UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN
ABTEILUNG INFORMATIK UND ANGEWANDTE KOGNITIONSWISSENSCHAFT

Master's Thesis

Infinite Games: Algorithms and Reductions

Maxime Nederkorn
Matrikelnummer: 3004376

UNIVERSITÄT DUISBURG ESSEN

Department of Computer Science and
Applied Cognitive Science
Faculty of Engineering
University of Duisburg-Essen

7. March 2022

Examiners:
Prof. Dr. Barbara König
Prof. Dr. Janis Voigtländer

Advisor:
Richard Eggert

Contents

1. Introduction **4**
- 1.1. Task . . . 4
- 1.2. Structure . . . 4

2. Preliminaries **5**
- 2.1. Directed Graphs . . . 5
- 2.2. Arenas . . . 5
- 2.3. Positions, Moves . . . 6
- 2.4. Strategies . . . 6
- 2.5. Plays . . . 6
- 2.6. Infinite Games . . . 6
  - 2.6.1. Parity Games . . . 7
  - 2.6.2. Mean Payoff Games . . . 7
  - 2.6.3. Energy Games . . . 8
  - 2.6.4. Discounted Payoff Games . . . 8
  - 2.6.5. Simple Stochastic Games . . . 9
- 2.7. Reductions . . . 9

3. Solutions and Reductions in Theory **11**
- 3.1. Solutions in Theory . . . 11
  - 3.1.1. Kleene Iteration . . . 12
  - 3.1.2. Strategy Iteration . . . 12
  - 3.1.3. PGs . . . 12
  - 3.1.4. MPGs . . . 12
  - 3.1.5. DPGs . . . 12
  - 3.1.6. EGs . . . 12
  - 3.1.7. SSGs . . . 12
- 3.2. Reductions in Theory . . . 13
  - 3.2.1. PGs to MPGs . . . 13
  - 3.2.2. MPGs to DPGs . . . 15
  - 3.2.3. MPGs to EGs . . . 18
  - 3.2.4. DPGs to SSGs . . . 19

4. Solutions and Reductions in Practice **20**
- 4.1. Solutions in Practice . . . 21
  - 4.1.1. PGs . . . 21
  - 4.1.2. MPGs . . . 24
  - 4.1.3. EGs . . . 27
  - 4.1.4. DPGs . . . 33
  - 4.1.5. SSGs . . . 34
- 4.2. Reductions in Practice . . . 37
  - 4.2.1. PGs to MPGs . . . 37
  - 4.2.2. MPGs to EGs . . . 38
  - 4.2.3. MPGs to DPGs . . . 40
  - 4.2.4. DPGs to SSGs . . . 41

5. Implementation **42**

6. Evaluation **43**
6.1. PGs . . . 43
6.1.1. Value Problem of PGs . . . 44
6.1.2. Strategy Problem of PGs . . . 45
6.2. MPGs . . . 46
6.2.1. Value Problem of MPGs . . . 47
6.2.2. Strategy Problem of MPGs . . . 48
6.3. EGs . . . 49
6.4. DPGs . . . 50
6.5. SSGs . . . 51

7. Conclusion and Future **52**
7.1. Conclusion . . . 52
7.2. Future . . . 53

A. Versicherung an Eides Statt **54**

References **55**

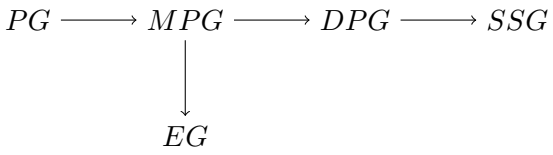
-

1. Introduction

There are practical applications (cite) which are able to be modeled by or reduced to various games. Considering the many ways there are to solve those games, the possibility to (repeatedly) reduce them to one another and various ways to implement any of those algorithms there are many potential means to solve problems on those games. We want to see how the different ways to solve problems, the reductions between those problems and the combination of the two play out and compare to one another in a real-world implementation.

1.1. Task

The games we concern ourselves here are *Parity Games* (PG), *Mean Payoff Games* (MPG), *Energy Games* (EG), *Discounted Payoff Games* (DPG) and (stopping) *Simple Stochastic Games* (SSG). For any of those games there are multiple problems, but the ones we concern outselves here are "what is the value of vertex v under optimal strategies from both players?" and "what are the optimal strategies?". For both of those problems, there may already be multiple algorithms to solve them for any specific game directly, some of which are generalizable to multiple of the games to some degree, such as Kleene Iteration and Strategy Iteration. Additionally we can reduce the games to another as follows:



The reductions themselves are computationally relatively simple compared to the problems themselves. Some of them, however, produce graphs that are bigger in some way ($PG \to MPG$, $DPG \to SSG$), which increases the complexity of subsequently applied algorithms, or recursively reduces to multiple sub-problems ($MPG \to EG$). Another part we have control over is how exactly the algorithms are implemented. The biggest factor here being that, due to their nature, the edge-weights can be seen as an incidence matrix and can therefore make use of matrix operations, especially for simple operations such as addition and multiplication.

1.2. Structure

The paper is structured as follows:

In Chapter 2 we will first define the different kinds of games we are concerned with and the notion of *value* as on objective for the respective games as well as (memory-less) strategies as a means to achieve those values.

In Chapter 3 we will first explain in 3.1 the idea of value and strategy iteration and then both concrete implementations of those as well as other algorithms for the respective games. In 3.2 we will then show how the different games and their underlying graphs can be translated to one another and how the problems posed on those games are finally reduced.
In Chapter 4 we show the specifics as to how the theoretical algorithms were internally implemented and elaborate on the choices and decisions arising in the implementations of the reductions and solutions.
In Chapter 5 we show how the implementations embeds to be used to solve specific problems or to demonstrate with the GUI.
In Chaper 6 we show the kinds of graphs and problems we used to evaluate the solutions and the potential benefit of prior reduction as well as the results of the evaluation.
In Chapter 7 we interpret the results of the evaluation and give suggestions and ideas for further improvement.

2. Preliminaries

Foundational to our task are the different kinds of *Infinite Games* and how to determine the outcome of each such game. To do so, we also want a notion of *strategies* on such Infinite Games. To later reduce the games to one another, we furthermore want a definition of a *reduction* in the computational complexity sense.

2.1. Directed Graphs

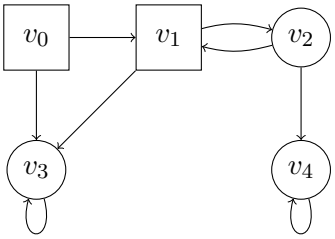
Let V be a finite set of Vertices, let $E \subseteq V \times V$ be a set of Edges, then $G = (V, E)$ is a *Directed Graph*. We also define

$$\begin{aligned} pre\colon V \to \mathcal{P}(V) \qquad & pre(v) = \{u \in V \mid (u, v) \in E\} \\ post\colon V \to \mathcal{P}(V) \qquad & post(v) = \{u \in V \mid (v, u) \in E\} \end{aligned}$$

as the set of vertices *from* which a vertex v is reachable (pre) and the set of vertices that *to* which a vertex v can reach ($post$).

2.2. Arenas

An arena is an extension a Directed Graph, where the set of Vertices, V, is partitioned into two disjunct subsets V_0 and V_1, respectively denoting the regions where Player 0, also represented by $\square$, and Player 1, also represented by $\bigcirc$, are to play. We also require that the out-degree of every vertex is at least one, so that a play on an Arena never stops.
Formally, let (V, E) be a non-trivial Directed Graph, $V_0 \cup V_1 = V$, $V_0 \cap V_1 = \emptyset$ be a partition of V and for all $v \in V\colon post(v) \neq \emptyset$, then $A = (V, (V_0, V_1), E)$ is an Arena.



Example 1:

An Arena

$$A = (\{v_0, v_1, v_2, v_3, v_4\}, (\{v_0, v_1\}, \{v_2, v_3, v_4\}),$$
$$\{(v_0, v_1), (v_0, v_3), (v_1, v_2), (v_2, v_1),$$
$$(v_2, v_4), (v_4, v_4), (v_1, v_3), (v_3, v_3)\})$$

Figure 1: Arena A

2.3. Positions, Moves

Starting at $v_0 \in V$, a *position* $\pi_i = (v_0, v_1, \ldots, v_i)$ in a Game describes a finite path on the underlying graph, for which $\forall n \in \{0, ..., i-1\} : (v_n, v_{n+1}) \in E$. E.g. in Fig. 1 starting at v_0, (v_0, v_1, v_2, v_4) could be a play.
A *move* is an extension of a position in the graph by one more step. E.g. in Fig. 1 $(v_0, v_1, v_2) \mapsto (v_0, v_1, v_2, v_4)$ could be a move. Player i chooses the n-th move $(\ldots, v_{n-1}) \mapsto (\ldots, v_{n-1}, v_n)$ for $v_{n-1} \in V_i \in (V_0, V_1)$.

2.4. Strategies

A *strategy* $V^* \times V_i \to V$, with $V^* \in \mathcal{P}(V)$ and $V^* \times V_i$ being a valid *position*, is a function by which Player i assigns the next move for any given position.
We call a strategy *memoryless*, if for any given position the next move only depends on the last vertex of the position, i.e. it is a function: $V_i \to V$. We will refer to memoryless strategies of Player 0 as σ and of Player 1 as τ. E.g. in Fig. 1 $\sigma = \{(v_0, v_1), (v_1, v_2)\}$ and $\tau = \{(v_2, v_1), (v_3, v_3), (v_4, v_4)\}$ could be strategies.
We call a memoryless strategy *optimal*, if it optimizes the value of the game that the opponent can force with their strategy. Notice that there may be multiple optimal strategies for any given game.

2.5. Plays

A *play* describes the path of arbitrary length $\langle v_0, v_1, \ldots \rangle$ the player go trough in the process of playing the game. We refer to the play generated by applying strategies σ, τ to game G starting at $v_0 \in V$ as $\pi_{\sigma,\tau}(G, v_0) = \langle v_0, v_1, \ldots \rangle$
E.g. if we take the previous example's strategies and apply them to $\pi_{\sigma,\tau}(A, v_0)$ we get the play $\langle v_0, v_1, v_2, v_1, \ldots \rangle$. The play can be arbitrarily prolonged by applying moves from σ, τ.

2.6. Infinite Games

Infinite Games are a category of games played by two players on a finite, directed graph. They are infinite in the sense that we require the out-degree of every vertex to be at least one and as such, regardless of the strategies chosen by the players, they never terminate.

2.6.1. Parity Games

Parity Games are played by two players, *Even* or Player 0 (□) and *Odd* or Player 1 (○). A Parity Game, $PG = (A, p)$, is played on an Arena A with a priority function $p\colon V \to \mathbb{N}_0$.
Let $\pi_{\sigma,\tau}(PG, v_0) = \langle v_0, v_1, \ldots\rangle$ be the *play* resulting from applying the strategies σ and τ to Parity Game PG. Let

$$\#_\infty(\pi_{\sigma,\tau}(PG, v_0)) =$$
$$\{i \in \{0, 1, \ldots, |V|\} \mid \forall j \in \mathbb{N}_0 \colon \exists n \in \mathbb{N}_0 \colon j < |\langle v \in \langle v_0, \ldots, v_n\rangle \colon p(v) = i\rangle|\}$$

be the set of priorities that appear arbitrarily often in the play.
If $max(\#_\infty(\pi_{\sigma,\tau}(PG, v_0))) \coloneqq v_{\sigma,\tau}(v_0)$, the value of vertex v_0 is even, then *Even* wins and vice versa. Optimal strategies for *Even*/*Odd* are those that result in the most starting vertices resulting in an even/odd value.

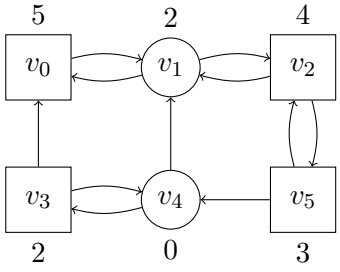


Figure 2: PG

Example 2:

Playing PG with $\sigma = \{(v_0, v_1), (v_2, v_5), (v_3, v_4), (v_5, v_2)\}$ and $\tau = \{(v_1, v_0), (v_4, v_1)\}$, the respective optimal strategies, results in play $\pi_{\sigma,\tau}(PG, v_0) = \langle v_0, v_1, v_0, ..\rangle$ for which $v_{\sigma,\tau}(v_0) = 5$ is odd and the play is therefore winning for *Odd*.

2.6.2. Mean Payoff Games

Mean Payoff Games are played by two players, *Max* or Player 0 (□) and *Min* or Player 1 (○). A Mean Payoff Game, $MPG = (A, w)$, is played on an Arena A and an edge-weight function $w\colon E \to \{-W, \ldots, -1, 0, 1, \ldots, W\}$, $W \in \mathbb{N}_0$.
Max aims to chose their strategy for a given play $\pi_{\sigma,\tau}(MPG, v_0) = \langle v_0, v_1, \ldots\rangle$ such as to maximize the *mean payoff*

$$\liminf_{n\to\infty} \left(\frac{1}{n} \sum_{i=0}^{n-1} w((v_i, v_{i+1})) \right).$$

We call this the *value* $v_{\sigma,\tau}(v)$ of a vertex v. Given that those values are real numbers, we will round them from here on out. Optimal strategies are those that maximize/minimize the mean payoff for a given MPG regardless of the initial vertex.

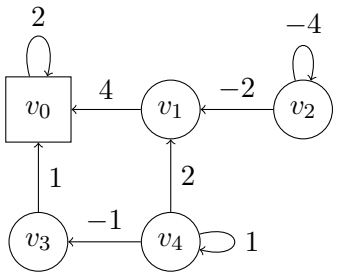


Figure 3: MPG

Example 3:

A Mean Payoff Game
$MPG = (A, \{((v_0, v_0), 2), ((v_1, v_0), 4), \ldots\})$.
Playing MPG with $\sigma = \{(v_0, v_0)\}$ and
$\tau = \{(v_1, v_0), (v_2, v_2), (v_3, v_0), (v_4, v_4)\}$, the respective optimal strategies, results in the values
$v_{\sigma,\tau}\colon v_0 \mapsto 2, v_1 \mapsto 2, v_2 \mapsto -4, v_3 \mapsto 2, v_4 \mapsto 1$

2.6.3. Energy Games

Energy Games are played by two players, *Charging* or Player 0 ($\square$) and *Depleting* or Player 1 ($\bigcirc$). An Energy Game, $EG = (A, w)$, is played on an Arena A and an edge-weight function $w\colon E \to \{-W, \ldots, -1, 0, 1, \ldots, W\}$, $d \in \mathbb{N}_0$.
Charging aims to chose their strategy for a given play $\pi_{\sigma,\tau}(EG, v_0) = \langle v_0, v_1, \ldots \rangle$ such as to minimize the *initial credit* $c \in \mathbb{N}_0$ needed to maintain the winning condition

$$\forall k \in \mathbb{N}_0\colon \left(\sum_{i=0}^{k} w((v_i, v_{i+1})) \right) + c \geq 0.$$

We call the smallest inital credit that still maintains the winning condition for a given play the *minimum initial credit* or *value* $v_{\sigma,\tau}(v)$ of a vertex v. If no such value exists, e.g. when the vertex is of *Depleting* and has a reflexive edge with negative weight, we write $v_{\sigma,\tau}(v) = \infty$. For any given position $\langle v_0, v_1, \ldots, v_k \rangle$ in a play we call $\left(\sum_{i=0}^{k-1} w((v_i, v_{i+1}))\right) + c$ the *energy level* at that position. Keep in mind that *Charging* doesn't necessarily aim to maximise their energy level at *any* specific position but rather to maintain a non-negative energy level at *every* position.

The goal for *Charging* is to avoid getting trapped in cycles with overall negative edge-weight, as these can deplete any initial credit given, as well as to minimize the initial credit necessary to the compliant with the winning condition in all other cycles. Conversely, the goal for *Depleting* is to trap *Charging* in negative cycles or at least to maximize the initial credit necessary for *Charging* to reach a non-negative cycle. Optimal strategies are those that minimize the initial credit necessary for *Charging* and those that either deny the winning condition entirely or maximize the initial credit necessary for *Depleting*.

Example 4:

We reuse A and w of Example 3 and create an Energy Game
$EG = (A, 0, 4, \{((v_0, v_0), 2), ((v_1, v_0), 4), \ldots\})$. Playing EG with $\sigma = \{(v_0, v_0)\}$ and $\tau = \{(v_1, v_0), (v_2, v_2), (v_3, v_0), (v_4, v_3)\}$, respective optimal strategies, results in the values $v_{\sigma,\tau}\colon v_0 \mapsto 0, v_1 \mapsto 0, v_2 \mapsto \infty, v_3 \mapsto 0, v_4 \mapsto 1$.

2.6.4. Discounted Payoff Games

Discounted Payoff Games are played by two players, *Max* or Player 0 ($\square$) and *Min* or Player 1 ($\bigcirc$). A Discounted Payoff Game, $DPG = (A, w, \lambda)$, is played on an Arena A, an edge-weight function $w\colon E \to \{-W, \ldots, -1, 0, 1, \ldots, W\}$, $d \in \mathbb{N}_0$ and a discount factor $0 < \lambda < 1$.
Max aims to chose their strategy for a given play $\pi_{\sigma,\tau}(DPG, v_0) = \langle v_0, v_1, \ldots \rangle$ such as to maximize the *discounted payoff*

$$(1 - \lambda) \left(\sum_{i=0}^{\infty} \lambda^i \cdot w((v_i, v_{i+1})) \right).$$

Again we call this the *value* $v_{\sigma,\tau}(v)$ of a vertex v, we will round them from here on out. Optimal strategies are those that maximize/minimize the discounted payoff for a given MPG regardless of the initial vertex.

Example 5:

Let us again reappropriate Arena A and edge-weight function w and create a Discounted Payoff Game $DPG = (A, \{((v_0, v_0), 2), ((v_1, v_0), 4), \ldots\}, 0.95)$.
Playing DPG with $\sigma = \{(v_0, v_0)\}$ and $\tau = \{(v_1, v_0), (v_2, v_2), (v_3, v_0), (v_4, v_4)\}$, the respective optimal strategies, results in the values $v_{\sigma,\tau}\colon v_0 \mapsto 2, v_1 \mapsto 2.1, v_2 \mapsto -4, v_3 \mapsto 1.95, v_4 \mapsto 1$.

2.6.5. Simple Stochastic Games

Simple Stochastic Games are played by two players, *Max* or Player 0 (□) and *Min* or Player 1 (○). A Simple Stochastic Game, $SSG = (G, (V_0, V_1, V_2), \mathtt{0}, \mathtt{1}, p)$, is played on a Directed Graph G, with a partition (V_0, V_1, V_2), two sink vertices $\mathtt{0}, \mathtt{1}$ and a probability function $p\colon V_2 \to (V \to [0, 1])$. We require that the probabilites of all outgoing edges of each V_2 vertex sum to 1: $\forall v \in V_2\colon \sum_{u \in V} p(v)(u) = 1$ and that $\forall (u, v) \in (V_2 \times V) \setminus E\colon p_u(v) = 0$. On vertices $v \in V_2$, called *Random* (◊) vertices, the next vertex gets chosen according to the probability function $p_v\colon V \to [0, 1]$ rather than a player. We also require, like for Arenas, that the out-degree of every vertex is at least one. For $\mathtt{0}, \mathtt{1}$ it is exactly one with each only having a reflexive edge.
Max wins if the $\mathtt{1}$ sink is reached, *Min* wins if the $\mathtt{0}$ sink is reached or the game doesn't reach a sink vertex. Since the result of a play $\pi_{\sigma,\tau}(SSG, v_0)$ can be probablistic, it is assigned a probabilty to reach the $\mathtt{1}$ sink rather than a fixed value. We display this probability as a rounded real number. We say that a SSG is *stopping* if for every possible combination of strategies σ, τ, there still is a path to one of the sink vertices. All SSGs we will consider from here on out will be stopping-SSGs.
The objective for *Max* in a stopping-SSG is to maximize the chance to reach the $\mathtt{1}$ vertex and for *Min* to reach the $\mathtt{0}$ vertex. Optimal strategies are those that maximize the chance to reach the desired sink-vertex of the respective player.

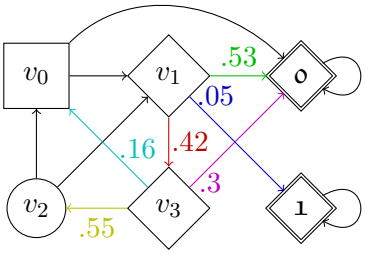


Figure 4: SSG

Example 6:

A Simple Stochastic Game

$$SSG = (G, (\{v_0\}, \{v_2\}, \{v_1, v_3\}), \mathtt{0}, \mathtt{1},$$
$$\{(v_1, \{(\mathtt{0}, .53), (\mathtt{1}, .05), (v_3, .42)\}),$$
$$(v_3, \{(\mathtt{0}, .3), (v_0, .16), (v_2, .55)\})\}).$$

Playing SSG with $\sigma = \{(v_0, v_1)\}$ and $\tau = \{(v_2, v_0)\}$, the respective optimal strategies, results in the values $v_{\sigma,\tau}\colon v_0 \mapsto .07, v_1 \mapsto .07, v_2 \mapsto .07, v_3 \mapsto .05, \mathtt{0} \mapsto 0, \mathtt{1} \mapsto 1$.

2.7. Reductions

A *reduction* describes an algorithm with which we can transform (reduce) one class of problems to another. For our purposes, we choose target problem classes that are

already solved. Aside from reducing to an already solved problem class and therefore, per definition, also solving the original problem class, reducing and subsequent solving of instances of the original problem may be faster than any attempt to directly solve the original problem without reducing to another intermediary infinite game problem.
Formally, we express our problem classes as formal languages A and B over the alphabets Σ^* and Γ^*. If f is totally computable and

$$f\colon \Sigma^* \to \Gamma^*, A \subseteq \Sigma^*, B \subseteq \Gamma^*$$
$$\forall w \in \Sigma^*\colon w \in A \iff f(w) \in B$$

then f is a reduction from Σ^* to Γ^*.
In practice our formal languages A and B will be some instances of types of infinite games together with statements about such games, such as the values under optimal play or optimal strategies. The reduction will then draw an equivalence to the other infinite game, e.g. if $w =$ "$v(v_u) = 5$ in G" is a word in A then $f(w) =$ "$v(v_v) = 16$ in H" is a word in B.

3. Solutions and Reductions in Theory

Before we go into the specifics of the implementation of the algorithms used to solve the problems and the reductions we first want to comprehensively explain the theory behind each of them.

3.1. Solutions in Theory

One of the key parts of our work is the solving of problems on our games, be it in conjunction with a reduction or not.
We can split our algorithms to find values into broadly three categories:

- Kleene Iteration Based:
  - Kleene Iteration for DPGs †
  - Kleene Iteration for SSGs †
- Strategy Iteration Based:
  - Strategy Iteration for DPGs ‡
  - Strategy Iteration for EGs:
    - from above, utilizes Linear programming ‡
    - from below, utilizes Kleene Iteration ‡
  - Strategy Iteration for SSGs ‡
- Unique
  - Zwick and Paterson's algorithm for MPGs
  - Zielonka's algorithm for PGs
  - BCDGR's algorithm for EGs ‡

Algorithms marked with † also provide optimal strategies alongside the values, those marked ‡ provide an optimal strategy for one of the players. Those that don't natively provide optimal strategies can be still be utilized to find them by progressively removing edges from the game and seeing if that affects the values until all but one outgoing edge per vertex have been removed from the game, which constitute edges that can form an optimal strategy.

First we shall explain the generalized idea of Kleene Iteration and Strategy Iteration, as they can be used to solve multiple of the problems posed. Then we will explain the specifics of each implementation of those as well as unique algorithms used to solve each of the games we are interested in.

3.1.1. Kleene Iteration

3.1.2. Strategy Iteration

3.1.3. PGs

3.1.4. MPGs

3.1.5. DPGs

3.1.6. EGs

3.1.7. SSGs

3.2. Reductions in Theory

The reductions we use here allow us to formulate the problem of finding the value or optimal strategy of one game in a way that we can find them by solving problems in another game. For $PG \to MPG$ and $MPG \to DPG$ the graph structure stays the exact same, for $MPG \to EG$ the graph is originally the same but while solving we recursively build a binary tree that proceeds on subgraphs of the original graph. For $DPG \to SSG$ the edges of the graph of the DPG get replaced by a certain construct that maintains a similarity between the two in *some* sense, but the set of vertices and edges grows considerably in doing so.

3.2.1. PGs to MPGs

The reduction from PGs to MPGs is based on the idea of [Jur98]. For a given Parity Game $PG = (A, p)$ we transform it into a Mean Payoff Game $MPG = (A, w)$, with the underlying arena A being the same and the weights w of MPG based directly on the priority function p of PG. Optimal strategies in MPG translate one to one to PG and the vertex v having value ≥ 0 in MPG translates to it being winning for *Even* in PG. Recall that for both PGs and $MPGs$ winning strategies can be memoryless.

Consider any play $\pi_{\sigma,\tau}$ in A with $n = |V|$. Given that A is finite and $\pi_{\sigma,\tau}$ can continue infinitely, any such play, regardless of starting vertex v_0 will necessarily have a repeat vertex after a path of length $k \leq n$ and will then, by virtue of the strategies being memoryless, indefinitely continue in a cycle of length $l \leq n - k$.

From the definition of the value of a PG it follows directly that only the priorities of the vertices in the cycle matter for the outcome to the game, specifically the highest priority. For MPGs we have:

$$\liminf_{n\to\infty} \left(\frac{1}{n} \sum_{i=0}^{n-1} w((v_i, v_{i+1})) \right) =$$

$$\liminf_{n\to\infty} \left(\frac{1}{n} \sum_{i=k}^{n-1} w((v_i, v_{i+1})) \right) + \liminf_{n\to\infty} \left(\frac{1}{n} \sum_{i=0}^{k-1} w((v_i, v_{i+1})) \right) =$$

$$\liminf_{n\to\infty} \left(\frac{1}{n} \sum_{i=0}^{n-1} w((v_{i+k}, v_{i+1+k})) \right) =$$

$$\liminf_{n\to\infty} \left(\frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k})) \right) =$$

$$\frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k}))$$

So again, the only edge-weights that matter for the value of a play are those that are part of the terminal cycle.

To equate the values of the games we want to set the edge-weights in such a way that the outgoing edge of any vertex of a given cycle alone can set the parity of the value of

the game if and only if the priority of the vertex is the highest in the cycle.
[Jur98] does this by setting $w((u,v)) = (-n)^{p(u)}$. This way, the absolute weight of the outgoing edge of the vertex with highest priority is always higher than any possible make-up of the rest of the cycle, i.e. $|(-n)^{p(u)}| > (n-1)|(-n)^{p(u)-1}|$.
This, however, is unsatisfying since the complexity of solving MPGs with the algorithms we use is dependent on d, the maximum absolute weight in the MPG. We therefore want to be conservative with the edge-weights and try to minimize them.

Rather than making sure that the weight is larger, in absolute terms, compared to vertices of lower priority regardless of the count and parity of the priority of the vertices with lower priority, we want to account for both of those.
To aid with that we construct an intermediary function w', which assigns weights to the vertices instead of the edges. Then we define the set V_u which for each vertex u represents the set of vertices with lower and opposing parity of priority:

$$V_u = \{v \in V \mid p(x) < p(u) \wedge (p(x) \neq p(u) \bmod 2)\}$$

The goal then is to define the weights of the vertices in such a way that they are always exactly as large as needed. For a given vertex u, we want it's weight to be exactly as large as the sum of the weights of the vertices in V_u if u belongs to *Even/Max* and exactly one larger than the sum if u belongs to *Odd/Min*. We thus construct w' as follows:

$$w'(u) = (-1)^{p(u)} \cdot \left(\left(\sum_{x \in V_u} |w'(x)| \right) + 1^{p(u) \bmod 2} \right)$$

Finally we simply define the edge-weights as $w((u,v)) = w'(u)$. This way any vertex will have an outgoing edge with weight larger or equal to what the opponent can produce without visiting a vertex of higher priority, regardless of their counterstrategy τ. Thus any play utilizing memoryless strategies on PG whose value is *Even/Odd* will have a value of $\geq 0 / \leq -1$ when translated to an MPG in such a way and any optimal strategy in *PG* will accordingly be an optimal strategy in *MPG*. Conversely, any MPG that will be reduced-to in such a way will have an edge in it's terminal cycle whose absolute weight will be greater or equal to the sum of all the edges of the opponent in said cycle. The vertex said greatest edge emanates from being exactly equivalent to the vertex with highest priority in the original PG. Thus any play on such MPG whose value, or equivalently whose greatest edge, is $\geq 0 / \leq -1$ will have *Even/Odd* value in the original MPG and any optimal strategy in MPG will be a optimal strategy in PG.

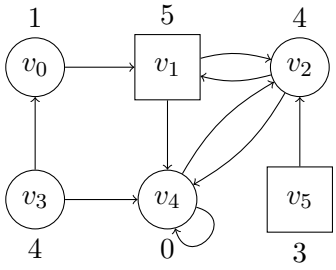


Figure 5: PG

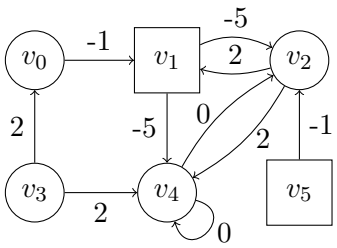


Figure 6: MPG

Example 7: We translate PG to MPG.
The edge-weights are successively built as follows:

$u \in V$	$p(u)$	V_u	$w'(u)$
v_4	0	$\emptyset$	0
v_0	1	$\{v_4\}$	-1
v_5	3	$\{v_4\}$	-1
v_2, v_3	4	$\{v_0, v_5\}$	2
v_1	5	$\{v_2, v_3, v_4\}$	-5

Here we can limit d to 5 instead of $|(-6)^5| = 7776$.

3.2.2. MPGs to DPGs

The reduction from MPGs to DPGs is the one described in [ZP96]. The underlying Arena A and edge-weights w stay the same. Recall from 3.2.1 that a play in an Arena eventually results in a repeat vertex after a path of length $k \leq n$ and then indefinitely continues in a cycle of length $l \leq n - k$. In a MPG the value of such play being:

$$\frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k}))$$

For DPGs we have the value being:

$$v(v_0) = \ (1-\lambda) \sum_{i=0}^{\infty} \left(\lambda^i \cdot w((v_i, v_{i+1}))\right)$$

We want to rearrange that value, in dependance of λ, as to approach the value of an equivalent MPG. We factor out the part that pertains to the path of length k that leads us to our terminal cycle.

$$\begin{aligned} &= \ (1-\lambda) \sum_{i=k}^{\infty} \left(\lambda^i \cdot w((v_i, v_{i+1}))\right) + (1-\lambda) \sum_{i=0}^{k-1} \left(\lambda^i \cdot w((v_i, v_{i+1}))\right) \\ &= \ (1-\lambda)\lambda^k \sum_{i=0}^{\infty} \left(\lambda^i \cdot w((v_{i+k}, v_{i+1+k}))\right) + (1-\lambda) \sum_{i=0}^{k-1} \left(\lambda^i \cdot w((v_i, v_{i+1}))\right) \end{aligned}$$

The terminal circle can then be reimagined: Instead of summing up every single step one by one we split the summation into two parts. One part, with the index j, describing the individual steps of the cycle and one part, with the index i, for each of the infinite circumnavigations of that cycle.

$$
\begin{aligned}
&= (1-\lambda)\lambda^k \sum_{i=0}^{\infty} \lambda^{il} \cdot \sum_{j=0}^{l-1} \left(\lambda^j \cdot w((v_{j+k}, v_{j+1+k}))\right) + (1-\lambda) \sum_{i=0}^{k-1} \left(\lambda^i \cdot w((v_i, v_{i+1}))\right) \\
&=^{1} \frac{(1-\lambda)\lambda^k}{1-\lambda^l} \sum_{j=0}^{l-1} \left(\lambda^j \cdot w((v_{j+k}, v_{j+1+k}))\right) + (1-\lambda) \sum_{i=0}^{k-1} \left(\lambda^i \cdot w((v_i, v_{i+1}))\right) \qquad (1)
\end{aligned}
$$

Now if we let $\lambda \to 1^-$ we get:

$$
\begin{aligned}
&\lim_{\lambda \to 1^-} \left(\frac{(1-\lambda)\lambda^k}{1-\lambda^l} \sum_{j=0}^{l-1} \left(\lambda^j \cdot w((v_{j+k}, v_{j+1+k}))\right) + (1-\lambda) \sum_{i=0}^{k-1} \left(\lambda^i \cdot w((v_i, v_{i+1}))\right) \right) \\
&= \lim_{\lambda \to 1^-} \left(\frac{1-\lambda}{1-\lambda^l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k})) \right) \\
&=^{2} \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k}))
\end{aligned}
$$

We can't let λ be 1, since both the normalizing $(1-\lambda)$ would result in a division by zero as well as the actual series diverging, we have to use a $\lambda < 1$ that is close enough to 1, and by extension the value close enough to that of the equivalent MPG, as to enable us to truncate to the nearest rational number as was done in 3.1.4.1. We can take (1) and add W to all the weights. This makes is so that all weights are effectively ≥ 0, the value

[1] Being a geometric series, $\sum_{i=0}^{\infty} \lambda^{il} = \sum_{i=0}^{\infty} (\lambda^l)^i = \frac{1}{1-\lambda^l}$, knowing that we have $l \geq 1$ and $|\lambda| \leq 1$.
[2] $\lim_{\lambda \to 1^-} \frac{1-\lambda}{1-\lambda^l} = \frac{1}{l}$ via L'Hôpital's rule

of the game is increased by W and we can simplify to a lower bound of $v(v_0)$:

$$\begin{aligned}
& v(v_0) + W \\
= \; & \frac{(1-\lambda)\lambda^k}{1-\lambda^l} \sum_{j=0}^{l-1} \left(\lambda^j \cdot (w((v_{j+k}, v_{j+1+k})) + W)\right) + (1-\lambda) \sum_{i=0}^{k-1} \left(\lambda^i \cdot (w((v_i, v_{i+1})) + W)\right) \\
\geq \; & \frac{(1-\lambda)\lambda^k}{1-\lambda^l} \sum_{j=0}^{l-1} \left(\lambda^j \cdot (w((v_{j+k}, v_{j+1+k})) + W)\right) \\
\geq \; & \frac{(1-\lambda)\lambda^{k+l-1}}{1-\lambda^l} \sum_{j=0}^{l-1} (w((v_{j+k}, v_{j+1+k})) + W) \\
\geq \; & \frac{l(1-\lambda)\lambda^{k+l-1}}{1-\lambda^l} \left(W + \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k}))\right) \\
\geq \; & (1 - n(1-\lambda)) \left(W + \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k}))\right) \\
\geq \; & W + \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k})) - n(1-\lambda) \left(W + \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k}))\right)
\end{aligned}$$

Now if we revert the scaling we arrive at:

$$\begin{aligned}
v(v_0) \geq \; & \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k})) - n(1-\lambda) \left(W + \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k}))\right) \\
\geq \; & \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k})) - n(1-\lambda) \cdot 2W
\end{aligned}$$

Similarly, we can rescale by -W, such that all weights are ≤ 0 and in total we arrive at:

$$\begin{aligned}
& \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k})) + n(1-\lambda) \cdot 2W \\
& \geq v(v_0) \geq \\
& \frac{1}{l} \sum_{j=0}^{l-1} w((v_{j+k}, v_{j+1+k})) - n(1-\lambda) \cdot 2W
\end{aligned}$$

From 3.1.4 we know that the minimum distance between two valid values of a MPG is $\frac{1}{n(n-1)}$, therefore we want to set λ in such a way as to constrict the size of the just derived interval to at most:

$$|2n(1-\lambda) \cdot 2W| \leq \frac{1}{n(n-1)}$$

We get this precision with $\lambda = 1 - \frac{1}{4n^3W}$. If we reduce a MPG to a DPG with such λ, we can deduce the value for a given vertex of the original MPG by taking the corresponding value v in the DPG and rounding to the unique rational number, with denominator at most n, in the closed interval $v \pm \frac{1}{2n(n-1)}$.

3.2.3. MPGs to EGs

The reduction from MPGs to EGs is the one described in [BCD+11].
If we play an EG on A, w of a MPG, any vertex for which the value is $v_{\sigma,\tau}(v) \neq \infty$ is equal to a vertex in the MPG whose value is $v_{\sigma,\tau}(v) \geq 0$, since the player can force a terminal cycle whose edge-weights sum up to ≥ 0. This effectively splits the vertices of the graph into two regions for which we know that the value is ≥ 0 for one region and < 0 for the other. We can use this dichotomy of spliting the MPG into two areas together with the fact that we can rescale the edge-weights of MPGs. By solving these rescaled games as EGs we can further and further restrict the possible values of any given vertex in the MPG, until only one possible solution remains. As laid out in 3.1.4, the value of the vertices of a MPG are all in the set

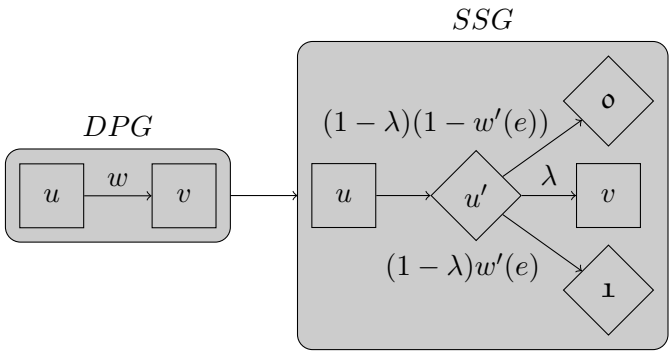
$$S = \left\{ \frac{p}{q} \in \mathbb{Q} \mid q \in [1, |V|] \subset \mathbb{N} \wedge p \in [-q \cdot W, q \cdot W] \subset \mathbb{Z} \right\}.$$

In particular, it is finite and after at most $log_2(|V| \cdot W)$ dichotomies, only one rational number, the value of the vertex, remains.
The way we rescale the MPG is by taking the interval of the range of possible values that are left, starting at S, and then split interval in about half by taking next lowest rational number $\frac{p}{q} \in S$ to the middle of the interval. We then rescale the edge-weights of the MPG, w, for the next dichotomy by $(w \cdot q) - p$ and solve it as an EG to split the vertices of the MPG into two regions with value $\geq \frac{p}{q}$ and $< \frac{p}{q}$.

To avoid having to solve the resulting EGs for the entire graph for any subdivision of S, we can subdivide the graph alongside the possible interval of values. In the MPG, for a given set of strategies σ, τ, any move will land on a vertex with the same value as the previous since the play ultimately always lands in the same terminal cycle with the same mean weight. This means that, for a fixed set of strategies, vertices with different values will necessarily be disconnected if one removes the edges that aren't in play under σ, τ and the values of all vertices stay the same if those edges are removed.
What this allows us to do is that after every dichotomy wrt. the values, where we split the set of vertices into three sets, one with value $> \nu$, one with value $< \nu$ and one with known value ν, we can restrict subsequent dichotomies to disjoint subsets of the graph in addition to halving the interval of possible values, since we know that vertices with different value, as shown to be the case by the dichotomy, have to be disconnected if unused edges are removed. This means subsequent solving via EGs happens on smaller graphs, making it faster.

3.2.4. DPGs to SSGs

The reduction from DPGs to SSGs is the one described in [ZP96].
The values of DPGs range from $-W$ to W whereas in SSGs they range from 0 to 1. We want to rescale our edge-weights in such a way that the values map to the new range while simultaneously maintaining their weight relative to one another.
We do so by rescaling the weights of the DPG to $w' = \frac{w+W}{2W}$ in the SSG. This maintains their relation to each other and by extension the relation between values of resulting plays and thus also the optimal strategies. After solving for the values of the SSG we can get the equivalent value in the DPG by simply reversing this rescaling.
For SSGs, unlike for DPGs, there is no "stack of previous passed edge-weights" to calculate the value off of, so we need to emulate such behaviour.
For any given $e = (u, v) \in E$ we can say that, after translation to an SSG, u has probability of λ to reach v. This means that, whatever value v has, it will be proportionally probabilistically represented in the value of u if e is played. Since the decision to play e is made by a player, but the probability to reach v from u is exactly that, a probability, and thus necessarily originates from a *Random* vertex, we have to insert an additional Random vertex, u', for each edge of the original DPG.
The equivalent aspect of the weight of the edge in the DPG, $w(e)$, also needs to be mirrored in the SSG. We can do so by giving the remaining probability $1 - \lambda$ the corresponding chance to reach $\mathfrak{o}$ and $\mathfrak{1}$. Using the previously rescaled edge-weights w' we can easily achieve this by setting $p(u')(\mathfrak{o}) = (1-\lambda) \cdot (1 - w'(e))$ and $p(u')(\mathfrak{1}) = (1-\lambda) \cdot w'(e)$ respectively. Effectively we are replacing every edge with the following construct:



Random vertices then have value $v(u') = \lambda \cdot v(v) + (1 - \lambda) \cdot w'(e)$ which exactly equal to the value of u in the DPG with rescaled edge-weights given that e is chosen as the outgoing edge. Since all outgoing edges will be replaced by this construct, the value of u is equal in the rescaled DPG and the SSG regardless of the strategies played. To translate back the optimal strategies generated on SSG one simply has to translate back every (u, u') that is part of a strategy in SSG to it's equivalent (u, v) in the DPG.

4. Solutions and Reductions in Practice

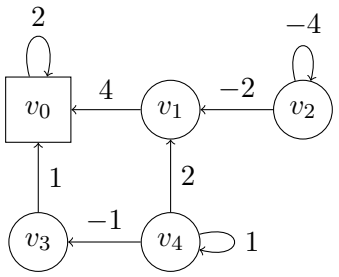
While the theoretical descriptions of the algorithms do a good job of explaining the idea of how the approaches work or ought to work, they may leave some room for interpretation about how exactly those should be implemented. We want to explain in detail how specifically we implemented them and what accommodations we made for the sake of performance.

All the games have, at their core, some directed graph as a key component. In addition to that there are some recurring themes that are exhibited by some or all of the games: The vertices of a graph belong to a player, the edges may have a weight or probability attached and for PGs, the vertices themselves have a value attached.

The vertices don't inherently have any ordering, so we think of them as just a numbering through the size of the set of vertices. We will call the generic size of that set $n = |V|$ and use 0-indexing from here on out, i.e. the "first" vertex is v_0 and the last is v_{n-1}.

The most fundamental question is how we represent the graph structure programmatically. There's two obvious choices:
Two or more parallel lists of length $|E|$ representing the edges of the game with the first two representing source vertices and target vertices and further lists representing additional values associated with that edge, e.g. edge-weights in MPGs *or*
An incidence matrix of size $n \times n$, The values of the matrix being the value that is associated with that edge, with a special value signaling that the edge doesn't exist and a simple boolean signaling doesn't/does exist for PGs and SSGs.
The first method has the advantage of using less memory, since we only use memory for edges that actually exist. The second method has the advantage that edges that have the same source or target vertex are memory-aligned, thus easy and fast to work with, as they are the same row or column in the matrix.
Given that a lot of operations include the semantic of "all edges from/to a certain vertex" and that even bigger graphs are manageable in memory size ($|V| = 16384$ with $int32$ means about 1GB) to the point where the runtime of algorithms applied will become a problem before memory size does we opt for the second variant. The *special value* that signals that the edge the matrix entry represents doesn't exist will generally, depending on context, be the minimum/maximum value the datatype can represent or NAN. If we represent matrices here, we will use $\times$ to represent these special values. Here we will pretend that operations on the matrices will effectively ignore these special values in all cases. Most prominently this means that their indices or values will never be returned by (arg)min or (arg)max operations. In practise there is some further logic involved in the handling of those special values. This doesn't add much complexity over how the implementations are explained here, but it does add varying degrees of runtime.

For example, the edges of a graph may be represented by a matrix as follows:



$$\begin{bmatrix} 2 & \times & \times & \times & \times \\ 4 & \times & \times & \times & \times \\ \times & -2 & -4 & \times & \times \\ 1 & \times & \times & \times & \times \\ \times & 2 & \times & -1 & 1 \end{bmatrix}$$

We shall simply call such matrices representing edges E. Similarly, the ownership of the vertices would be denoted as:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

We refer to such ownership matrices as O. 0 or $False$ denotes ownership to Player 0, 1 or $True$ to Player 1 and for SSGs we also have 2 for *Random* vertices.

For slicing vectors and matrices we will use the following notation: Given a matrix A, for which we only want to keep the edges for which the source and target vertices' indices are in $su = [0, 1, 4]$, we write $A[su, su]$. If a dimension should be unaffected by slicing, i.e. if we want to maintain all the entries from that dimensions we write ":". E.g. $A[:, su]$ means to keep all rows but only the columns with their index in su.

$$A = \begin{bmatrix} 2 & \times & \times & \times & \times \\ 4 & \times & \times & \times & \times \\ \times & -2 & -4 & \times & \times \\ 1 & \times & \times & \times & \times \\ \times & 2 & \times & -1 & 1 \end{bmatrix} \quad A[su, su] = \begin{bmatrix} 2 & \times & \times \\ 4 & \times & \times \\ \times & 2 & 1 \end{bmatrix} \quad A[:, su] = \begin{bmatrix} 2 & \times & \times \\ 4 & \times & \times \\ \times & -2 & \times \\ 1 & \times & \times \\ \times & 2 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & \times & -4 & 5 & 7 \end{bmatrix} \qquad B[su] = \begin{bmatrix} 2 & \times & 7 \end{bmatrix}$$

The code shown henceforth is a simplification of the actual code. Especially peculiarities of python or numpy, such as the notation of slicing matrices or the handling of aforementioned special values, are simplified without affect the logic of the underlying algorithm.

4.1. Solutions in Practice

4.1.1. PGs

Parity games additionally have priorities associated with it's vertices. We shall represent them similarly to how we represent the ownership of the vertices except their range is over $\mathbb{N}_0$ instead of a boolean and called P.

Zielonka's algorithm:

```
1  def zielonka(O, E, P):
2      if len(O) == 0:
3          return []
4      else:
5          max_prio = max(P)
6          player = max_prio % 2
7          A = find_attractor(player, O, E, P == max_prio[3])
8          z1 = zielonka(O[A^-1][4], E[A^-1,A^-1], P[A^-1])
9          if not any(z1 == player^-1):
10             return [player]*len(O)[5]
11         else:
12             losing = [False]*len(O)
13             losing[A^-1[z1 == ¬player]] = True
14             B = find_attractor(¬player, O, E, losing)
15             z2 = zielonka(owner[B^-1], edges[B^-1], priorities[B^-1])
16             ret = [¬player]*len(O)
17             ret[B^-1[z2 == player]] = player
```

L.2-3: The recursion in Zielonka's algorithm eventually hit it's bottom by reaching an empty graph. Since obviously there is no vertex to be won, we return an empty vector.
L.5-6: Each iteration starts with finding the attractor set of the highest priority. We determine such region and the associated player.
L.7: To find the attractor we need the player for whom the attractor set works and the set of initial vertices that actually posses the priority in question. In return we receive the set of vertices from which the player can and the opponent cannot avoid reaching the set of highest priority effectively making it an *attractor*.
L.8: The first recursive call.
L.9-10: If the first recursive call returns that the game contains no winning region for the opponent then we can conclude that the current game is entirely won by the current player.
L.12-14: If there is a winning region for the opponent in the first recursive call on the game sans the first attractor, that means that the opponent can win in the attractor of such winning region since they can avoid A entirely.
L.15: We know that B is a winning region for the opponent. We continue to recursively find winning regions for subgames without B, until we reach an empty game by courtesy of L.2-3.
L.16-17: Such winning regions then get forwarded back through the recursion, together with B, as a win or lose of the respective player.

The algorithm used to find the attractors:

```
1  def find_attractor(player, O, E, Nh):
2      if len(Nh) == 0:
3          return []
4      us = owner == player
5      them = owner != player
6      atr = Nh
7      while True:
```

[3] As in the list of indices of P where the value is equal to max_prio
[4] A^{-1} as in the inverse of A with respect to the preexisting indexing of O
[5] A vector of length len(O) only containing the value "player"

```
8           old = atr
9           us_add = any[6](edges[us,atr], axis=1)
10          them_add = any(edges[them, atr^-1], axis=1)^-1
11          atr[us[us_add]] = True
12          atr[them[them_add]] = True
13          if all(atr == old):
14              return atr
```

L.1: The algorithm is of course dependent on the graph (O, E) but also on the player whose attractor we want to find, since one player want to avoid it, i.e. have edges that *don't* lead into the attractor and the other one wants to reach it. It also needs an initial set of vertices, Nh, which the attractor actually attracts to.
L.2-3: Same as with Zielonka's algorithm, an empty set will of course have no attractor.
L.4-5: We precompute once the indices of the vertices that belong to the player of the opponent respectively.
L.6-8: We iterate over the original set Nh until we hit an iteration that didn't lead to the attractor growing in size.
L.9: We gather all the indices of vertices belonging to the player that have edges into the current iteration of the attractor set.
L.10: We gather all the indices of vertices belonging to the opponent that have no edges that don't go into the current iteration of the attractor set. Note the double negation.
L.11-12: The vertices we gathered in L.9-10 get added to the attractor set.
L.13-14: If L.11-12 didn't lead to a change in the attractor set then we return the current, final, attractor set.

Given any algorithm to find values, we can use it to also find strategies in a costly manner. We show this implementation here exemplary for Zielonka's algorithm.

Finding strategies via Zielonka's algorithm:

```
1  def solve_strat_zielonka(self[7]):
2      z = self.solve_value_zielonka()
3      ret = [-1]*len(self.O)
4      edges = self.E
5      for i,v in enumerate[8](edges):
6          w = v == True
7          while True:
8              cl = ceil(len(w) / 2)
9              one, two = w[:cl], w[cl:]
10             e = edges
11             e[i] = False
12             e[i, one] = True
13             x = ParityGame(self.O, e,
                   self.P).solve_value_zielonka()
14             if all(x == z):
15                 if len(one) == 1:
16                     ret[i] = one[0]
17                     break
```

[6] Which rows contain any *True* value; Effectively: All vertices of *player* that have edges into the attractor; see https://numpy.org/doc/stable/reference/generated/numpy.any.html
[7] *self* refers to a game object with the fields O, E and P, it's ownership, edge and priority vector/matrix
[8] enumerate iterates over an enumerable and provides the index (i) together with the element (v)

```
18                  else:
19                      w = one
20              else:
21                  w = two
22          edges[i] = [False]*len(self.O)
23          edges[i, ret[i]] = True
24      return ret
```

L.2: z are the values of the vertices which we compare to while removing all but one outgoing edge for each vertex.
L.3: This holds the strategy, we start with -1 i.e. no strategy for each vertex.
L.4: We create a copy of the edges from which we progressively delete edge.
L.5-7: For each vertex we iteration until we find an edge that is part of an optimal strategy.
L.8-12: We part the outgoing edges of our current vertex into two sets and create an edge matrix that only contains the edges of the first set.
L.13-14: We solve for the values for our game with restricted edges and check if the values stay the same. If they do, clearly the removed edges are not necessary for an optimal strategy.
L.15-19: If the values did stay the same then the edge part of optimal strategies are in the "one" half of the edges. If there is only one edge, this is *the* edge for an optimal strategy.
L.20-21: If the values change, the edges for optimal strategies must have been in the "two" half.
L.22-23: After we choose a successor vertex for a vertex we need to make the corresponding edge mandatory, since optimal strategies aren't necessarily unique but may depend on the decision we made for the current vertex. Otherwise we may choose an edge for subsequent vertices that are part of *some* optimal strategy, but not an optimal strategy that is compatible with the choice we already made here.

4.1.2. MPGs

Mean Payoff games are solely descibed by the ownership vector, O, and the edge matrix, E that also contains the edge-weights.

Zwick and Paterson's algorithm:

```
1  def solve_value_zwick_paterson(O, E):
2      W = max(abs(E))
3      k = 4 * (len(O) ** 3) * W
4      v = [0]*len(O)
5      for _ in range(k):
6          edges_weight = edges + v
7          v[O] = min(edges_weight, axis=1)
8          v[¬O] = max(edges_weight, axis=1)
9      v = v / k
10     return trunc(len(O), v)
```

L.2-3: First we determine the iteration depth required by the algorithm.
L.4-5: We instantiate with 0 and then iterate as often as the algorithm requires.

L.6: For every iteration we generate the k-th iteration of all possible vertices and outgoing edges combinations.
L.7-8: For vertices belonging to *Min* we take the vertex-edge combination that leads to the lower value, for *Max* the one that leads to the highest.
L.9: Since the tabulation is cumulative we need to rescale once at the end.
L.10: Since this approach only gives approximations the algorithm prescribes rounding to the appropriate rational number at the end.

The algorithm used to truncate to the appropriate rational number:

```
1  def trunc(d, v):
2      lower = v - (1 / (2 * d * (d - 1)))
3      upper = v + (1 / (2 * d * (d - 1)))
4      for v_n in range(d):
5          for denominator in range(1, d + 1):
6              f = floor(lower[v_n] * denominator)
7              c = ceil(upper[v_n] * denominator)
8              num = range(f, c + 1) / denominator
9              if any((lower[v_n] < num) & (upper[v_n] > num)):
10                 v[v_n] = num[((lower[v_n] < num) & (upper[v_n] >
                       num))][0]
11                 break
12     return v
```

L.1: Given a maximum denominator and a vector of values, we truncate to the nearest valid rational number.
L.2-3: From the algorithm we know that this is the interval in which the actual rational values have to be.
L.4-7: For every entry in the vector and every possible denominator we determine a lower and an upper bound for possible numerators.
L.8: We create a range of possible rational numbers that correspond to the rational numbers that possibly lie in the interval.
L.9-11: If we find a number that fits into the interval it must be the rational number we are looking for and we proceed with the next number.

Finding strategies via Zwick and Paterson's algorithm: Similar to finding Strategies for PGs under Zielonka's algorithms we can perform the same kind of search here. Since it's perfectly analogous, with the exception of solving for the values of a MPG, we won't repeat it here.

4.1.3. EGs

Similarly to Mean Payoff games, Energy games are solely descibed by the ownership vector and the edge matrix.

BCDGR's algorithm:

```
def solve_both_bcdgr(O, E):
    l = [False]*len(O)
    for v in (O == False):
        if all(E[v] < 0):
            l[v] = True
    for v in (O == True):
        if any(E[v] < 0):
            l[v] = True
    f = [0]*len(O)
    cnt = [0]*len(O)
    for v in (O == False):
        for w in E[v]:
            if leq[9](minus[10](f[w], E[v,w]), f[v]):
                cnt[v] += 1
    while any(l):
        v = argmax(l)
        l[v] = False
        old = f[v]
        if not O[v]:
            f[v], _ = min*[11]([minus(f[w], E[v, w]) for w in E[v]])
        else:
            f[v], _ = max*([minus(f[w], E[v, w]) for w in E[v]])
        if not O[v]:
            cnt[v] = 0
            for w in E[v]:
                if leq(minus(f[w], E[v, w]), f[v]):
                    cnt[v] += 1
        for u in [u for u in E[:, v] if not leq(minus(f[v], E[u,
            v]), f[u])]:
            if not O[u]:
                if leq(minus(old, E[u, v]), f[u]):
                    cnt[u] -= 1
                if cnt[u] <= 0:
                    l[u] = True
            else:
                l[u] = True
    return_strat = [-1]*len(E)
    for i in range(len(O)):
        if not O[i]:
            _, cand = min*([minus(f[w], E[i, w]) for w in E[i])
            return_strat[i] = E[i,cand]
    return f, return_strat
```

[9]The modified kind of $\preceq$ as laid out in the formal description of the algorithm
[10]The modified kind of $\ominus$ as laid out in the formal description of the algorithm
[11]Gives the value and the index of the minimum as a tuple

L.2: l is the list of vertices that potentially have an outdated progress measure.
L.3-8: We initialize l with the vertices that can't have a progress measure of 0.
L.9-14: f is the current progress measure and cnt is count as laid out in the formal description fo the algorithm and it's initialization.
L.15-18: We go through the list of vertices that we have to update, keeping a copy of f[v].
L.19-22: We update the current progress measure of the vertex depending on whose vertex it is with the minimum/maximum progress measure possible, depending on the successor vertices.
L.23-27: We update the count of vertices that depend on the current vertex v.
L.28: For reach vertex in pre(v) that is now outdated:
L.29-33: If vertex if of *Charging* decrement count and add vertex to l
L.34-35: If vertex if of *Depleting* add vertex to l
L.36-40: Read out optimal strategy for *Charging* based off of the lowest cost successor.

Strategy Iteration from below:

```
1  def solve_both_strat_iter_below(self):
2      p1 = self.O == True
3      nW = len(self.O) * max(abs(self.E))
4      E_p1 = self.E[p1, p1]
5      cycle_nodes, neg_strat = find_all_negative_cycle_nodes(E_p1)
6      O = self.O + [False]
7      E = self.E
8      E[len(O)+1] = ×
9      E[:len(O)+1] = ×
10     E[len(O)+1, len(O)+1] = 0
11     E[self.O, len(O)+1] = -2 * nW
12     restriction = p1[cycle_nodes]^-1
13     O = O[restriction]
14     E = E[restriction, restriction]
15     strat = [-1]*len(O)
16     strat[O] = apply_along_axis(lambda v: random.choice(v), 1,
           E[O])
17     while True:
18         strat_hist = strat.copy()
19         f = [0]*len(O)
20         while True:
21             old = f.copy()
22             E_weight = maximum(minimum(f - E, 3 * nW), 0)
23             f[O] = E_weight[O, strat]
24             f[¬O] = min(E_weight[¬O], axis=1)
25             if f == old:
26                 break
27         g = E_weight[:, strat] < E_weight[:, argmax(E_weight,
               axis=1)]
28         strat[O&g] = argmax(E_weight, 1)[O&g]
29         if strat_hist==strat:
30             break
31     final_f = [-1]*len(self.O)
32     f[¬(f < nW)] = -1
33     final_f[p1[cycle_nodes]^-1] = f
34     return_strat = [-1]*len(self.E)
35     return_strat[p1[cycle_nodes]] = p1[neg_strat]
```

```
36      return_strat[p1[cycle_nodes]^-1][strat!=-1] =
            restriction[strat][strat!=-1]
37      return final_f, return_strat
```

L.2-14: First we add the $s \in V_0$ vertex and restrict the game to a version without negative cycles that are in total control of *Depleting*.
L.15-16: We start of with a random strategy. For every vertex of *Depleting* we choose a random successor.
L.17-18,29-30: We iterate until our strategy doesn't change anymore, i.e. we have found an optimal stratey for *Depleting*.
L.19-26: We solve a game based on the fixed strategy of *Depleting* via Kleene Iteration.
L.27-28: We check for which vertices we have an actual improvement and change the strategy accordingly.
L.31-37: Return the values and strategy after some scaling back wrt the negative cycles removed previously. Set values in excess of nW to infinite.

Strategy Iteration from above:

```
1   def solve_both_strat_iter_above(self):
2       p1 = self.O == True
3       nW = len(self.O) * max(abs(self.E))
4       E_p1 = self.E[p1, p1]
5       cycle_nodes, neg_strat = find_all_negative_cycle_nodes(E_p1)
6       O = self.O + [False]
7       E = self.E
8       E[len(O)+1] = ×
9       E[:len(O)+1] = ×
10      E[len(O)+1, len(O)+1] = 0
11      E[self.O, len(O)+1] = -2 * nW
12      restriction = p1[cycle_nodes]^-1
13      O = O[restriction]
14      E = E[restriction, restriction]
15      strat = [-1]*len(O)
16      strat[¬O] = [len(O)-1]*len(O)
17      while True:
18          strat_hist = strat.copy()
19          solver = pywraplp.Solver.CreateSolver("GLOP")
20          v = [solver.NumVar(float(0), float(3 * nW), str(x)) for x
                in range(len(O))]
21          for s, p in enumerate(O[:-1]):
22              if not p:
23                  solver.Add(v[s] >= (v[strat[s]] - float(E[s,
                        strat[s]])))
24              else:
25                  for t in where(E[s] != mini)[0]:
26                      solver.Add(v[s] >= (v[t] - float(E[s, t])))
27          solver.Add(v[-1] == float(0))
28          obj_func = v[0]
29          for v_n in v[1:]:
30              obj_func += v_n
31          solver.Minimize(obj_func)
32          status = solver.Solve()
33          v = [v_n.solution_value() for v_n in v]
34          while True:
```

```
35              strat[¬O] = argmin(clip(v - E, 0, None), 1)
36              if any(strat != strat_hist):
37                  break
38              v_ = [True]*len(v)
39              while True:
40                  v_h = v_.copy()
41                  c1 = v != 0
42                  c2 = v.transpose() == clip(v - E, 0, 3 * nW)
43                  c3 = 0 < v
44                  c4 = 0 < clip(v - E, 0, None)
45                  c5 = clip(v - E, 0, None) <= (3 * nW)
46                  c6 = v_
47                  V_p0 = c1 & any(c2 & c3 & c4 & c5 & c6, 1)
48                  V_p1 = c1 & all((¬c2 or (c3 & c4 & c5 & c6)), 1)
49                  v_[O] = V_p1
50                  v_[¬O] = V_p0
51                  if all(v_ == v_h):
52                      break
53              if not any(v_):
54                  final_f = [-1]*len(self.O)
55                  v[v < nW] =  v[v < nW]
56                  final_f[p1[cycle_nodes]^-1] = v
57                  return_strat = [-1] * len(self.E)
58                  return_strat[p1[cycle_nodes]] = p1[neg_strat]
59                  return_strat[p1[cycle_nodes]^-1][strat != -1]
                        restriction[strat]
60                  return_strat[return_strat == len(self.O)] =
                        apply_along_axis(lambda v:
                        random.choice(v),1,self.E)
61                  return final_f, return_strat
62              else:
63                  v[v_] -= 1
```

L.2-14: We again add the $s \in V_0$ vertex and restrict the game to a version without negative cycles that are in total control of *Depleting*.
L.15-16: The starting strategy for *Charging* is for every vertex to go to the special sink vertex.
L.18-33: We solve a linear program based on the current strategy for *Charging*.
L.35-37: The find an optimal counter strategy for *Depleting*. If the strategy changed we iterate to the next linear program, if it didn't we check if our fix-point is the least.
L.38-52: We determine $\bar{V}$.
L.53-61: If $\bar{V}$ is empty we have our least fix-point. We return the values and strategy minus reverting the preprocessing as we did for the Strategy Iteration from below.
L.62-63: If $\bar{V}$ isn't empty we decrement it's contents and try to find a another counter-strategy.

The algorithm used to find all negative cycles:

Since Bellman-Ford only gives at least some negative cycles but not all, we need to iterate to find all such cycles.

```
1  def find_all_negative_cycle_nodes(E):
2      neg_strat = [-1]*len(E)
3      cycle_nodes = [False]*len(E)
4      while True:
5          restriction = ¬cycle_nodes
6          ret, ret_strat = find_negative_cycle_nodes(E[restriction,
               restriction])
7          if len(ret) == 0:
8              break
9          else:
10             neg_strat[restriction[ret]] = restriction[ret_strat]
11             cycle_nodes[restriction[ret]] = True
12     if any(cycle_nodes):
13         while True:
14             old = cycle_nodes.copy()
15             adds = any(E[:, cycle_nodes], axis=1)
16             adds[cycle_nodes] = False
17             neg_strat[adds] = cycle_nodes[argmax(E[adds,
                   cycle_nodes],axis=1)]
18             cycle_nodes = (cycle_nodes or adds)
19             if all(cycle_nodes == old):
20                 break
21     return cycle_nodes, neg_strat[cycle_nodes]
```

L.2-11: We start with no knowledge of any vertices being part of any negative cycle and then progressively add those that we find.
L.6: Any search may give us cycles and the strategy that would be deployed to stay in such circle, which is what *Depleting* wants to do.
L.12-21: Once we have found all negative cycles, we also need to include all all vertices that inevitably lead into those cycles.

The Bellman-Ford algorithm derivate used to find negative cycles:

We add a special vertex that connects to every other vertex. If after $|V|$ steps, i.e. after we necessarily have entered a loop somewhere, we can still lower the cost to reach a certain vertex, we must have discovered a negative cycle.

```
1  def find_negative_cycle_nodes(E):
2      edges_src = E
3      edges_src[:len(E)] = 0
4      edges_src[len(E)] = ×
5      dist = [∞]*len(edges_src)
6      dist[len(E)] = 0
7      pred = [-1]*len(edges_src)
8      for i in range(1, len(edges_src) + 1):
9          src_is_pred = (dist != ∞)*len(edges_src).transpose()
10         new_dist = dist.transpose() + edges_src
11         shorter = new_dist < dist
12         valids = edges_src & src_is_pred
13         exists_shorter = any(valids & shorter, axis=0)
14         shorter_idx = argmin(new_dist if valids else ∞, axis=0)
15         pred = shorter_idx if exists_shorter else pred
16         dist[exists_shorter] = new_dist[shorter_idx]
17     cycle_nodes_t = [False]*len(E)
18     for n in (exists_shorter == True):
19         cycle_nodes = [False]*len(E)
20         s = [n]
21         for x in range(len(edges_src) - 1):
22             if pred[n] == s[0]:
23                 cycle_nodes[s] = True
24                 break
25             else:
26                 s += pred[n]
27                 n = pred[n]
28         cycle_nodes_t = (cycle_nodes_t or cycle_nodes)
29     return pred[ret], cycle_nodes_t
```

L.2-4: We start by adding a special vertex that has an edge to every other vertex.
L.5-7: We start of by knowing no path towards any other vertex.
L.8-16: We iterate until we surpass the size of the vertex, so that every other vertex will have been able to be reached. For every iteration we check if we can discover a new, shorter path to any of the vertices and note the distance and predecessor.
L.17-29: The previous step only shows us vertices that are part of a negative cycle. To find *all* the vertices that are part of such cycle we need to unwind them.

4.1.4. DPGs

DPGs get the additional field D for their discount. *player* describes from whose perspective we want to find the values and strategy, with *False* meaning *Max*.

Strategy Iteration:

```
def solve_both_strat_iter(self, player):
    strat = [-1]*len(self.O)
    strat[player==self.O] = apply_along_axis(lambda x:
       random.choice(x), 1, self.E)
    while True:
        strat_hist = strat.copy()
        W = max(abs(self.E))
        solver = pywraplp.Solver.CreateSolver("GLOP")
        v = [solver.NumVar(float(-W), float(W), str(x)) for x in
           range(len(self.O))]
        if not player:
            for s, p in enumerate(self.O):
                if not p:
                    solver.Add(v[s] == (1 - float(self.D)) *
                       float(self.E[s, strat[s]]) + float(self.D)
                       * v[strat[s]])
                else:
                    for t in self.E[s]:
                        solver.Add(v[s] <= (1 - float(self.D)) *
                           float(self.E[s, t]) + float(self.D) *
                           v[t])
        else:
            for s, p in enumerate(self.O):
                if p:
                    solver.Add(v[s] == (1 - float(self.D)) *
                       float(self.E[s, strat[s]]) + float(self.D)
                       * v[strat[s]])
                else:
                    for t in self.E[s]:
                        solver.Add(v[s] >= (1 - float(self.D)) *
                           float(self.E[s, t]) + float(self.D) *
                           v[t])
        obj_func = v[0]
        for v_n in v[1:]:
            obj_func += v_n
        if not player:
            solver.Maximize(obj_func)
        else:
            solver.Minimize(obj_func)
        status = solver.Solve()
        if not player:
            strat[¬self.O] = argmax(((1 - self.D) * self.E) +
               (self.D * ([v_n.solution_value() for v_n in
               v])),axis=1)
        else:
            strat[self.O] = argmin(((1 - self.D) * self.E) +
               (self.D * ([v_n.solution_value() for v_n in
               v])),axis=1)
```

```
35          if all(strat_hist==strat):
36              break
37      return [v_n.solution_value() for v_n in v], strat
```

L.2-3: We start of with a random strategy. For every vertex of *player* we choose a random successor.
L.4-5,35-37: We iterate until the strategy doesn't change any more and return the current strategy and values.
L.6-30: We solve the Linear Program for the given strategy to find the values of the game under optimal play from the opponent.
L.9-15,16-22: We in particular distinguish whether we are minimizing or maximizing (the inequality on L15 & 22) for the vertices of the opponent depending on whose perspective we are searching from.
L.31-34: Depending on the perspective we change strategy to one that maximizes or minimized the value based on the current values.

Kleene Iteration:

```
1  def solve_both_kleene(O, E, D):
2      W = max(abs(E, 0))
3      cur = [-W]*len(O)
4      while True:
5          old = cur.copy()
6          E_weight = ((1 - D) * E) + D * cur
7          cur = [0]*len(O)
8          cur[¬O] = max(edges_weight, axis=1)
9          cur[O] = min(edges_weight, axis=1)
10         if max(abs(cur - old)) < 1e-14:
11             break
12     strat = [-1]*len(O)
13     strat[¬O] = argmax(((1 - D) * E) + D * cur, axis=1)
14     strat[O] = argmin(((1 - D) * E) + D * cur, axis=1)
15     return cur, strat
```

L.2-3: We start our iteration with the bottom element.
L.4-5,10-11: We iteration until the the maximum change in any value falls below a certain threshold. The threshold can in theory be 0 which would be equal to iterating until the (theoretical) change in values is smaller than the precision floats numerically can handle.
L.6: For every iteration we generate the entire matrix of possibilites of outgoing edges for every vertex and the accompanying value.
L.7-9: We continue with the maximum/minimum value for the respective player's vertices.
L.12-14: Given the values of the game we can also read out optimal strategies based off of the successor vertices needed to reach the already calculated values.

4.1.5. SSGs

For SSGs we have an additional matrix, AVG, that gives us the transition probability from any *Random* vertex to it's successor vertices.

Strategy Iteration:

```
1  def solve_both_strat_iter(self, player):
2      strat = [-1]*len(self.O)
3      strat[self.O == player] = apply_along_axis(lambda x:
           random.choice(x), 1, self.E)
4      while True:
5          strat_hist = strat.copy()
6          solver = pywraplp.Solver.CreateSolver("GLOP")
7          v = ([solver.NumVar(float(0), float(1), str(x)) for x in
               range(len(self.O))] +
8               [solver.NumVar(float(0), float(0),
                   str(str(len(self.O) + 1)))] +
9               [solver.NumVar(float(1), float(1),
                   str(str(len(self.O) + 2)))])
10         for s, p in enumerate(self.O):
11             if p == player:
12                 solver.Add(v[s] == v[strat[s]])
13             else:
14                 if p == 0:
15                     for t in self.E[s]:
16                         solver.Add(v[s] >= v[t])
17                 elif p == 1:
18                     for t in self.E[s]:
19                         solver.Add(v[s] <= v[t])
20                 else:
21                     val = 0
22                     for t in self.E[s]:
23                         val += v[t] * self.AVG[s, t]
24                     solver.Add(v[s] == val)
25         obj_func = v[0]
26         for v_n in v[1:]:
27             obj_func += v_n
28         if not player:
29             solver.Maximize(obj_func)
30         else:
31             solver.Minimize(obj_func)
32         status = solver.Solve()
33         if not player:
34             strat[¬self.O] = argmax(self.E *
                   [v_n.solution_value() for v_n in v]), axis=1)
35         else:
36             strat[self.O] = argmin(self.E * [v_n.solution_value()
                   for v_n in v]), axis=1)
37         if all(strat_hist == strat):
38             break
39     return [v_n.solution_value() for v_n in v], strat
```

L.2-3: We start of with a random strategy. For every vertex of *player* we choose a random successor.

L.4-5,37-39: We iterate until the strategy doesn't change any more and return the current strategy and values.

L.6-32: We solve the Linear Program for the given strategy to find the values of the game under optimal play from the opponent.

L.21-24: Unlike the degrees of freedom the opponent has in choosing a successor vertex which results in a half-space in the Linear Program, *Random* just has an equality relationship in dependance of it's successors and transition probabilities.
L.33-36: Depending on the perspective we change strategy to one that maximizes or minimized the value based on the current values.

Kleene Iteration:

```
1  def solve_both_kleene(O, E, AVG):
2      cur = [0]*len(O) + [0] + [1]
3      while True:
4          old = cur.copy()
5          edges_weight = E*cur
6          cur[O == 2] = sum(AVG * cur, 1)
7          cur[O == 1] = min(edges_weight, axis=1)
8          cur[O == 0] = max(edges_weight, axis=1)
9          max_err = max(abs(cur - old))
10         if max_err < 1e-14:
11             break
12     strat = [-1]*len(O)
13     strat[O==0] = argmax(cur, axis=1)
14     strat[O==1] = argmin(cur, axis=1)
15     return cur, strat
```

L.2: We start our iteration with the bottom element.
L.3-4,10-11: We iteration until the the maximum change in any value falls below a certain threshold. The threshold can in theory be 0 which would be equal to iterating until the (theoretical) change in values is smaller than the precision floats numerically can handle.
L.6: For every iteration we generate the entire matrix of possibilites of outgoing edges for every vertex and the accompanying value.
L.7-9: We continue with the maximum/minimum value for the respective player's vertices.
L.12-14: Given the values of the game we can also read out optimal strategies based off of the successor vertices needed to reach the already calculated values. For *Random* we simply leave these as -1 as to not disturb the indexing.

4.2. Reductions in Practice

4.2.1. PGs to MPGs

As already shown in 3.2.1, the original algorithm in [Jur98] stipulates the weights to be rewritten as $w((u,v)) = (-n)^{p(u)}$ which is wasteful wrt. the size of the range of weights in the resulting MPG.

Reducing a PG to a MPG:

```
1  def to_mpg(self):
2      prios_k = argsort(self.P)
3      prios = self.P[prios_k]
4      prios_even = prios % 2 == 0
5      weight = [1]*len(self.O)
6      for i, pr in enumerate(prios):
7          if pr % 2 == 0:
8              weight[i] = sum(weight[:i][¬prios_even[:i]])
9          else:
10             weight[i] = sum(weight[:i][prios_even[:i]]) + 1
11     weight[¬prios_even] = -weight[¬prios_even]
12     y = [0]*len(self.O)
13     y[prios_k] = weight
14     E = [[×]*len(self.O)]*len(self.O)
15     E = y*len(self.O)
16     return MeanPayoffGame(self.O, E)
```

L.2-3: We produce a sorted copy, *prios* of the priority list along with a list of indices that sort the original priority list.
L.4: We make a boolean list can masks *prios* wrt parity.
L.5: We intialize the list specifically with one, since if we start with an odd number, the following even number would not actually be greater.
L.6-10: We go through the list one by one. It the priority is even, it's value is the sum of all prior odd numbers. Since a value of ≥ 0 is winning for *Max* in the MPG, reaching equality is sufficient. For odd priorities, at sum all the prior even numbers plus one, to ensure win in a MPG that would otherwise have a value of zero.
L.11: The calculated weights so far are absolutes. We negate the priorities associated with *Odd* to get the desired effect of the reduction.
L.13: We undo the prior reindexing to sort the list.
L.14-15: We scale the resulting list to a matrix to correspond with the edges in the MPG in a way that is congruent with the incidence matrix of the PG, i.e. we translate over the $\times$.

4.2.2. MPGs to EGs

There's two significant differences to [BCD+11] compared to how we implemented the idea here.
Firstly, in the original the algorithm solves four energy games at every "level of recursion". This means that two possible values can be "detected" for vertices at every level but also means that the runtime is significantly larger for every level. To depress the runtime we only run two energy games per level. This means every level is faster but also only detects for one value and we might have to iterate deeper but since deeper iteration runs on subgames it is faster.
Secondly, we also extract the strategies from the subgames. Out of the original proof of the algorithm one can relatively straight forward reason that strategies on subgames are strategies in the total game.

Reducing the value & strategy problem in a MPG to a EG:

```
1  def solve_both_eg_alg(O, E, lower, upper):
2      r = range(1, len(O) + 1)
3      l = r * (lower + upper) / 2
4      l1 = floor(l)
5      l2 = ceil(l)
6      ar1 = argmax(l1 / r)
7      ar2 = argmin(l2 / r)
8      a1 = [l1[ar1], r[ar1]]
9      a2 = [l2[ar2], r[ar2]]
10     e1 = (a1[1] * E) - a1[0]
11     e2 = (-a1[1] * E) + a1[0]
12     f1, s1 = EnergyGame(O, e1).solve_both()
13     f2, s2 = EnergyGame(¬O, e2).solve_both()
14     v = [-1]*len(O)
15     s = [-1]*len(O)
16     v[(f1 != -1) & (f2 != -1)] = a1[0] / a1[1]
17     s[(f1 != -1) & (f2 != -1)] = s1 if s1 != -1 else s2
18     v1 = (f1 == -1)
19     v2 = (f2 == -1)
20     if len(v1) != 0:
21         v[v1], st = solve_both_eg_alg(O[v1], E[v1, v2)], lower,
               a1[0] / a1[1])
22         s[v1] = v1[st]
23     if len(v2) != 0:
24         v[v2], st = solve_both_eg_alg(O[v2], E[v2, v2)], a2[0] /
               a2[1], upper)
25         s[v2] = v2[st]
26     return v, s
```

L.2-9: First we calculate the two rational numbers closest to middle of our interval. Since actually need the numerator and denominator, we save each rational number as a tuple of those two.
L.10-13: We calculate the two reweighted edge-matrices based on the rational number next lowest to the middle of the interval and solve them via some EG algorithm that solves for both value and optimal strategy.
L.16: If the value of a vertex if finite for both games then the rational number we

rescaled for must be the value of the corresponding vertex in the original MPG.
L.17: The strategy of that vertex can be read from one of the games, depending on whose vertex it is.
L.20-25: The remaining vertices get split into two recursive searches based on if their values was higher or lower than the rational number we used as a division for this level. Critically, for the "higher" search we use rational number that was *higher* than the middle of our current interval.

4.2.3. MPGs to DPGs

The reduction from MPGs to DPGs is trivial but for completeness sake we include it:

Reducing a PG to a MPG:

```
def to_dpg(self):
    W = max(abs(self.edges))
    discount = 1 - (1 / (4 * (len(self.O) ** 3) * W))
    return DiscountedPayoffGame(self.owner, self.edges, discount)
```

The owner vector and edge matrix stay exactly the same, the discount gets calculated off of the size of the graph and range of the edge-weights as described in the theory.

4.2.4. DPGs to SSGs

During the process of translating a DPG into an SSG edges get rewired. To be able to correctly align the vertices for the purpose of finding strategies for the DPG and for the purpose of scaling back the values to what they were previous to the translate this function also gives back *strat_map* and W with which the strategies and values and trivially be corrected wrt the translation.

Reducing a DPG to a SSG:

```
1  def to_ssg(self):
2      W = max(abs(self.E, 0))
3      E = (self.E + W) / (2 * W)
4      f3 = count_nonzero(self.E)[12]
5      vertices = len(self.O) + f3
6      ssg_E = ([False]*vertices)*(vertices + 2)
7      O = self.O + [2]*f3
8      AVG = ([0]*f3)*(vertices + 2)
9      strat_map = [0]*f3
10     for i, edge in enumerate(self.E):
11         ssg_E[edge[0], i + len(self.O)] = True
12         ssg_E[i + len(self.O), edge[1]] = True
13         strat_map[i] = edge[1]
14         ssg_E[i + len(self.O), -1] = True
15         ssg_E[i + len(self.O), -2] = True
16         AVG[i, edge[1]] = self.D
17         AVG[i, -2] = (1 - self.D) * (1 - (E[edge]))
18         AVG[i, -1] = (1 - self.D) * E[edge]
19     return SimpleStochasticGame(O, ssg_E, AVG, True), strat_map, W
```

L.2-3: We rescale the weights to fit in the [0,1] interval appropriate for the SSG.
L.4: We count the number of edges in the DPG. This the amount by which we need to increase the ownership vector, edge matrix and probability matrix.
L.5-9: We prepare the aforementioned vector and matrices as well as a vector for translating back the strategies formulated in the SSG.
L.10-18: For every edge in the DPG we add the equivalent construct in the SSG as well as it's implications in the *strat_map*.

[12] As in count the number of non-$\times$ edges

5. Implementation

6. Evaluation

Our goal was to compare the different algorithms used to solve problems, especially under reduction to other games, and to see how they compare. To that end we generated random games with multiple parameters and benchmarked all methods we had to solve the respective problems. For any combination of problem, parameters for generation and algorithm used to solve the problem we have $N = 100$ and the times shown repesent the average time of those 100 runs. For their generation, all games have a parameter $n = |V|$ that measures the size of the graph by means of the number of vertices and a parameter p which describes the probability that any possible edge $e \in (V \times V)$ is part of the generated game, meaning the expected out-degree of every vertex is $|V| \cdot p$. Since we require for all games that each vertex has outdegree ≥ 1, we consequently require that $\frac{1}{|V|} \leq p \leq 1$. We then assign each vertex one random edge and all other potential outgoing edges of that vertex have probability $\frac{(p \cdot |V|)-1}{|V|-1}$ of existing. p can be understood as a measure of how connected a graph is. In fact, towards the extreme ends, for $p = \frac{1}{|V|}$ and sufficiently large $|V|$ one would expect a disconnected graph on one end and for $p = 1$ a complete graph on the other end. Each vertex is randomly assigned to a player with equal probability, including the "Random" player in SSGs. Games may have additional parameters that are described seperately for each game. The parameters were chosen in a way that they intuitively represent extremes towards each end and then some reasonable middle ground. For all cases, the runtimes are either weakly dependent on p other parameters or they scale monotonously wrt. to those parameters. One can therefore think of intermediate parameterizations as some interpolation between the shown values. For readability we only highlight some algorithms here. Those that work without reduction to another game and any other that are of interest. The other ones are faded out into the background, with the full results in the appendix.
Some of the algorithms for solving for optimal strategies only return partial strategies, i.e. the strategy for only one of the players. If that is the case, we mark them $\square$ and $\bigcirc$ depending on whose strategy they can solve for. If they can solve for both but not at once, running the algorithm twice, once for each player, yields the entire strategy.

6.1. PGs

For Parity games, the additional parameter is w, which describes the range over which the priorities are handed out. Similar to the ownership of the vertices, the priority of vertices is randomly chosen from the possible range with equal chance for each. In theory as well as in this implementation, for the purpose of solving the problems one can rescale the range of the priorities to $\{0, 1, ...|V|\}$ or smaller without affecting the values or optimal strategies of the game. A higher w however decreases the overlap of priorities between vertices on generation, which has a minor impact on the runtime of Zielonka's algorithm.

In theory, Strategy Iteration via DPGs ought to be a contender for large enough n for both the value and stratey problem. The reason for why it isn't depends on the reduction from MPGs to DPGs and is laid out in the subsection for MPGs.

Strategy Iteration from above via EGs can outperform Zielonka's algorithm for large $|V|$ and reasonably dense graphs. However, the break-even point is only reached for

such large $|V|$ and consequently such long runtimes that gathering statistically significant runtime data is impractical, it can however be confirmed for individual instances. We therefore cannot describe exact inflection points for the parameterization when the Strategy Iteration starts outperforming Zielonka's algorithm precisely.

Other routes of reduction and applied algorithms fail to be useful, because the edge-weights in the resulting MPG scale very strongly wrt. to the size of the PG and it's range of priorities. Many subsequent algorithms scale in runtime in dependence of the range of the edge-weights of the resulting MPG or equivalent parameter that are downstream from the range of the edge-weights under further reduction (range of edge-weights of EGs, discount factor of DPGs...). Strategy Iteration from above via EGs works decently here precisely because it performs indifferently to the range of edge-weights.

6.1.1. Value Problem of PGs

For solving the Value Problem of PGs the result are very clear. For all cases tested, Zielonka's Algorithm is straight up the fastest. It's simplicity demands very little overhead and as such small graphs are solved near instantaneously. It's recursive nature allows it to scale very well to big graphs as well. Less connectivity, as shown by smaller p, mean the attractors as laid out in the algorithm tend to not reach as far – meaning more levels of recursion are needed to cover the entire graph. Similargly, bigger w means that the winning regions that act as a crystallisation point for the attractors tend to start out smaller and thus form smaller attractors.

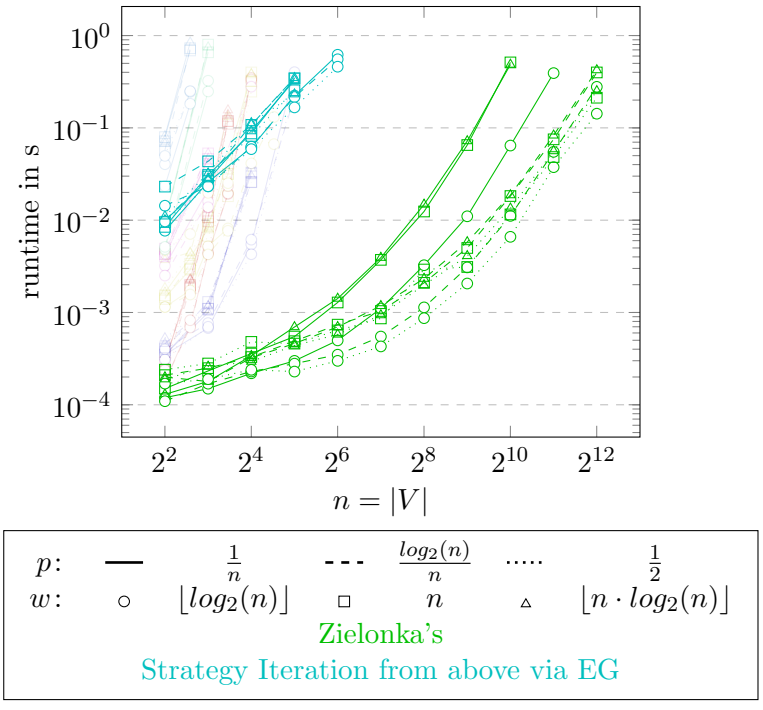


Figure 7: Solving the value problem of PGs

6.1.2. Strategy Problem of PGs

Unfortunately our naive approach to finding optimal strategies via Zielonka's algorithm is $|V| \cdot log_2(\frac{|E|}{|V|})$ times slower than finding the values alone, meaning the big lead it had in finding the values of games is diminished significantly, bringing it more in line with other algorithms. Depending on the exact p and w, the BCDGR algorithm via EGs, which has similarly low overhead to Zielonka's algorithm, but doesn't require repeated invokation to deliver optimal strategies, has better runtimes for reasonably small graphs. However, since it scales worse, such advantage is quickly lost with growing $|V|$. After that, Zielonka's algorithm, even with the malus it suffer for finding optimal strategies, is the fastest available method until Strategy Iteration from above via EGs may take over
.

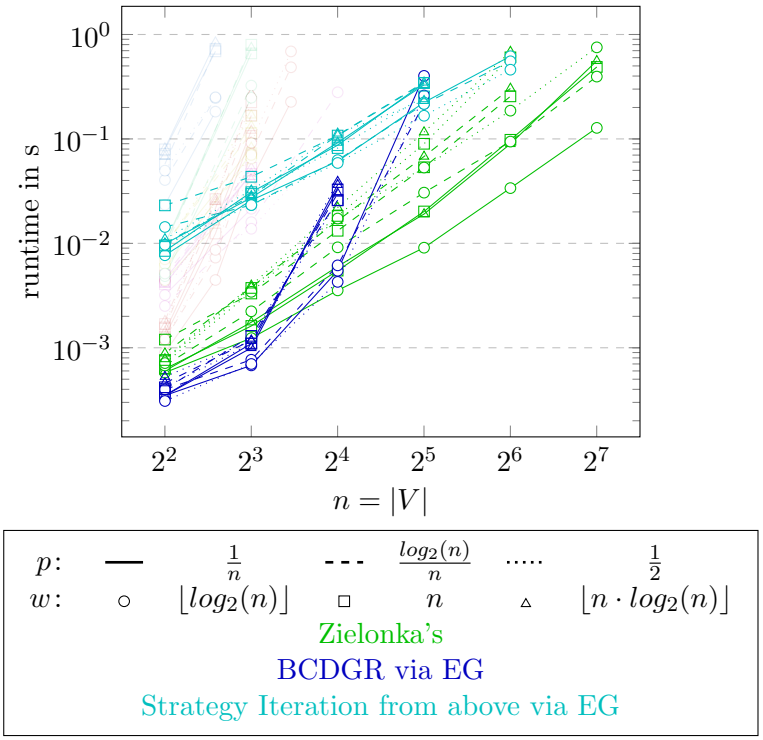


Figure 8: Solving the strategy problem of PGs

6.2. MPGs

The additional parameter is w, describing the range over which the edge-weights are handed out and for our random generation. They are once again randomly and uniformly distributed over their range. Unlike with PGs, in this case the meaning of w cannot be scaled down to make subsequently applied algorithms faster. As a result we see some algorithms scale strongly in response to changing w across it's range and even beyond.

In theory, Strategy Iteration via DPGs or SSGs is faster than other algorithms for large enough n. In practice however this isn't easily achieved. The method by which MPGs are reduced to DPGs results in discount factors very close to 1. So close in fact, that for big enough MPGs the difference $1 - \lambda$ approaches the limit of the precision standard floating point implementations can represent [75419]. As a result, standard Linear Program solvers are unable to solve such Linear Programs that are part of the strategy iteration with adequate precision. One could conceive a Linear Program solver that works with larger floats that enable higher precision. However this only raises the ceiling of what is solveable marginally. It also eventually forces the floats to exceed standard register sizes, meaning the floating point arithmetic fundamental to the Linear Program solver would have to resort to emulating floating point arithmetic for larger floats, which would take more clock cycles per operation and ultimately longer runtimes, undermining the usefulness of the approach in the first place. Effectively this mean that, even if this approach is plausible, it is never pratically viable for MPGs reduced to DPGs.

Kleene Iteration via DPGs or SSGs also fails to be useful because of the discount factor being close to one. The convergence rate of the fundamental fix-point iteration depends directly on the discount factor, with a higher discount factor directly leading to slower convergence.

6.2.1. Value Problem of MPGs

Unlike for PGs the native approach used here, Zwick and Paterson's algorithm isn't a clear winner. Intuitively the $k = 4 \cdot |V|^3 \cdot d$ iterations the algorithm goes through to guarantee correcteness are way in excess of what's practically needed. This means it's only competetive for graphs with very small $|V|$. As with PGs, solving with BCDGR via EG offers low overhead and thus reasonably fast runtimes for smaller graphs, but again as with PGs the approach scales mediocrely wrt. $|V|$.
As with PGs, Strategy Iteration from above via EGs eventually outperforms BCDGR via EG for some parameterizations. *Unlike* with PGs, we don't suffer from an as large d any more, meaning the approach becomes competitive much sooner. The exact inflection point depend on the specific MPG in question, with a relatively large overlap where either may be superior depending on the underlying MPG in question.

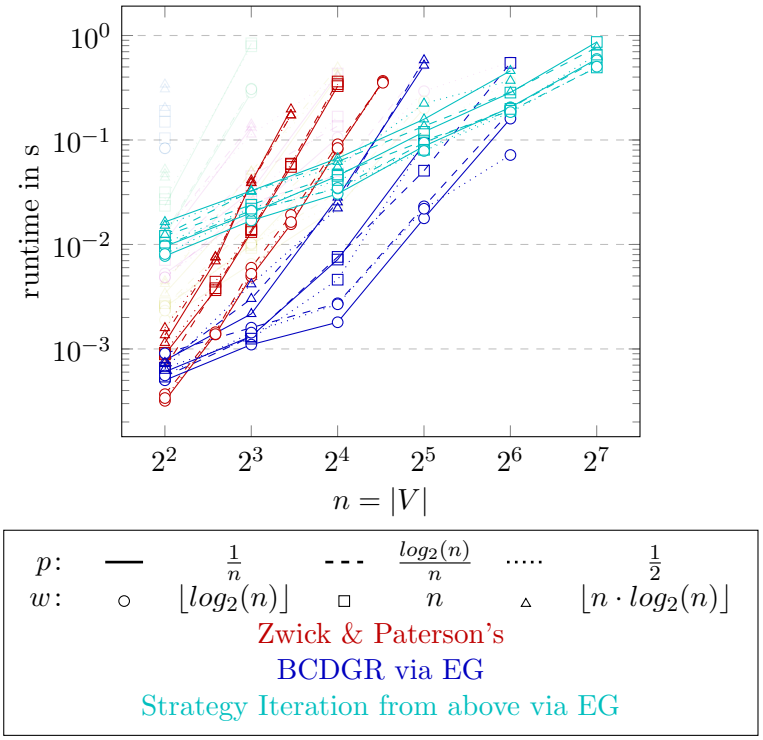


Figure 9: Solving the value problem of MPGs

6.2.2. Strategy Problem of MPGs

As with Zielonka's algorithm on PGs, Zwick and Paterson's algorithm for optimal strategies has an additional cost of $|V| \cdot log_2(\frac{|E|}{|V|})$ over finding the values of the game. Unlike with Zielonka's algorithm on PGs this makes Zwick and Paterson's algorithm non-viable for all conditions for finding strategies. Since BCDGR and Strategy Iteration from above via EG both don't have any malus when finding optimal strategies over values, their relative performance stays the same is was for finding values, with BCDGR being the choice for smaller graphs and Strategy Iteration for larger graphs.

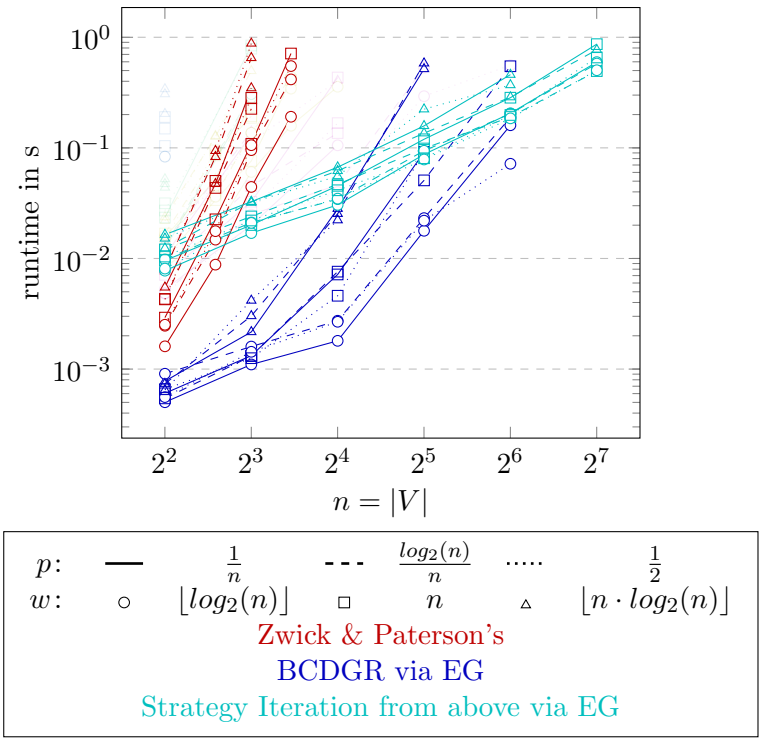


Figure 10: Solving the strategy problem of MPGs

6.3. EGs

Once more the additional parameter w describes the range over which the edge-weights are handed out and they are once again randomly and uniformly distributed over their range.

Since all algorithms of interest perform the same for solving for values and optimal strategies, we do not distinguish between the two problems here.

Strategy Iteration from above/below differ internally in whether they use Linear Programming/Kleene Iteration to solve the partial problem within them. Kleene Iteration has very little constant overhead, which allows Strategy Iteration from below to outperform Strategy Iteration from above for smaller graphs. Since it scales significantly worse than the solving of Linear programs, Strategy Iteration from above eventually overtakes for all paramterizations. The exact inflection point varies widely, with the tendency of Strategy Iteration from below to scale worse the less connected the graph is while Strategy Iteration from above scales very indepedant from anything but size.

BCDGR outperforms both for all parameterizations. Especially at the low end, but it maintains a > 3 fold advantage across all cases tested. However, BCDGR and Strategy Iteration from above seem to scale very similarly asymptotically. If one can improve the runtime of Strategy Iteration from above by just a *constant* factor of > 3, which is easily conceivable, then it stands to be a contender for fastest method for at least some parameterizations.

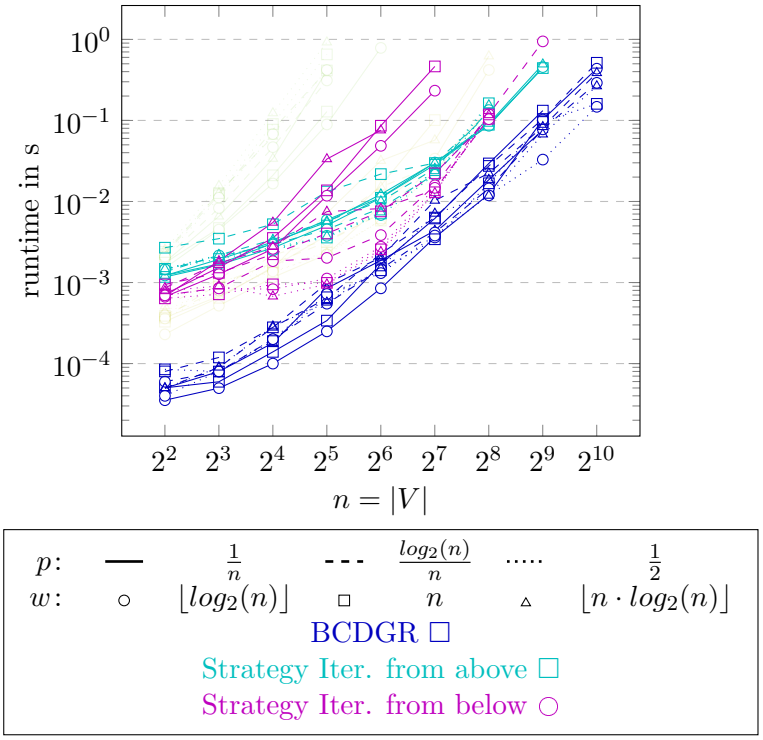


Figure 11: Solving both EG problems

From looking at the performance of solving MPGs via EGs one might expect BCDGR

to underperform compared to Strategy Iteration from above. However the relative performance of the two is heavily dependent on the way the reduction from MPGs to EGs works. The sub-problems spawned during the reduction from MPGs to EGs do no represent a random sample and are significantly skewed in a way that it disadvantages BCDGR.

6.4. DPGs

For DPGs we could have two additional parameters: λ which is the discount factor and w which again describes the range of the edge-weights. Since all approaches are indifferent to w, we omit it.

As with EGs, all algorithms solve for the value as well as optimal strategies, so we do not distinguish them here.

As already hinted at while solving MPGs by Kleene Iteration via DPGs, Kleene Iteration for DPGs has a convergence rate directly tied to the discount factor while being indifferent towards the connectivity of the graph. Meanwhile Strategy Iteration is weakly dependent on the discount factor while being strongly dependent on the connectivity of the graph, especially towards the lower end.
This leads to an interesting situation where instead of one approach eventually outperforming all others for some $|V|$ threshold, it equally depends on the connectivity and the discount factor which approach is favourable.
Strategy Iteration and Kleene Iteration via SSGs are viable, but consistently slower by some constant factor.

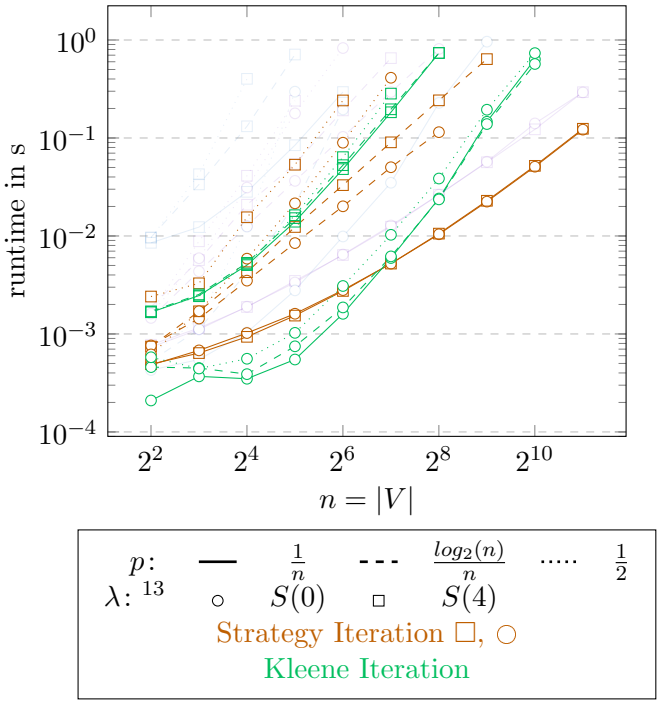


Figure 12: Solving both DPG problems

[13] $S(x) = \frac{1}{1+e^{-x}}$

6.5. SSGs

For evaluating SSGs we run into a problem: Strategy Iteration only works for stopping-SSGs and we have no generalized way to generate those. What we do is we compare the runtimes for DPGs converted to SSGs, which *will* be stopping-SSGs but *will not* be a random sample of stopping-SSGs. Here, p, λ and n refer to the original DPG before reduction. Again, both approaches solve for both value and optimal strategies.

The result is, unsurprisingly, reminiscent of the results for Strategy Iteration and Kleene Iteration for DPGs. Computationally speaking, SSGs are solved very similarly to DPGs for both Strategy Iteration as well as Kleene Iteration. For both, the sink vertices and the introduction of the *Random* vertices don't have significant implications towards the complexity of the underlying Linear Programs or fix-point iterations. As a result, the runtime behaviour is essentially the same as for DPGs when adjusted for the contortions caused by the reduction from DPGs to SSGs. Particularly the diffraction of the runtime of the Kleene Iteration wrt. p_{DPG} is an artefact of the reduction, where a higher p_{DPG}, i.e. a higher $|E_{DPG}|$, leads to a higher $|V_{SSG}|$ in the resulting graph which in turn reduces the convergence rate of the fix-point iteration. Whereas the Linear Program *does* grows fourfold in dimensionality with respect to $|E_{DPG}|$, three of those are equality relationships and only one of them is a half-space, effectively only impacting the runtime complexity of the Linear Program by a constant factor as can be seen by the difference between solving DPGs via Strategy Iteration natively on DPGs versus after reducing to SSGs.

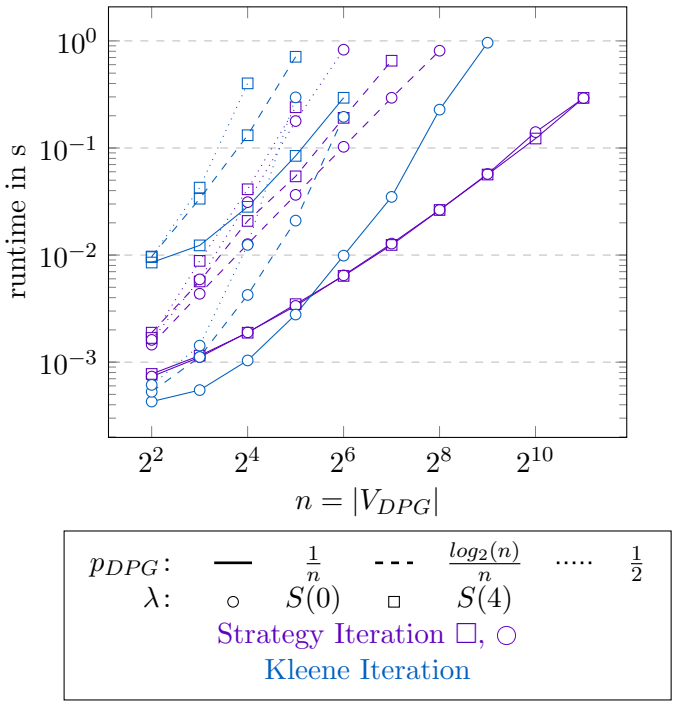


Figure 13: Solving both stopping-SSG problems

7. Conclusion and Future

7.1. Conclusion

We set out to see how the different ways to solve problems, the reductions between those problems and the combination of the two play out and compare to one another in a real-world implementation. To that end we implemented all algorithms and reduction and benchmarked all combinations of those.

As laid out in the Introduction, during the reduction the games tend to grow in some way. $PG \to MPG$ causes the range of the edge-weights in MPG to grow faster than the range of priorities in PG, in $DPG \to SSG$ both $|V_{SSG}|$ and $|E_{SSG}|$ grow faster than $|V_{SSG}|$ and $|E_{SSG}|$ and in $MPG \to EG$ the underlying graph stays unchanged, but we need to recursively solve multiple instances of EGs to solve one MPG. Additionally, for the algorithms we tested, $MPG \to DPG$ resulting in a discount factor very close to one means that the resulting DPG becomes more demanding to solve.
This may make it seem like approaches that involve reduction are futile, however this is not always the case.
One exception is that despite the disadvantage, a reduction and algorithm may still outperform algorithms that are native to the original game. This is the case for $MPG \to EG$, where BCDGR via EGs, despite the necessary recursion it outperforms Zwick and Paterson's algorithm on the original MPG. This was in fact a major motivation for the BCDGR algorithm [BCD$^+$11] and we can confirm that this theoretical advantage holds in practice.
Another is that the algorithm does get outperformed asymptotically in part due to the reduction, but performs well enough for smaller problems that it's still viable for some range of problems. This happens while solving for optimal strategies via $PG \to EG$. Here BCDGR via EGs suffers from the large range of edge-weights the reduction $PG \to MPG$ creates, but isn't affected enough for smaller games as to be immediately non-viable.

The most noteable cases are those involving reduction and Strategy Iteration utilizing Linear Programming (i.e. all but Strategy Iteration from below for EGs).
For PGs, the reduction to EGs and solving by Strategy Iteration from above, particularly for optimal strategies, does seem to have some use cases. However these are for relatively large games and Zielonka's algorithm is usefull for a wide range of games, in part by a huge margin.
The really interesting case seems to be reducing MPGs to EGs and then solving by Strategy Iteration from above. Beyond a certain size of the game, this method seems to outperform all other approaches for all cases. In theory the same logic should extend to solving MPGs via Strategy Iteration via DPGs, however as laid out in 6.2, this fails due to the inability to get a satisfying amount of numerical precision.

Keep in mind that for all approaches the exact runtimes depend on the specific implementation in question and can thus change to some degree. The general relations between the runtimes of the approaches would be expected to stay broadly the same, but details, particularly the inflection points of when exactly one approach is expected to start to outperform another are subject to the specific implementation and can change in response to a change in implementation.

The exception would be BCDGR versus Strategy Iteration from above on EGs, which display similar asymptotic behaviour with seemingly only some constant difference in runtime. If Strategy Iteration from above could be sped up significantly, if could potentially not just change the inflection point but the whole relation of the two where Strategy Iteration from above is strictly faster for some or all EGs above a certain size.

7.2. Future

If the objective is the absolute speed of solving any of the problems, the way to go would be an implementation dedicated to that. Here, we focused on relative speed rather than absolute speed. A first step would be an implementation in a compiled language, rather than Python. Explorative tests showed a speed-up of about 30 times in C++ for Kleene Iterations and Strategy Iteration with Linear Programming.
Another improvement that can be made use of is the continual development and improvement of Linear Programm solvers. For all but the smallest graph, the Strategy Iteration algorithms that utilize Linear Programming do so in a way that Linear Programming represents most of the actual workload. If significant advances are made here, these would be expected to translate very well into the runtime behaviour of Linear Program based Strategy Iteration.
Some of the approaches such as all those utilizing Kleene Iteration and Zwick and Paterson's algorithm for MPGs make heavy use of matrix and vector operations in a way that they could greatly benefit from hardware acceleration such as on a GPU.

Ultimately the most significant improvements are probably possible by way of new algorithms altogether. For some problems there are already algorithms that are theoretically better, like [LPSW22] for PGs, but a common theme is that those do not actually perform better or even worse in practise than previous approaches.
Ideally we'd want a polynomial-time algorithm for any of our problems, but so far those have eluded us [BDM16].

A. Versicherung an Eides Statt

Ich versichere an Eides statt durch meine untenstehende Unterschrift,

- dass ich die vorliegende Arbeit - mit Ausnahme der Anleitung durch die Betreuer - selbstständig ohne fremde Hilfe angefertigt habe und
- dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus fremden Quellen entnommen sind, entsprechend als Zitate gekennzeichnet habe und
- dass ich ausschließlich die angegebenen Quellen (Literatur, Internetseiten, sonstige Hilfsmittel) verwendet habe und
- dass ich alle entsprechenden Angaben nach bestem Wissen und Gewissen vorgenommen habe, dass sie der Wahrheit entsprechen und dass ich nichts verschwiegen habe.

Mir ist bekannt, dass eine falsche Versicherung an Eides Statt nach § 156 und nach § 163 Abs. 1 des Strafgesetzbuches mit Freiheitsstrafe oder Geldstrafe bestraft wird.

Ort, Datum	Unterschrift

References

[75419] IEEE Standard for Floating-Point Arithmetic. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), S. 1–84. http://dx.doi.org/10.1109/IEEESTD.2019.8766229. – DOI 10.1109/IEEESTD.2019.8766229

[BCD+11] Brim, L. ; Chaloupka, J. ; Doyen, L. ; Gentilini, R. ; Raskin, J. F.: Faster algorithms for mean-payoff games. In: *Formal Methods in System Design* 38 (2011), Apr, Nr. 2, 97-118. http://dx.doi.org/10.1007/s10703-010-0105-x. – DOI 10.1007/s10703–010–0105–x. – ISSN 1572–8102

[BDM16] Benerecetti, Massimo ; Dell'Erba, Daniele ; Mogavero, Fabio: Solving Parity Games via Priority Promotion. In: Chaudhuri, Swarat (Hrsg.) ; Farzan, Azadeh (Hrsg.): *Computer Aided Verification*. Cham : Springer International Publishing, 2016. – ISBN 978–3–319–41540–6, S. 270–290

[Jur98] Jurdzinski, Marcin: Deciding the Winner in Parity Games is in UP \cap co-Up. In: *Inf. Process. Lett.* 68 (1998), Nr. 3, 119–124. http://dx.doi.org/10.1016/S0020-0190(98)00150-1. – DOI 10.1016/S0020–0190(98)00150–1

[LPSW22] Lehtinen, Karoliina ; Parys, Pawel ; Schewe, Sven ; Wojtczak, Dominik: A Recursive Approach to Solving Parity Games in Quasipolynomial Time. In: *Logical Methods in Computer Science* Volume 18, Issue 1 (2022), Januar. http://dx.doi.org/10.46298/lmcs-18(1:8)2022. – DOI 10.46298/lmcs–18(1:8)2022

[ZP96] Zwick, Uri ; Paterson, Mike: The complexity of mean payoff games on graphs. In: *Theoretical Computer Science* 158 (1996), Nr. 1, 343-359. http://dx.doi.org/https://doi.org/10.1016/0304-3975(95)00188-3. – DOI https://doi.org/10.1016/0304–3975(95)00188–3. – ISSN 0304–3975