $d[u_\ell] \geq \lambda_{i+1}$. As $d[u_0] = -\infty < \lambda_{i+1}$, we have $\ell \geq 1$. By definition $d[u_{\ell-1}] < \lambda_{i+1}$. As $d[u_{\ell-1}] < \lambda_{i+1}$, $u_{\ell-1}$ must have been moved to A either in the i-th phase, or before. After the edge $(u_{\ell-1}, u_\ell)$ is examined, at or before the i-th phase, we have $d[u_\ell] \leq \bar{w}(u_{\ell-1}, u_\ell) = \max\{d[u_{\ell-1}], w(u, v)\} < \lambda_{i+1}$, a contradiction.

Thus, if $\lambda_i \leq w^*(G) < \lambda_{i+1}$, then at the end of the i-th phase B is empty and the

algorithm terminates. The algorithm cannot terminate before the i-th phase as there is at least one vertex u for which there is no path all whose edges have weights strictly smaller than λ_i. For such a vertex we must have $d[u] \geq \lambda_i$. Thus, u must be in B until the beginning of phase i.

The running time of the algorithm is $O(m + nk)$ as each edge is examined at most once, and each vertex is examined at most once at each one of the k iterations. ◀

Although the $O(m + nk)$ running time of LOCATE is sufficient for our purposes, the running time can be reduced to $O(m + n \log k)$ using a *lazy binary search*. Details will appear in the full version of the paper.

3.2 A Randomized $O(m\beta(m, n))$-time Algorithm

Let $r \geq 1$. Choose $k = \log^{(r)} n$ *random* edges $e_1, e_2, \ldots, e_k$ from E and sort their edge weights so that $w(e_1) < w(e_2) < \cdots < w(e_k)$. Let $\lambda_i = w(e_i)$, for $i = 1, 2, \ldots, k$, and $\lambda_0 = -\infty$, $\lambda_{k+1} = \infty$. (Sorting the edge weights takes $O(k \log k)$ time, which will be negligible.) We now use LOCATE to find the index i for which $\lambda_i \leq w^*(G) < \lambda_{i+1}$. This takes only $O(m+n \log^{(r)} n)$ time. In $O(m)$ further time, we can compute the sets $E_0 = \{e \in E \mid w(e) < \lambda_i\}$ and $F = \{e \in E \mid \lambda_i \leq w(e) < \lambda_{i+1}\}$. The next lemma shows that the expected size of $F = E_i$ is $O(m/k)$.

▶ **Lemma 2.** *Let $f \in E$ be a fixed edge, and let $e_1, e_2, \ldots, e_k$ be k random edges from E such that $w(e_1) < w(e_2) < \cdots < w(e_k)$. Let $\lambda_i = w(e_i)$, for $i = 1, 2, \ldots, k$, and let $\lambda_0 = -\infty$ and $\lambda_{k+1} = \infty$. Let $E_i = \{e \in E \mid \lambda_i \leq w(e) < \lambda_{i+1}\}$, for $i = 0, 1, \ldots, k$, and let j be such that $f \in E_j$. Then, $\mathbb{E}[|E_j|] \leq 2m/k$, where $m = |E|$.*

Proof. Let $f_1, f_2, \ldots, f_m$ be the edges of E sorted according to weight, i.e., $w(f_1) < w(f_2) < \cdots < w(f_m)$. Let $f = f_r$, where $1 \leq r \leq m$. The probability of a given edge f_i to be one of the k randomly chosen edges, given that no edge from a set F' is in the sample, is $k/(m - |F'|) \geq k/m$. Examine the edges $f_r, f_{r+1}, \ldots, f_m$ one by one until finding an edge from the sample, or until reaching the last edge. As the probability of each inspected edge to be in the sample is at least k/m, the expected number of edges inspected is at most m/k. Similarly, the expected number of edges among $f_{r-1}, f_{r-2}, \ldots, f_1$ that need to be inspected until finding an edge from the sample, or until reaching the first edge, is also at most m/k. ◀

It is not difficult to extend the proof of Lemma 2 to show that the size of F is $O(m/k)$ with high probability. For our purposes it is enough to rely on Markov's inequality to infer that the probability that $|F| \geq 4m/k$ is at most $1/2$. If $|F| \geq 4m/k$, we simply choose a new sample. The expected number of samples needed is at most 2.

After running LOCATE with $k = \log^{(r)} n$ *random* edges, we get that $|F| \leq 4m/\log^{(r)} n$. We now run one iteration of the algorithm of Gabow and Tarjan from the previous section with $k = \log^{(r-1)} n$. The running time is $O(m + |F| \log k) = O(m)$ and the size of F is reduced to $O(m/\log^{(r-1)} n)$. In at most $r - 1$ additional iterations, we can thus reduce the size of F to $O(m/\log n)$, at which point we can afford to sort F and find the bottleneck edge in $O(m)$ time. The total running time of the algorithm is therefore $O(rm + n \log^{(r)} n)$. If we choose $r = \beta(m, n)$, we get an expected running time of $O(m\beta(m, n))$. As mentioned, it is easy to see that a running time of $O(m\beta(m, n))$ is obtained not only in expectation, but also in very high probability.

▶ **Theorem 3.** *The Bottleneck Path (BP) and Bottleneck Spanning Tree (BST) problems can be solved in the comparison model in $O(m\beta(m, n))$ expected time.*

4 Equivalence of Bottleneck Paths and Bottleneck Spanning Trees

In this section we show that if there is an $O(f(m,n))$ time algorithm for the BST problem, then there is also an $O(f(m,n))$ time algorithm for the BP problem, and vice versa. The reduction from BP to BST is immediate. The reduction from BST to BP is slightly more complicated, requires randomization and needs some mild assumptions on $f(m,n)$, which are satisfied if $f(m,n) = m+n$.

▶ **Lemma 4.** *If there is an $f(m,n)$-time algorithm for the Bottleneck Spanning Tree (BST) problem, then there is an $O(f(m+n,n))$-time algorithm for the Bottleneck Path (BP) problem.*

Proof. Given an instance $G=(V,E)$ of the BP problem, with source s and target t, simply add edges (t,v), for every $v \in V$, of weight $-\infty$. The path from s to t in a bottleneck spanning tree of the resulting graph is a bottleneck path from s to t. ◀

Before describing the reduction in the opposite direction, we need to introduce some more notation. If $G=(V,E)$ is a weighted graph with source s, we let $w^*(v)$ be the weight of the bottleneck edge on a min-max path from s to v in G. We then define $E(v) = \{e \in E \mid w(e) \leq w^*(v)\}$ and let $S(v)$ be the set of vertices reachable from s in $(V,E(v))$. Finally, we let $E_{in}(S(v))$ be the set of edges that enter vertices of $S(v)$. We now have the following simple probabilistic lemma.

▶ **Lemma 5.** *Let $G=(V,E)$ be a weighted graph with source s.*
(i) *If v is a randomly chosen vertex, then $\mathbb{P}[|S(v)| \geq \frac{n}{2}] \geq \frac{1}{2}$.*
(ii) *If (u,v) is a randomly chosen edge, then $\mathbb{P}[|E_{in}(S(v))| \geq \frac{m}{2}] \geq \frac{1}{2}$.*

Proof. (i) Let $v_1, v_2, \ldots, v_n$ be an ordering of the vertices such that $w^*(v_1) \leq w^*(v_2) \leq \cdots \leq w^*(v_n)$. Note that $S(v_i) \supseteq \{v_1, v_2, \ldots, v_i\}$. Thus, if v is among $\{v_{\lceil n/2 \rceil}, \ldots, v_n\}$, then $|S(v)| \geq n/2$, and this happens with a probability of at least $1/2$.

(ii) Let d_i be the in-degree of v_i. Let k be the minimal index for which $\sum_{i=1}^{k} d_i \geq \frac{m}{2}$. (In particular, we have $\sum_{i=1}^{k-1} d_i < \frac{m}{2}$. Let (u,v) be a random edge. If $v = v_i$, and $i \geq k$, then $E_{in}(S(v)) \geq m/2$. This happens with a probability of at least $\frac{1}{m}\sum_{i=k}^{n} d_i = \frac{1}{m}(m - \sum_{i=1}^{k-1} d_i) \geq \frac{1}{2}$. ◀

▶ **Lemma 6.** *If there is an $f(m,n)$-time algorithm for the Bottleneck Path (BP) problem, then there is a randomized algorithm whose expected running time is $O(\sum_{i\geq 0} f(\frac{m}{2^i}, \frac{n}{2^i}))$ for the Bottleneck Spanning Tree (BST) problem.*

Proof. Let $G=(V,E)$ be an instance of the BST problem with source s. Choose a random vertex $v \in V$ and solve the BP problem with $t = v$. The bottleneck edge weight $w^*(v)$ returned is clearly a lower bound on the bottleneck edge weight in a spanning tree. If $S(v) = V$, we are done. Otherwise, all vertices of $S(v)$ may be replaced by a new source $\bar{s}$, and all edges of $E_{in}(S(v))$, which now enter $\bar{s}$, may be removed. By Lemma 5(i), $|S(v)| \geq \frac{n}{2}$ with a probability of at least $\frac{1}{2}$. If this is not the case, we can repeat this step. (This is not really required, but it slightly simplifies the analysis.) The expected number of repetitions is constant. We are now left with an instance with at most $\frac{n}{2}$ vertices. To reduce the number of edges we now sample a random edge $(u,v) \in E$ and solve the BP problem with $t = v$. If $S(v) = V$, we are again done. Otherwise, we can again replace $S(v)$ by a new source $\bar{s}$ and remove all the edges of $E_{in}(S(v))$. By Lemma 5(ii), $|E_{in}(S(v))| \geq \frac{m}{2}$ with a probability of at least $\frac{1}{2}$. If this is not the case, we can repeat this step. We continue in this way, alternatingly sampling vertices and edges. The total expected running time is then $O\big(\sum_{i\geq 0}\big(f(\frac{m}{2^i}, \frac{n}{2^i})) + f(\frac{m}{2^i}, \frac{n}{2^{i+1}})\big)\big) = O\big(\sum_{i\geq 0} f(\frac{m}{2^i}, \frac{n}{2^i})\big)$. ◀

If we are only interested in a time bound in terms of m, we can do only edge sampling steps. The running time is then $O\big(\sum_{i\geq 0} f(\frac{m}{2^i})\big)$. If $\frac{f(m)}{m}$ is monotone non-decreasing, the resulting running time is $O(f(m))$.

5 Bottleneck Paths and Trees in the Word-RAM Model

On the word-RAM with word length $w \geq \log n$, we can use a constant number of levels of *fusion nodes* (Fredman and Willard [14]) to split the m edge weights into $k = \log n$ sets $E_1, E_2, \ldots, E_k$ of size $O(m/\log n)$ such that the weights of all edges in E_i are smaller than the weights of all edges in E_{i+1}, for $i = 1, \ldots, k-1$. This requires only $O(m)$ time. Using $O(m)$ further time we can use the simple algorithm of Section 2 to find the subset containing the bottleneck weight. We can afford to completely sort this subset using a standard comparison-base algorithm, as this takes only $O((m/\log n)\log n) = O(m)$ time. As the relevant edge weights are now sorted, we can use the algorithm of Section 2 again to completely solve the problem, using only $O(m)$ additional time.

Using a word-RAM algorithm of Han and Thorup [18], we can actually split the edge weights into $\sqrt{m}$ sets of size $O(\sqrt{m})$, again in $O(m)$ time.

6 An $O(m)$-time Algorithm for Deterministic Graphical Games

A *Deterministic Graphical Game* (DGG) is composed of directed graph $G = (V, E)$, a partition $V = V_0 \uplus V_1 \uplus T$, an initial vertex $s \in V_0 \cup V_1$ and a payoff function $p : T \to \mathbb{R}$. For the exact definition refer to the Introduction and Section 1.2. We begin with the following folklore lemma which is also used in Andersson et al. [1].

▶ **Lemma 7.** *Let $G = (V, E)$ be a DGG with a unique target t of payoff 1. Let W_1 be the set of vertices of value 1, $E_1 = \{(u, v) \in E \mid v \in W_1\}$ and $m_1 = |E_1|$. Then, there is a deterministic algorithm with running time $O(m_1)$ for computing W_1 and for constructing a strategy for player 1 that ensures value 1 from all the vertices of W_1. The set of vertices $W_0 = V \setminus W_1$ of value 0 and an optimal strategy for player 0 from all vertices can be found, if required, in $O(n)$ additional time.*

Proof. We use a backward search from t to find all the vertices in G whose value is 1. The value of all the remaining vertices is 0. Let $W_1 \leftarrow \{t\}$ and $A \leftarrow \{t\}$. While A is not empty, extract a vertex $v \in A$. For every incoming edge $(u, v) \in E$, do the following. If $u \in V_1$, or (u, v) is the last remaining outgoing edge of u, then add u to W_1 and A and set $\pi(u) \leftarrow v$. Otherwise, simply remove (u, v) from the graph. We refer to handling such an incoming edge (u, v) as a *basic step*. When the algorithm terminates, W_1 is the set of all vertices of value 1. The running time of the algorithm is $O(m_1)$ as each incoming edge of a vertex of W_1 is examined exactly once. The strategy that from each vertex $u \in V_1 \cap W_1$ chooses the edge $(u, \pi(u))$ is an optimal strategy for player 1. (The choice at vertices of $V_1 \setminus W_1$ may be arbitrary.) The set $W_0 = V \setminus W_1$ can be computed in $O(n)$ time. An optimal strategy for player 0 is obtained by choosing for each vertex $u \in V_0 \cap W_0$ the first remaining outgoing edge of u. (There must be at least one such edge and it must lead to a vertex of W_0.) Constructing such a strategy also requires only $O(n)$ additional time. ◀

If the terminals $t_1, t_2, \ldots, t_k$ are given in sorted order, i.e., $p(t_1) < p(t_2) < \ldots p(t_k)$, we can apply the algorithm above repeatedly to find the values of all vertices. To find the set of vertices W_k of value $p(t_k)$ we add self-loops to terminals $t_1, t_2, \ldots, t_{k-1}$ and move them from T to either V_0 or V_1, so that t_k is the only remaining terminal, and run the algorithm of

Lemma 7. We then remove the vertices of W_k and all their incoming edges, find all vertices whose value is $p(t_{k-1})$, and so on. The total running time is $O(m)$, as we do not examine again edges that were removed from the graph. If some of the payoffs are negative, we stop when we reach the last positive payoff and then start in a symmetric manner from the smallest negative payoff. The remaining vertices are the vertices of value 0.

If the payoffs of $t_1, t_2, \ldots, t_k$ are not given to us in sorted order, we can sort them in $O(k \log k)$ time and then run the linear time algorithm above. The running time is then $O(m + k \log k)$. This is the fastest known algorithm for finding the values of *all* vertices.

Andersson et al. [1] gave an $O(m\beta(m,k))$-time algorithm for finding the value of a specific start vertex s. Their algorithm is similar to the algorithm of Gabow and Tarjan [16] for the BP and BST problems sketched in Section 2. We obtain an improved deterministic $O(m)$-time algorithm. The key ingredient in our $O(m)$-time algorithm is the following simple lemma.

▶ **Lemma 8.** *Let $G = (V, E)$ be a DGG such that $V = V_0 \cup V_1 \cup T$ where $T = \{t_1, t_2\}$ and $0 < p(t_1) < p(t_2)$. Let W_i be the vertices of G whose value is $p(t_i)$, let $E_i = \{(u, v) \in E \mid v \in W_i\}$, and $m_i = |E_i|$, for $i = 1, 2$. Assume that $W_1 \cup W_2 = V$, i.e., no vertex has value 0. Then, there is a deterministic algorithm for computing either W_1 or W_2 in $O(\min\{m_1, m_2\})$ time.*

Proof. We run in *parallel* two instances of the algorithm of Lemma 7, one on a game obtained by adding a self-loop to t_1, which is no longer a terminal, and one on a game obtained by adding a self-loop to t_2 and replacing the roles of the two players. The first instance is trying to construct W_2 while the second is trying to construct W_1. We alternatingly perform basic steps in these two instances. When one of these instances finishes, we stop the other. The running time of the resulting algorithm is clearly $O(\min\{m_1, m_2\})$. ◀

Using Lemma 8 we obtain the main result of this section.

▶ **Theorem 9.** *There is a deterministic $O(m)$-time algorithm for finding the value and optimal strategies for both players in a Deterministic Graphical Game (DGG) with a given start vertex.*

Proof. Let $G = (V, E)$ be a DGG, where $V = V_0 \cup V_1 \cup T$, $s \in V_0 \cup V_1$ is the start vertex, and $p : T \to \mathbb{R}$ is the payoff function. We begin by describing an algorithm for finding the value of s.

We first perform a preprocessing step that determines for each vertex $u \in V$ whether its value $val(u)$ is positive, zero, or negative. To find all vertices of positive value, we merge all terminals of positive payoff into a single terminal, give this terminal a payoff of 1, and run the algorithm of Lemma 7. Similarly, we can find all vertices with negative values. The remaining vertices have value 0. If $val(s) = 0$, we are done. If $val(s) > 0$, we can remove from the game all vertices with non-positive value and all edges entering them. Similarly, if $val(s) < 0$, we can remove from the game all vertices with non-negative value and all edges entering them. For concreteness, we assume that $val(s) > 0$. The case $val(s) < 0$ is analogous.

Assume therefore that $G = (V, E)$ is a DGG for which $val(u) > 0$, for every $u \in V$, with $|T| = k$. We assume, for simplicity, that all payoffs are distinct. This assumption can be easily removed. Find the *median* of the payoffs and split the terminal set T into two subsets T_1 and T_2 of sizes $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$ such that for every $t_1 \in T_1$ and $t_2 \in T_2$ we have $p(t_1) < p(t_2)$. Merge all the terminals in T_i into a new terminal t_i with payoff i, for $i = 1, 2$.

Let W_i be the set of vertices in the new game whose values are i, $E_i = \{(u, v) \in E \mid v \in W_i\}$, and $m_i = |E_i|$, for $i = 1, 2$.

We now run the algorithm of Lemma 8 and in $O(\min\{m_1, m_2\})$ time construct either W_1 or W_2. If the construction of W_1 is complete and $s \in W_1$, or the construction of W_2 is complete but $s \notin W_2$, we know that $val'(s) = 1$, otherwise $val'(s) = 2$, where $val'(s)$ is the value of s in the new game. If $val'(s) = 1$, we construct W_2 and E_2 in $O(m_2)$ time. (If the construction of W_2 was not complete, we let $W_2 \leftarrow V \setminus W_1$ and then compute E_2.) We can now remove all edges of E_2 and all terminals of T_2 from the original game G without changing $val(s)$. Similarly, if $val'(s) = 2$, we construct W_1 and E_1 in $O(m_1)$ time and remove all edges of E_1 and all terminals of T_1 from G. In both cases, in $O(m' + k')$ time we removed m' edges and k' terminals from the game.

We repeat the process until we are left with only one terminal whose payoff is then the value of s. As the running time of each iteration is proportional to the number of edges and terminals removed from the graph, the total running time of the algorithm is $O(m + k) = O(m)$.

Once $val(s)$ is known, it is easy to find optimal strategies for both players from s. To find an optimal strategy for player 1, we merge all terminals with payoffs at least $val(s)$ into a new terminal. To all terminals with payoffs less than $val(s)$ we add a self-loop, so that they are not terminals any longer. An optimal strategy for player 1 from s in this new game, which can be found in $O(m)$ time using the algorithm of Lemma 7, is also an optimal strategy for player 1 in the original game. An optimal strategy for player 0 from s can be found in a similar manner. ◀

7 Concluding Remarks and Open Problems

We presented an improved randomized algorithm for the Bottleneck Path (BP) and Bottleneck Spanning Tree (BST) problems with an expected running time of $O(m\beta(m, n))$ and a deterministic $O(m)$-time algorithm for solving a Deterministic Graphical Game (DGG) with a given start vertex. Many open questions remain. Is there an $O(m)$-time algorithm for the BP and BST problems? Is there a deterministic $O(m\beta(m, n))$-time algorithm for the BP and BST problems? Can the $O(m + n \log n)$-time algorithm for Single-Source Bottleneck Paths (SS-BP) problem be improved? Can the $O(m + k \log k)$-time algorithm for finding the values of *all* vertices of a DGG be improved?

References

1 D. Andersson, K.A. Hansen, P.B. Miltersen, and T.B. Sørensen. Deterministic graphical games revisited. *Journal of Logic and Computation*, 22(2):165–178, 2010.

2 H. Björklund, S. Sandberg, and S. Vorobyov. Memoryless determinacy of parity and mean payoff games: a simple proof. *Theoretical Computer Science*, 310(1-3):365–378, 2004.

3 H. Björklund and S. Vorobyov. Combinatorial structure and randomized subexponential algorithms for infinite games. *Theoretical Computer Science*, 349(3):347–360, 2005.

4 M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

5 P.M. Camerini. The min-max spanning tree problem and some extensions. *Information Processing Letters*, 7(1):10–14, 1978. doi:10.1016/0020-0190(78)90030-3.

6 B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

7 E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

8 R. Duan and S. Pettie. Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. In *Proc. of 20th SODA*, pages 384–391, 2009.

9 J. Edmonds and D.R. Fulkerson. Bottleneck extrema. *Journal of Combinatorial Theory*, 8(3):299–306, 1970.

10 J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

11 A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8:109–113, 1979.

12 E.A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. of 32nd FOCS*, pages 368–377, 1991.

13 M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

14 M.L. Fredman and D.E. Willard. Surpassing the information-theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

15 H.N. Gabow, Z. Galil, T.H. Spencer, and R.E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6:109–122, 1986.

16 H.N. Gabow and R.E. Tarjan. Algorithms for two bottleneck optimization problems. *Journal of Algorithms*, 9(3):411–417, 1988.

17 V.A. Gurvich, A.V. Karzanov, and L.G. Khachiyan. Cyclic games and an algorithm to find minimax cycle means in directed graphs. *USSR Computational Mathematics and Mathematical Physics*, 28:85–91, 1988.

18 Y. Han and M. Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *Proc. of 43rd FOCS*, pages 135–144, 2002.

19 T.C. Hu. The maximum capacity route problem. *Operations Research*, 9(6):898–900, 1961.

20 M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM Journal on Computing*, 38(4):1519–1532, 2008.

21 D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42:321–328, 1995.

22 J. Matoušek. Computing dominances in E^n. *Information Processing Letters*, 38(5):277–278, 1991. doi:10.1016/0020-0190(91)90071-O.

23 A.P. Punnen. A fast algorithm for a class of bottleneck problems. *Computing*, 56(4):397–401, 1996. doi:10.1007/BF02253463.

24 A. Shapira, R. Yuster, and U. Zwick. All-pairs bottleneck paths in vertex weighted graphs. *Algorithmica*, 59:621–633, 2011.

25 V. Vassilevska, R. Williams, and R. Yuster. All pairs bottleneck paths and max-min matrix products in truly subcubic time. *Theory of Computing*, 5(1):173–189, 2009.

26 A. Washburn. Deterministic graphical games. *Journal of Mathematical Analysis and Applications*, 153(1):84–96, 1990.

27 E. Zermelo. Über eine anwendung der mengenlehre auf die theorie des schachspiels. In *Proceedings of the Fifth International Congress of Mathematicians*, pages 501—504, 1913.

28 U. Zwick and M.S. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1–2):343–359, 1996.