Faster Algorithm for Mean-Payoff Games*

Jakub Chaloupka and Luboš Brim

Faculty of Informatics, Masaryk University,
Botanická 68a, 60200 Brno, Czech Republic
{xchalou1,brim}@fi.muni.cz

Abstract. We study some existing techniques for solving mean-payoff games (MPGs), improve them, and design a randomized algorithm for solving MPGs with currently the best expected complexity.

Key words: mean-payoff games, randomized algorithms, complexity

Introduction

A *Mean-Payoff Game (MPG)* [7, 8, 14] is a two-player infinite game on a finite weighted directed graph. The game is given by a graph $\mathcal{G} = (V, E, w)$ with integer edge-weights and a partition of the set of vertices V into the sets V_{Max} and V_{Min}. The two players, named Max and Min, move a token along the edges of $\mathcal{G}$ ad infinitum. If the token is on a vertex $v \in V_{\text{Max}}$, Max chooses an edge $(v, u) \in E$ and the token goes to u. If the token is on a vertex $v \in V_{\text{Min}}$, it is Min's turn to choose an outgoing edge. This way an infinite path is formed. Max wants to maximize the average edge weight of the path and Min wants to minimize it. It was proved [7] that each vertex $v \in V$ has a *value*, denoted by $\nu(v)$, which each player can secure by a positional strategy, i.e. strategy that always chooses the same outgoing edge in the same vertex. To solve a MPG is to find the values of all vertices and, optionally, also optimal strategies for both players, i.e. strategies that secure the values.

MPGs have many applications, especially in the synthesis, analysis and verification of reactive (non-terminating) systems. Many natural models of such systems include quantitative information, and the corresponding question requires the solution of quantitative games, like MPGs. Quantities may represent, for example, the power usage of an embedded component, or the buffer size of a networking element [4].

Examples of applications include various kinds of scheduling, finite-window online string matching, or more generally, analysis of online problems and algorithms, and selection with limited storage [14]. Moreover, μ-calculus model-checking is polynomial-time reducible to MPGs via parity games [9]. MPGs can even be used for solving the max-plus algebra $Ax = Bx$ problem, which in turn has further applications [6].

* This work has been partially supported by the Grant Agency of the Czech Republic grant No. 201/09/1389.

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2348

MPGs are also important from a theoretical point of view. The problem whether the value of a certain vertex is greater or less than a certain threshold is in the complexity class $\mathsf{NP} \cap \mathsf{co}\text{–}\mathsf{NP}$ and it is not known whether the problem is in P.

Because of their importance, MPGs have attracted many researchers, especially in the last decade, and several algorithms for solving MPGs have been proposed. They can be roughly divided into two categories. In the first category are algorithms based on linear programming [2, 13]. This category also includes algorithms based on reduction to discounted payoff games [11] and simple stochastic games [5]. In the second category are pure combinatorial graph algorithms [14, 3, 10, 6, 8, 12].

The best complexity attained by a deterministic algorithm for solving MPGs is pseudo-polynomial, namely $O(|V|^3 \cdot |E| \cdot W)$, where $|V|$ and $|E|$ are numbers of vertices and edges, respectively, and W is the maximal absolute edge-weight. It is the complexity of the algorithm of Zwick and Paterson [14] (ZP).

In this paper, we design a deterministic combinatorial algorithm with the complexity $O(|V|^2 \cdot |E| \cdot W \cdot \log(|V| \cdot W))$, which is better for W up to $2^{O(|V|)}$. To get an algorithm which is better for all values of W, we combine our algorithm with the randomized algorithm of Andersson and Vorobyov [1], and get an algorithm with currently the best expected complexity. We note that in typical applications, where the edge-weights represent, for example, the energy consumption of a physical device, W is usually small in comparison with $|V|$, in which case our deterministic algorithm is significantly better than ZP without the need to combine it with any other algorithm.

1 Preliminaries

A *Mean-Payoff Game* (MPG) [7, 8, 14] is given by a triple $(\mathcal{G}, V_{\mathrm{Max}}, V_{\mathrm{Min}})$, where $\mathcal{G} = (V, E, w)$ is a finite weighted directed graph such that V is a disjoint union of the sets V_{Max} and V_{Min}, $w : E \to \mathbb{Z}$ is the weight function, and each $v \in V$ has out-degree at least one. The game is played by two opposing players, named Max and Min. A play starts by placing a token on some given vertex and the players then move the token along the edges of $\mathcal{G}$ ad infinitum. If the token is on vertex $v \in V_{\mathrm{Max}}$, Max moves it. If the token is on vertex $v \in V_{\mathrm{Min}}$, Min moves it. This way an infinite path $p = (v_0, v_1, v_2, \ldots)$ is formed. Max's aim is to maximize his gain: $\liminf_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} w(v_i, v_{i+1})$, and Min's aim is to minimize her loss: $\limsup_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} w(v_i, v_{i+1})$. For each vertex $v \in V$, we define its *value*, denoted by $\nu(v)$, as the maximum gain that Max can ensure if the play starts at vertex v. It was proved that it is equal to the minimum loss that Min can ensure. Moreover, both players can ensure $\nu(v)$ by using positional strategies defined below [7]. A strategy that ensures $\nu(v)$, for all $v \in V$ is called an *optimal* strategy. To solve an MPG is to find the values of all vertices and, optionally, also optimal positional strategies for both players.

A *positional strategy* for Max is a function $\sigma : V_{\mathrm{Max}} \to V$ such that $(v, \sigma(v)) \in E$, for each $v \in V_{\mathrm{Max}}$ (Recall that each vertex has out-degree at least one).

A positional strategy for Min is defined analogously, except that it is usually denoted by π. We define $\mathcal{G}_\sigma$, the restriction of $\mathcal{G}$ to σ, as the graph (V, E_σ, w_σ), where $E_\sigma = \{(u,v) \in E \mid u \in V_{\text{Min}} \vee \sigma(u) = v\}$, and $w_\sigma = w|_{E_\sigma}$. That is, we get $\mathcal{G}_\sigma$ from $\mathcal{G}$ by deleting all the edges emanating from Max's vertices that do not follow σ. Let now σ be a strategy of Max and let π be a strategy of Min. $\mathcal{G}_\sigma$ has just been defined, $\mathcal{G}_\pi$ is defined analogously, and $\mathcal{G}_{\sigma\cup\pi}$ is the intersection of $\mathcal{G}_\sigma$ and $\mathcal{G}_\pi$, i.e., $\mathcal{G}_{\sigma\cup\pi} = (V, E_{\sigma\cup\pi}, w_{\sigma\cup\pi})$, where $E_{\sigma\cup\pi} = \{(u,v) \in E \mid (u \in V_{\text{Min}} \wedge \pi(u) = v) \vee (u \in V_{\text{Max}} \wedge \sigma(u) = v)\}$, and $w_{\sigma\cup\pi} = w|_{E_{\sigma\cup\pi}}$.

From the existence of optimal positional strategies it follows that $\nu(v)$ for each $v \in V$ is a a fraction with denominator at most $|V|$. It is because the ν values are the mean-weights of certain cycles, namely the cycles in $\mathcal{G}_{\sigma\cup\pi}$, where σ and π are optimal positional strategies for Max and Min, respectively. In $\mathcal{G}_{\sigma\cup\pi}$, each vertex has out-degree exactly one, and so for each $v \in V$, there is a unique cycle reachable from v, and $\nu(v)$ is equal to the mean-weight of that cycle. The fact that the ν values of vertices are mean-weights of cycles also implies that $\nu(v) \in [-W, W]$, for each $v \in V$, where $W = \max_{e\in E} |w(e)|$.

2 Algorithm

Our algorithm solves only the 0-mean partition problem. That is, it divides the vertices of the graph into those with $\nu \geq 0$ and those with $\nu < 0$. How to use the algorithm to compute the exact ν values will be described later in this section.

Our algorithm computes, for each vertex $v \in V$, the value $d_{\geq 0}(v)$ – the minimum value such that Max can ensure that the sum of traversed edges in a play starting from v, plus $d_{\geq 0}(v)$, never goes below 0. The $d_{\geq 0}$ value is finite only for vertices with $\nu \geq 0$, because for plays starting from vertices with negative ν value, Min has a strategy that ensures that all traversed cycles are negative, and so Max is unable to keep the sum of traversed edges nonnegative forever, no matter how high his starting "energy" is. Therefore, for each vertex $v \in V$ such that $\nu(v) < 0$, the algorithm sets $d_{\geq 0}(v) = \infty$.

Chakrabarti et al. [4] proposed a simple algorithm based on value iteration that solves a similar problem. The difference is that the weights are on vertices, not edges. The complexity of their algorithm is $O(|V|^2 \cdot |E| \cdot W)$. We adjusted the algorithm so that it works with weights on edges and improved its complexity to $O(|V| \cdot |E| \cdot W)$.

Our algorithm proceeds in iterations. It starts with $d_0(v) = 0$, for each $v \in V$, and then computes $d_1, d_2, \ldots$ according to the following rules.

$$d_{i+1}(v) = \begin{cases} \min_{(v,u)\in E} \max(0, d_i(u) - w(v,u)) & \text{if } v \in V_{\text{Max}} \\ \max_{(v,u)\in E} \max(0, d_i(u) - w(v,u)) & \text{if } v \in V_{\text{Min}} \end{cases} \tag{1}$$

It is easy to see that for each $v \in V$ and $k \in \mathbb{N}_0$, $d_k(v)$ is the minimum amount of Max's starting energy, that enables him to keep the sum of traversed edges, plus $d_k(v)$, greater or equal to zero in a k-step play. The computation continues until two consecutive d vectors are equal. The last d vector is then the

desired vector $d_{\geq 0}$. Please note that the d value of each vertex is non-decreasing, that is, for each $v \in V$, $d_0(v) \leq d_1(v) \leq d_2(v) \leq \cdots$.

This works well if all vertices have $\nu \geq 0$. If some vertex has $\nu < 0$, then its d value never stops increasing and the value iteration does not terminate. Fortunately, there is an upper bound on the d values of vertices with $\nu \geq 0$ and if for some vertex v, $d(v)$ goes past that bound, we know that $\nu(v) < 0$. The said bound is $(|V| - 1) \cdot W$. The reason is the following.

Max has a positional strategy σ such that for each vertex $v \in V$ such that $\nu(v) \geq 0$, all cycles reachable from v in $\mathcal{G}_\sigma$ are non-negative. For the sake of contradiction, let's suppose that Min has a strategy π, not necessarily positional, that for a play starting from some vertex v_0 such that $\nu(v_0) \geq 0$ guarantees that at some point the traversed path has weight less than $-(|V|-1)\cdot W$. Let Max use the strategy σ against π and let $p = (v_0, \ldots, v_k)$ be a path agreeing with σ and π such that $w(p) < -(|V|-1)\cdot W$. Recall that since Max uses the strategy σ, all traversed cycles must be non-negative. Since $w(p) < -(|V| - 1) \cdot W$, the number of vertices in p must be greater than $|V|$. Therefore, we can apply the following transformation on p. Start from v_0 and go along the path until an already visited vertex is encountered, then remove the found cycle from p. Since the cycle is non-negative, we get a shorter path p' such that $w(p') \leq w(p) < -(|V| - 1) \cdot W$. We can continue in this fashion until the path has less than $|V|$ vertices and get a contradiction with the fact than no path with less than $|V|$ vertices can have weight less than $-(|V| - 1) \cdot W$. Therefore, if $\nu(v) \geq 0$, then $(|V| - 1) \cdot W$ is a sufficient amount of starting energy to keep the energy level non-negative ad infinitum.

Based on the facts above, we complete the algorithm by adding a test, for each vertex $v \in V$, whether $d(v)$ is greater than $(|V| - 1) \cdot W$ or not. If it is, we set $d(v)$ to ∞, which is then handled in the usual way: $\infty - a = \infty$, where $a \in \mathbb{Z}$. This guarantees the termination of the algorithm. It follows that the d value of each vertex can be improved at most $O(|V| \cdot W)$ times, so the algorithm makes at most $O(|V|^2 \cdot W)$ iterations. Since the complexity of one iteration is $O(|E|)$, we get the overall complexity $O(|V|^2 \cdot |E| \cdot W)$. However, this can be improved to $O(|V| \cdot |E| \cdot W)$. The key ingredients are the following. The first is to keep track of the vertices that increase their d values, so that vertices that cannot bring any improvement are not explored. The second is, for each vertex $v \in V_{\text{Max}}$, to keep track of the number of successors of v that give it the minimum from (1) to be able to quickly determine whether the d value of v has to be improved.

In Figure 1 is a pseudo-code of our improved algorithm. It works with three vectors of size $|V|$, $d_{pre}, d, d' \in \mathbb{N}_0^V$. The vector d contains the current iteration d values of vertices, while d_{pre} and d' contain the previous and the next iteration d values, respectively. The initialization is on lines 2–14. It initializes all the vectors to vectors of zeros, except for the elements of d' that correspond to Max's vertices. For these elements, the algorithm already computes the next iteration d values using (1). For each Max's vertex, it also computes the number of "optimal" edges, the edges which give the vertex the minimum. These numbers are stored in the vector *nopt*. There are also two queues maintained by the

algorithm. The queue q contains the vertices with improved (increased) d value. Initially, the queue contains all vertices of the graph. The other queue is q' and it accumulates the vertices with improved d value for the next iteration.

The main loop of the algorithm is on lines 15–43. The termination criterion of the loop is the queue of vertices with improved d values being empty. Each iteration performs the following steps. It loops over the vertices with improved distance on lines 16–35 and for each improved vertex v, it explores all of its predecessors. The following steps depend on whether the predecessor u is Max's or Min's vertex. For $u \in V_{\text{Min}}$, we check if the improvement of the d value of v yields also an improvement of the d value of u. If yes, we update $d'(u)$ and if it is the first improvement of u's d value in this iteration of the main loop, we also put u into the queue of improved vertices for the next iteration, q'. If u is Max's vertex, things are a little bit more complicated.

Since according to (1), minima are computed at Max's vertices, it isn't that easy to determine whether the improvement of v's d value yields an improvement of u's value. This is where the vector *nopt* helps. The d value of u is improved only if all the successors giving it the current minimum increase their d value. The vector *nopt* holds, for each Max's vertex, the number of successors that give the vertex the current minimum. Therefore, if v is one of the vertices that give u its current minimum (condition on line 20), we decrease $nopt(u)$ by one, but only if it no longer gives u the current minimum (condition on line 21). If $nopt(u)$ drops to zero, we recompute the minimum at u, compute the value of $nopt(u)$ for the new minimum, and put u to the queue of improved vertices for the next iteration (lines 22–26). When all vertices from q are explored, we prepare the algorithm for the next iteration of the main loop.

On lines 36–41, we move all the elements from q' to q and update the vectors d_{pre} and d. The current d values of the improved vertices become their previous d values (line 39) and the new d values of these vertices become their current d values (line 40). If a new value is greater than the bound, it becomes infinity. After the queue q' is processed, we increase i and start another iteration, but only if there are some improved vertices.

The complexity analysis of the algorithm is quite simple. The d value of each vertex can be improved at most $O(|V| \cdot W)$ times, so let's determine what is the complexity of one improvement of one vertex. For definiteness, let's denote the vertex by v. If v is improved, we explore it's predecessors in the next iteration and that's $O(indegree(v))$ operations. For $v \in V_{\text{Max}}$, we also have to compute the actual value of the improvement if we detect that v's d value has to be improved. The computation explores all of the successors of v which takes $O(outdegree(v))$ time. So the overall complexity of the algorithm is $O(|V| \cdot |W| \cdot \sum_{v \in V}(indegree(v) + outdegree(v)))$, and since $\sum_{v \in V}(indegree(v) + outdegree(v)) = 2 \cdot |E|$, the overall complexity is $O(|V| \cdot |E| \cdot W)$.

Our algorithm solves only the 0-mean partition problem. It divides the vertices of $\mathcal{G}$ into those with $\nu \geq 0$ (the vertices with finite d values after the termination of the algorithm) and those with $\nu < 0$ (the vertices with infinite d values after the termination). The algorithm can also be used to solve the p-mean

```
1  proc VI(G = (V, E, w), V_Max, V_Min)
2    foreach v ∈ V do
3      d_pre(v) := 0
4      d(v) := 0
5      if v ∈ V_Max then
6        d'(v) := max(0, min_{(v,z)∈E}(0 − w(v, z)))
7        nopt(v) :=| {(v, z) ∈ E | d'(v) = max(0, 0 − w(v, z))} |
8        if d'(v) > 0 then q'.enqueue(v) fi
9      else
10       d'(v) := 0
11     fi
12     q.enqueue(v)
13   od
14   i := 0
15   while ¬q.empty() do
16     while ¬q.empty() do
17       v := q.dequeue()
18       foreach (u, v) ∈ E do
19         if u ∈ V_Max then
20           if i > 0 ∧ d(u) ≥ d_pre(v) − w(u, v) then
21             if d(v) − w(u, v) > 0 then nopt(u) := nopt(u) − 1 fi
22             if nopt(u) = 0 then
23               d'(u) := min_{(u,z)∈E}(d(z) − w(u, z))
24               nopt(u) :=| {(u, z) ∈ E | d'(u) = d(z) − w(u, z)} |
25               q'.enqueue(u)
26             fi
27           fi
28         else
29           if d'(u) < d(v) − w(u, v) then
30             if d'(u) = d(u) then q'.enqueue(u) fi
31             d'(u) := d(v) − w(u, v)
32           fi
33         fi
34       od
35     od
36     while ¬q'.empty() do
37       v := q'.dequeue()
38       q.enqueue(v)
39       d_pre(v) := d(v)
40       if d'(v) > (|V|−1) · W then d'(v) := ∞ fi; d(v) := d'(v)
41     od
42     i := i + 1
43   od
44   return ({v ∈ V | d(v) < ∞}, {v ∈ V | d(v) = ∞})
45 end
```

Fig. 1. Improved value iteration for solving the zero-mean partition problem

partition problem for arbitrary rational number p, because if we subtract p from all edge-weights, then the ν values of all vertices also decrease by p. The exact ν values of all vertices can then be computed by binary search. The complexity analysis of the resulting algorithm follows.

Since the ν values are fractions with denominators at most $|V|$, we have to solve the p-mean partition problems only for ps with denominators at most $|V|$. This increases the complexity of the 0-mean partitioning algorithm to $O(|V|^2 \cdot |E| \cdot W)$, because the smallest possible increase of a d value of a vertex is not 1 but $1/|V|$. To be able to determine the ν values exactly, we run each branch of the binary search until the size of the search interval is at most $1/|V|^2$. In such a small interval there can be only one rational number with denominator at most $|V|$, and this is the ν value of the vertices in that branch of the binary search. Since the ν value of each vertex is in the interval $[-W, W]$, the depth of the binary search is in $O(\log(|V|^2 \cdot W)) = O(\log(|V| \cdot W))$, and so the overall complexity of the algorithm is $O(|V|^2 \cdot |E| \cdot W \cdot \log(|V| \cdot W))$.

To obtain an algorithm with currently the best complexity we combine our algorithm with the randomized algorithm of Andersson and Vorobyov [1]. It has the complexity $|V|^2 \cdot |E| \cdot e^{2 \cdot \sqrt{|V| \cdot \ln(|E|/\sqrt{|V|}) + O(\sqrt{|V|} + \ln|E|)}}$, which is better than the complexity of our algorithm for large W. If we interleave the two algorithms and add a stopping criterion which terminates the computation when either of the two algorithms finishes, we get a randomized algorithm with the expected complexity $\min(O(|V|^2 \cdot |E| \cdot W \cdot \log(|V| \cdot W)), |V|^2 \cdot |E| \cdot e^{2 \cdot \sqrt{|V| \cdot \ln(|E|/\sqrt{|V|}) + O(\sqrt{|V|} + \ln|E|)}})$, which is currently the best expected complexity of an algorithm for solving mean-payoff games. This deserves a little more comments, which the following subsection is dedicated to.

2.1 Comparison with other algorithms

The algorithm of Zwick and Paterson [14] (ZP) has the complexity $O(|V|^3 \cdot |E| \cdot W)$, which is better than the first term of the complexity of our algorithm for very large W. However, it is not better for W up to $2^{O(|V|)}$, and for W in $2^{O(|V|)}$, the second term of our algorithm is already better. Therefore, our algorithm has better complexity than ZP.

We must also compare our algorithm with the algorithm of Björklund and Vorobyov [3] (BV), which has the complexity $\min(O(|V|^3 \cdot |E| \cdot W \cdot \log(|V| \cdot W)), (\log W) \cdot 2^{O(\sqrt{|V| \cdot \log|V|})})$. This can be improved to $\min(O(|V| \cdot (|V| \cdot \log|V| + |E|) \cdot W \cdot \log(|V| \cdot W)), (\log W) \cdot 2^{O(\sqrt{|V| \cdot \log|V|})})$ by the following modifications to the algorithm. The first is to use Dijkstra's algorithm instead of Bellman-Ford's algorithm, which is one of the subroutines of BV. This is made possible by a potential transformation of the edge-weights as described by Schewe [12]. The second modification is to use a technique similar to the key technique of our algorithm that improves its complexity by a factor of $|V|$. However, the first term of the complexity of BV exceeds the first term of the complexity of our algorithm, and the second term of the complexity of BV exceeds the second term

of the complexity of our algorithm, even for small W. Therefore, BV is worse. We stress that the improvement of BV outlined above is not straightforward and it is an interesting result per se, but since we achieved better complexity with our algorithm, and also for space reasons, we decided not to give the details of the improvement here.

The algorithm of Svensson and Vorobyov [13] based on linear programming has the complexity $O(|V| \cdot |E| \cdot W)$. However, it solves only the 0-mean partition problem for bipartite games with no zero cycles. The 0-mean partition problem for general games can be reduced to the special case, but the reduction increases edge-weights by a factor of $|V|$. The exact ν values of all vertices can be computed by binary search, but, as already mentioned, it requires solving p-mean partition problems for rational ps, which increases the complexity by another factor of $|V|$. All in all, computation of the exact ν values using the algorithm of Svensson and Vorobyov and binary search has the complexity $O(|V|^3 \cdot |E| \cdot W \cdot \log(|V| \cdot W))$, which exceeds the complexity of our algorithm.

The algorithm of Lifshits and Pavlov [10] has the complexity $O(|V| \cdot |E| \cdot 2^{|V|} \cdot \log(W))$, which is worse than the second term of the complexity of our algorithm.

All the other algorithms for solving MPGs we know of have either the same or worse complexity than the algorithms we have already compared our algorithm with.

3 Conclusion

We designed a deterministic algorithm for solving mean-payoff games with the complexity $O(|V|^2 \cdot |E| \cdot W \cdot \log(|V| \cdot W))$. In combination with the randomized algorithm of Andersson and Vorobyov [1], it is a randomized algorithm with currently the best expected complexity, namely:

$$\min(O(|V|^2 \cdot |E| \cdot W \cdot \log(|V| \cdot W)), |V|^2 \cdot |E| \cdot e^{2 \cdot \sqrt{|V| \cdot \ln(|E|/\sqrt{|V|})} + O(\sqrt{|V|} + \ln|E|)})$$

References

1. D. Andersson and S. Vorobyov. Fast algorithms for monotonic discounted linear programs with two variables per inequality. Technical Report Preprint NI06019-LAA, Isaac Newton Institute for Mathematical Sciences, Cambridge, UK, 2006.
2. H. Björklund, O. Svensson, and S. Vorobyov. Linear complementarity algorithms for mean payoff games. Technical Report DIMACS-2005-13, DIMACS, New Jersey, USA, 2005.
3. H. Björklund and S. Vorobyov. A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. *Discrete Applied Math.*, 155(2):210–229, 2007.
4. A. Chakrabarti, L. de Alfaro, T. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. Embedded Software*, volume 2855 of *LNCS*, pages 117–133. Springer, 2003.

5. A. Condon. On algorithms for simple stochastic games. In *Advances in Computational Complexity Theory*, volume 13 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 51–73. American Mathematical Society, 1993.
6. V. Dhingra and S. Gaubert. How to solve large scale deterministic games with mean payoff by policy iteration. In *Proc. Performance evaluation methodolgies and tools,* article no. 12. ACM, 2006.
7. A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8(2):109–113, 1979.
8. V. A. Gurvich, A. V. Karzanov, and L. G. Khachivan. Cyclic games and an algorithm to find minimax cycle means in directed graphs. *USSR Comput. Math. and Math. Phys.*, 28(5):85–91, 1988.
9. M. Jurdziński. Deciding the winner in parity games is in UP ∩ co-UP. *Inf. Process. Lett.*, 68(3):119–124, 1998.
10. Y. Lifshits and D. Pavlov. Fast exponential deterministic algorithm for mean payoff games. *Zapiski Nauchnyh Seminarov POMI*, 340:61–75, 2006.
11. A. Puri. Theory of hybrid systems and discrete event systems. Phd thesis, EECS University of Berkeley, Berkeley, CA, USA, 1995.
12. S. Schewe. An optimal strategy improvement algorithm for solving parity and payoff games. In *Proc. Computer Science Logic*, volume 5213 of *LNCS*, pages 369–384. Springer, 2008.
13. O. Svensson and S. Vorobyov. Linear programming polytope and algorithm for mean payoff games. In *Proc. Algorithmic Aspects in Information and Management*, volume 4041 of *LNCS*, pages 64–78. Springer, 2006.
14. U. Zwick and M. S. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1–2):343–359, 1996.