DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

HAMBURG UNIVERSITY OF TECHNOLOGY

Bachelor's Thesis in Informatics: Game Theory

New Algorithms for Classes of Mean-Payoff Games

Anna Hauschild

TUHH
Hamburg
University of
Technology

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

HAMBURG UNIVERSITY OF TECHNOLOGY

Bachelor's Thesis in Informatics: Game Theory

New Algorithms for Classes of Mean-Payoff Games

Author:	Anna Hauschild
Supervisor:	Prof. Matthias Mnich
Advisor:	Dr. Julian Golak
Submission Date:	May 27, 2022

TUHH
Hamburg
University of
Technology

I confirm that this bachelor's thesis in informatics: game theory is my own work and I have documented all sources and material used.

Hamburg, May 27, 2022 Anna Hauschild

Abstract

This theses provides an implementation of a new algorithm that solves a special class of mean payoff games. These are games played between two players on weighted and directed graphs with the aim of minimizing or maximizing their payoffs. So far there is no polynomial time algorithm known for computing the winning sets of the players and the algorithm is classified to the complexity class NP $\cap$ coNP. To solve these types of games, the algorithm removes edges without effect on the outcome and applies a reachability game to a found and solved subgraph. These steps are derived in detail with pseudo code and calculations, followed by a complete example of solving that game. Concluding this thesis the implementation runtime is validated and improvements suggested.

Contents

1 Introduction

Many processes in everyday life can be interpreted as a kind of game between two interacting players, with each player acting strategically to achieve their own goal. Basically, every competition can be viewed as a game with a specific outcome (win, lose, draw)[6].

In areas such as artificial intelligence, attempts are made to reproduce decision-making structures by allowing programs to process problems relatively independently. Decision situations are modeled and predictions are made. Regarding game theory the decisions of one player always affect the outcome of the opponent.

Mean payoff games, introduced by Ehrenfeucht and Mycielski in [4], are played on directed and weighted graphs. During the game the players seek to minimize or maximize their mean payoff. In practice they could be used, for example, to model worst-case scenarios of systems, in which the system plays against the user. In general they are found when scheduling and analyzing applications with cost functions where costs could be power usage or buffer size.[9]

These types of games represent a large part of game theory and have an interesting aspect in complexity theory. We study research to find solutions to such games, always with the aim of optimizing them. So far we know that there are algorithms that solve games like mean payoff games in exponential time. However it is an unsolved problem whether there are also polynomial time algorithms. But we know that this decision problem is in the complexity class NP $\cap$ coNP, which means it is not facile to verify correct answers, but at least to exclude incorrect ones.

M. Akian, S. Gaubert, O. Lorscheid and M. Mnich devised in the preprint [1] a new algorithm that finds the winning positions of a mean payoff game. With the idea of filtering edges and solving subgraphs followed by reachability games they succeed in solving subclasses of games with certain properties even in linear time.

The new algorithm is presented in this thesis, also we provide a related implementation in C that should allow us to solve games in practice and to compare the practical runtime with the theoretical.

2 Mean payoff games

2.1 Introduction to mean payoff games

We consider an infinite deterministic game with perfect information between two players *Min* and *Max* on a finite directed and weighted graph G with vertex set V, edge set E and cost function $\mu : E \to \mathbb{Z}$. The vertices are divided between the players into the sets V_+ and V_- while $V = V_+ \cup V_-$. During the game, defined as a sequence of vertices $\pi = (v_0, v_1, ...)$, each player decides which path to choose outgoing from its vertices. If $v \in V_+$ *Max* decides the next move and if $v \in V_-$ it is *Mins* turn. In the case of a bipartite graph the players take turns. The goal of player *Max* is to obtain a positive mean payoff

$$\limsup_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} \mu(v_i, v_{i+1}),$$

while player *Min* strives to minimize the weights

$$\liminf_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} \mu(v_i, v_{i+1}).$$

Our algorithm finds the winning positions of each player. For every node as starting position following an *optimal positional strategy* the vertices are assigned to the winning sets W_+ for *Max*, W_- for *Min* and W_0 in case of a draw. An optimal positional strategy means the strategy that leads to an optimal outcome assuming the opponent also makes use of his optimal positional strategy. Furthermore the result does not depend on previous moves, only on the current position.

A game begins in an arbitrary vertex v_1 and ends either in a vertex v_n without outgoing edges, called *sink*, or in an infinite circle. In case of an end due to a sink *Max* wins if $v_n \in V_-$ and accordingly *Min* wins if $v_n \in V_+$. If the game ends in a loop the mean payoff converges to the *characteristic value*

$$\chi(v_1) = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} \mu(v_i, v_{i+1}), \tag{2.1}$$

with $\mu(v_i, v_{i+1})$ depending on the optimal positional strategies of *Min* and *Max*. If $\chi(v_1) < 0$ *Min* wins, if $\chi(v_1) > 0$ *Max* wins and for $\chi(v_1) = 0$ it is a draw [1, 4].

In this way v_1 is assigned to W_+, W_- or W_0. Computing all characteristic values we can assign every individual vertex to its winning set and determine the winner based from the starting position.

Considering the example of the graph G_1 in Figure 2.1 with vertex set $V = \{v_1, v_2, v_3, v_4\}$ and edge set $E = \{(v_1, v_2), (v_2, v_1), (v_2, v_3), (v_3, v_1)\}$ the winning sets are

$$W_+ = \{v_4\} \qquad W_- = \emptyset \qquad W_0 = \{v_1, v_2, v_3\}.$$

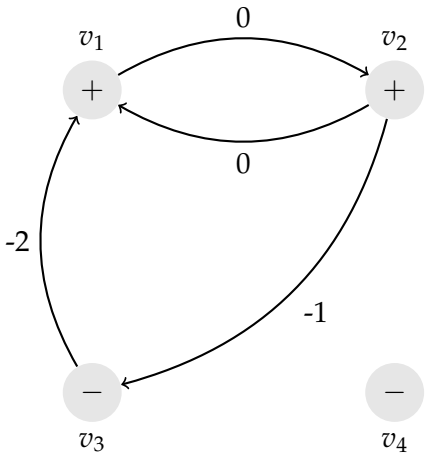

Figure 2.1: Graph G_1

Obviously v_4 is a sink and can be assigned to W_+. Starting from v_1 the only possible move is towards v_2. From v_2 *Max* has the option to move back to v_1 what ends for both of them in W_0 otherwise edge (v_2, v_3) would lead to a negative payoff and *Max* would lose. The only option for *Min* starting in v_3 is to follow the edge (v_3, v_1). Since the optimal positional strategy for v_2 implies the move to v_1 we end up in the circle $(..., v_1, v_2, v_1, v_2, ...)$ and $\chi(v_3) = \lim_{n\to\infty} \frac{-2}{n} = 0$ as well as $\chi(v_1) = \chi(v_2) = 0$.

What is noticeable is that the edge (v_2, v_3) is never played. This edge will be defined as *obsolete*. Also notable is that the vertices belonging to a circle are refound in the same winning set. So does the vertex reaching that circle.

2.2 State of the art

With the aim of optimizing the runtime of an algorithm for solving this and similar games, there are many attempts to find structures in the games for which a faster runtime can be determined. In this way it is tried to narrow down the unsolvable problems and improve them step by step. From a successful discovery we may someday be able to extend the game form and find a way back to the general case. Such approaches consider for example sinkless or drawless games as used in the by J. Chaloupka and L. Brim designed algorithm that solves games with these properties in $\mathcal{O}(|V|^2 \cdot |E| \cdot W \cdot \log((|V| \cdot W)$.[3]

The basic idea comes from Ehrenfeucht and Mycielski who derived the characteristic value and showed that calculating the optimal positional strategy suffices to obtain it.[4]

Based on that work U. Zwick and M. Paterson developed an $\mathcal{O}((|V|^3 \cdot |E| \cdot W)$ time algorithm for calculating the characteristic values of mean payoff games without restrictions. They also provide an algorithm for finding optimal positional strategies. Regarding the weight structure their algorithm works in polynomial time if the weight function is limited to $\{-1, 0, 1\}$.[9]

So also K. Chatterjee, M. Henzinger, S. Krinninger and D. Nanonkai focus in their work on the weights. Research like this paper faces to examine connections to similar games in order to find parallel behavior and finally to be able to transfer the algorithms to larger classes of games.[7]

2.3 Signature of a potential and derived reachability games

In this thesis we consider the class of mean payoff games with the certain property of a *signature of a potential*. Therefore a value is assigned to each node which describes the potential. This potential can be seen as an opportunity to win regarding it's neighbours. Then the relation between two adjacent vertices is described by the following definition.

Definition 1. A *potential* on G is a function $\psi : V \to \mathbb{Z}$. The mean payoff game on G has the *signature of a potential* $\psi : V \to \mathbb{Z}$ if for all edges $(v, w) \in E$,

$$(P0) \qquad \mu(v,w) = \mu(w,v) = \psi(v) - \psi(w) = 0 \qquad if\ \epsilon(v) = \epsilon(w),$$

$$(P1) \qquad sign(\epsilon(v)\mu(v,w) - \epsilon(w)\mu(w,v)) = sign(\psi(v) - \psi(w)) \qquad if\ \epsilon(v) \neq \epsilon(w)$$

The costs between two adjacent vertices belonging to the same player are 0 and the potential is the same. If a player moves multiple consequently times the score does not change. This indicates similarity to a bipartite graph i.e. the players take turns. In addition the edges proceed in both directions. We find mean payoff games with the signature of a potential for example in *parity games* on symmetric bipartite graphs. Parity games are similar and can be reduced to mean payoff games in polynomial time. But until today there is no possibility known to generalize algorithms for solving parity games in polynomial time to mean-payoff games.[8]
Graph G_2 in Figure 2.2 gives an example for a game with a signature of a potential.

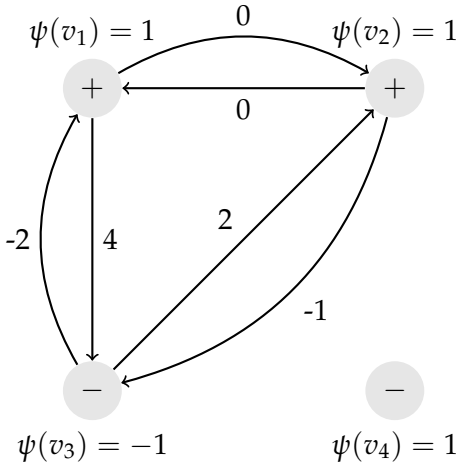

Figure 2.2: Graph G_2

(P0) is fulfilled since $\mu(v_1, v_2) = \mu(v_2, v_1) = \psi(v_1) - \psi(v_2) = 0$
For (P1) we check exemplary for (v_1, v_3):

$$\begin{aligned} \text{sign}(\epsilon(v_1)\mu(v_1,v_3) - \epsilon(v_3)\mu(v_3,v_1)) &= \text{sign}(\psi(v_1) - \psi(v_3)) \\ \text{sign}(1 \cdot 4 - (-1)(-2) &= \text{sign}(1 - (-1)) \\ 1 &= 1 \end{aligned}$$

A reachability problem in general consists of checking whether there exists a path from a vertex v to a vertex u in a graph. Now let us define the *derived reachability game*.

Definition 2. The *derived reachability game of* G is the reachability game on the graph G' with vertex set $V' = V$, such as $V'_+ = V_+$ and $V'_- = V_-$, and edge set

$$E' = \{(v, w) \in E \mid \epsilon(v) = \epsilon(w) \text{ or } \epsilon(v)(\mu(v, w) + \mu(w, v)) > 0\}.$$

The winning sets are

$$W'_+ = \{v \in V_- \mid v \text{ is a sink in } G'\} \qquad \text{and} \qquad W'_- = \{v \in V_+ \mid v \text{ is a sink in } G'\}$$

Regarding Graph G_2 in Figure 2.2, Graph G'_2 in Figure 2.3 is shown in Figure 2.3 and has the winninng sets

$$W'_+ = \{v_1, v_2, v_3, v_4\} \text{ and } W'_- = \emptyset,$$

since the sinks v_3 and v_4 laying in V_- and v_1 as well as v_2 succeed to reach one of them.

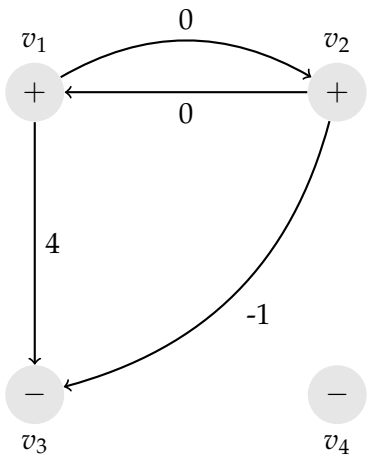

Figure 2.3: Graph G'_2

3 Algorithm structure

The idea of solving this game is based on four steps. Removing all obsolete edges, finding circles, solving them and assign the remaining vertices to a circle they can reach. Let $e(v) \in \{+, -\}$, with $+$ interpreted as 1 and $-$ interpreted as -1 and $n = \#V$.

3.1 Obsolete edges

In case of a sink the player belonging to the vertex immediately loses this node. If the vertex has exactly one outgoing edge and this edge could lead to a circle ending at the same point and yields a disadvantage for the player this edge could be removed and the vertex treated directly as a sink. The idea is, even for higher degrees, to interrupt unnecessary circles. In [1] obsolete edges are defined as follows:

Definition 3. Let $k \in \mathbb{N}$. An edge (v, w) is $k - obsolete$ if the opponent can always achieve to close the circle, such that $v_0 = w$ and $v_l = v$ with $l = min\{j \geq 1 \mid v_j = v\} \leq k - 1$ and

$$e(v) \sum_{i=0}^{l-1} \mu(v_i, v_{i+1}) < -e(v)\mu(v, w).$$

An edge (v, w) is *obsolete* if it is n-obsolete.

Proposition 1. *Let $k \in \mathbb{N}$ and $(v, w) \in E$ an k-obsolete edge. Omitting (v, w) from E does not change the winning positions W_+, W_- and W_0. The k-obsolete edges of G can be found in $\mathcal{O}(k \cdot n^2)$ steps.*

The first function *Obsolete* of our algorithm finds all obsolete edges of the game and represents a crucial part of the structure. The essential part is to define for a fix vertex v a recursive function $\eta(v)$ that indicates which neighbour finds a path with optimal positional strategy landing in v. If then the equation

$$e \cdot \eta_{k-1}(w) < -e \cdot \mu(v, w) \tag{3.1}$$

holds an obsolete edge is found and will be added to the set F of obsolete edges. Each vertex is considered individually with its own η -function. The pseudo code of the function *Obsolete* is described as follows.

Algorithm 1 Find all k-obsolete edges in G

function OBSOLETE(k, V_+, V_-, E, μ)
  $F = \emptyset$
  for $e \in \{+, -\}$ and $v \in V_e$ **do**
    for $i = 0, ..., k-1$ **do** $\eta_i(v) = 0$
    end for
    for $w \in (V_+ \cup V_-) - \{v\}$ **do** $\eta_0(w) := e \cdot \infty$
    end for
    for $i = 1, ..., k-1$ **do**
      for $w \in (V_+ \cup V_-) - \{v\}$ **do**
        $\eta_i(w) := \epsilon(w) \cdot max\{\epsilon(w) \cdot (\mu(w,u) + \eta_{i-1}(u)) \mid (w,u) \in E\}$
      end for
    end for
    for $(v,w) \in E$ **do**
      if $e \cdot \eta_{k-1}(w) < -e \cdot \mu(v,w)$ **then** $F := F \cup \{(v,w)\}$
      end if
    end for
  end for
  return F
end function

To implement this part we imagine $\eta(v)$ as $n \times n$ matrix for each vertex with the rows representing the step length i and columns the vertices. The matrix stores the path costs for a smallest path reaching the original node v. Starting to initialize the first row with $\eta_{i=0}(w) = e(v) \cdot \infty$ for all vertices $w \in V \setminus \{v\}$ indicates that no vertex reaches v with 0 steps and to initialize the $v - th$ column $\eta_i(v) = 0$ means v needs 0 steps to reach v. Row by row with increasing i paths could be found with lower costs. The last row stands for the $\eta_{k-1}(w)$ for $w \in V$. Considering the direct neighbours of v and comparing their path costs with the edge cost $\mu(v,w)$ an edge can be evaluated as obsolete or not obsolete. This way is applied to the following example graph G_3 in Figure 3.1 with $n = 5$.

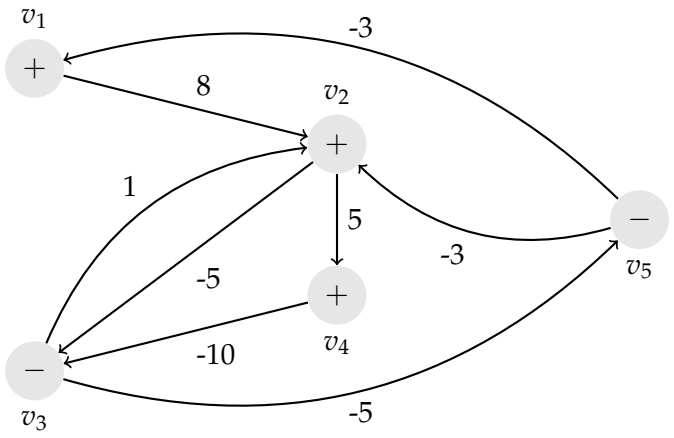

Figure 3.1: Graph G_3

For each node v:
Starting with $v = v_1$, $e = +$

for $i = 0$ initialize

$$\eta(v_1) = \begin{bmatrix} 0 & \infty & \infty & \infty & \infty \\ 0 & & & & \\ 0 & & & & \\ 0 & & & & \\ 0 & & & & \end{bmatrix}$$

and for all other nodes calculate
$\eta_i(w) := \epsilon(w) \cdot max\{\epsilon(w) \cdot (\mu(w,u) + \eta_{i-1}(u)) \mid (w,u) \in E\}$

$i = 1$
$v_2: \quad e(v_2) = +$
$\eta_1(v_2) = 1 \cdot \max\{1 \cdot \infty, 1 \cdot \infty\} = \infty$

$v_3: \quad e(v_3) = -$
$\eta_1(v_3) = (-1) \cdot \max\{(-1) \cdot \infty, (-1) \cdot \infty\} = \infty$

$v_4: \quad e(v_4) = +$
$\eta_1(v_4) = 1 \cdot \max\{1 \cdot \infty\} = \infty$

$v_5: \quad e(v_5) = -$
$\eta_1(v_5) = (-1) \cdot \max\{(-1) \cdot \infty, (-1) \cdot (-3+0)\} = -3$

filling the $i-th$ row the matrix $\eta(v_1)$ becomes

$$\eta(v_1) = \begin{bmatrix} 0 & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & -3 \\ 0 & & & & \\ 0 & & & & \\ 0 & & & & \end{bmatrix}$$

Vertex v_5 reaches v_1 in one step with the costs of -3.

Continuing with $i = 2$
v_2: $\eta_2(v_2) = 1 \cdot \max\{1 \cdot (-5-\infty), 1 \cdot (5+\infty)\} = \infty$
v_3: $\eta_2(v_3) = (-1) \cdot \max\{(-1) \cdot (1+\infty), (-1) \cdot (-5-3)\} = -8$
v_4: $\eta_2(v_4) = 1 \cdot \max\{1 \cdot (-10+\infty)\} = \infty$
v_5: $\eta_2(v_5) = (-1) \cdot \max\{(-1) \cdot (-3+\infty), (-1) \cdot (-3+0)\} = -3$

$i = 3$
v_2: $\eta_3(v_2) = 1 \cdot \max\{1 \cdot (-5-\infty), 1 \cdot (5+\infty)\} = -\infty$
v_3: $\eta_3(v_3) = (-1) \cdot \max\{(-1) \cdot (1+\infty), (-1) \cdot (-5-3)\} = -8$
v_4: $\eta_3(v_4) = 1 \cdot \max\{1 \cdot (-10-8)\} = -18$
v_5: $\eta_3(v_5) = (-1) \cdot \max\{(-1) \cdot (-3+\infty), (-1) \cdot (-3+0)\} = -3$

$i = 4$
v_2: $\eta_4(v_2) = 1 \cdot \max\{1 \cdot (-5-8), 1 \cdot (5-18)\} = -13$
v_3: $\eta_4(v_3) = (-1) \cdot \max\{(-1) \cdot (1+\infty), (-1) \cdot (-5-3)\} = -8$
v_4: $\eta_4(v_4) = 1 \cdot \max\{1 \cdot (-10-8)\} = -18$
v_5: $\eta_4(v_5) = (-1) \cdot \max\{(-1) \cdot (-3+\infty), (-1) \cdot (-3+0)\} = -3$

Finally the matrix becomes

$$\eta(v_1) = \begin{bmatrix} 0 & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & -3 \\ 0 & \infty & -8 & \infty & -3 \\ 0 & \infty & -8 & -18 & -3 \\ 0 & -13 & -8 & -18 & -3 \end{bmatrix}$$

For the only outgoing edge (v_1, v_2) it is checked whether $e \cdot \eta_{k-1}(w) < -e \cdot \mu(v, w)$ with $w = v_2$, $k = 5$ and $e = 1$. Since,

$$\begin{aligned} 1 \cdot \eta_4(v_2) &< (-1) \cdot \mu(v_1, v_2) \\ 1 \cdot (-13) &< (-1) \cdot 8 \end{aligned}$$

the edge (v_1, v_2) is obsolete.
For the remaining vertices v_2, v_3, v_4 and v_5 the matrices $\eta(v)$ are are listed below.

$$\eta(v_2) = \begin{bmatrix} \infty & 0 & \infty & \infty & \infty \\ 8 & 0 & 1 & \infty & -3 \\ 8 & 0 & -8 & -9 & -3 \\ 8 & 0 & -8 & -18 & -3 \\ 8 & 0 & -8 & -18 & -3 \end{bmatrix} \quad \eta(v_3) = \begin{bmatrix} -\infty & -\infty & 0 & -\infty & -\infty \\ -\infty & -5 & 0 & -10 & -\infty \\ 3 & -5 & 0 & -10 & -\infty \\ 3 & -5 & 0 & -10 & -8 \\ 3 & -5 & 0 & -10 & -8 \end{bmatrix}$$

$$\eta(v_4) = \begin{bmatrix} \infty & \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 0 & \infty \\ \infty & 5 & 6 & 0 & 2 \\ \infty & 5 & -3 & 0 & 2 \\ \infty & 5 & -3 & 0 & 2 \end{bmatrix} \quad \eta(v_5) = \begin{bmatrix} -\infty & -\infty & -\infty & -\infty & 0 \\ -\infty & -\infty & -5 & -\infty & 0 \\ -\infty & -10 & -5 & -15 & 0 \\ 2 & -10 & -5 & -15 & 0 \\ 2 & -10 & -5 & -15 & 0 \end{bmatrix}$$

This results in:
for v_2: v_2 has the neighbours v_3 and v_5

for $w = v_3$, $1 \cdot (-8) < (-1) \cdot 5$ $\Rightarrow (v_2, v_3)$ is obsolete
for $w = v_4$, $1 \cdot (-18) < (-1) \cdot (-5)$ $\Rightarrow (v_2, v_4)$ is obsolete

for v_3: v_3 has the neighbours v_2 and v_5

for $w = v_2$, $(-1) \cdot (-5) < 1 \cdot 1$ $\not\downarrow$ $\Rightarrow (v_3, v_2)$ is not obsolete
for $w = v_5$, $(-1) \cdot (-8) < 1 \cdot (-5)$ $\not\downarrow$ $\Rightarrow (v_3, v_5)$ is not obsolete

for v_4: v_4 has the neighbour v_3

for $w = v_3$, $1 \cdot (-3) < (-1) \cdot (-10)$ $\Rightarrow (v_4, v_3)$ is obsolete

for v_5: v_5 has the neighbours v_1 and v_2

for $w = v_1$, $(-1) \cdot 2 < 1 \cdot (-3)$ $\not\downarrow$ $\Rightarrow (v_5, v_1)$ is not obsolete

for $w = v_2$, $(-1) \cdot (-10) < 1 \cdot (-3)$ $\not\downarrow$ $\Rightarrow (v_3, v_5)$ is not obsolete

$\Rightarrow$ Concluding, the set of all obsolete edges becomes $F = \{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_4, v_3)\}$. Removing the obsolete edges F from all edges E leads to the graph G_3' in Figure 3.2 with vertex set V and edge set $E = E \setminus F$.

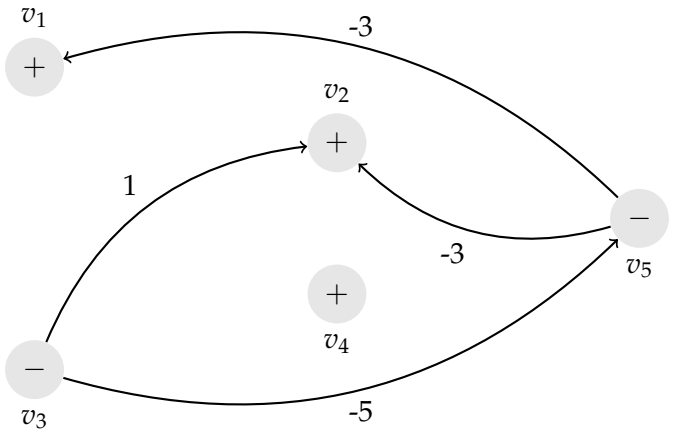

Figure 3.2: Graph G_3' after removing obsolete edges

Remark. *Computing one of these matrices requires $k \cdot n$ steps, repeating this calculation for all n matrices leads us to the complexity class $\mathcal{O}(k \cdot n^2)$ and here, since $k = n$, to $\mathcal{O}(n^3)$.*

Remark. *During the implementation special attention should be paid to the sinks, since in this particular case the algorithm searches for the maximum of an empty set. It makes sense to fill these entries with $e \cdot \infty$ as well.*

Remark. *It can occur that removing obsolete edges results in new obsolete edges, so we repeat this function until it finds no more of them.*

What we also see is that removing the obsolete edges from graph G_2 in Figure 2.2 results in the Graph G_2' in Figure 2.3. That observation indicates that for this particular class of mean payoff games the deletion of obsolete edges yields a complete reduction to a reachability game. Since reachabilty games can be solved in linear time the following theorem applies.

Theorem 1. *Let G be a mean payoff game that has the signature of a potential. Then the winning positions for the mean payoff game G are equal to the winning positions of the derived reachability game of G. Therefore the mean payoff game in G can be solved in linear time in $\#V + \#E$.*

3.2 Strongly connected components

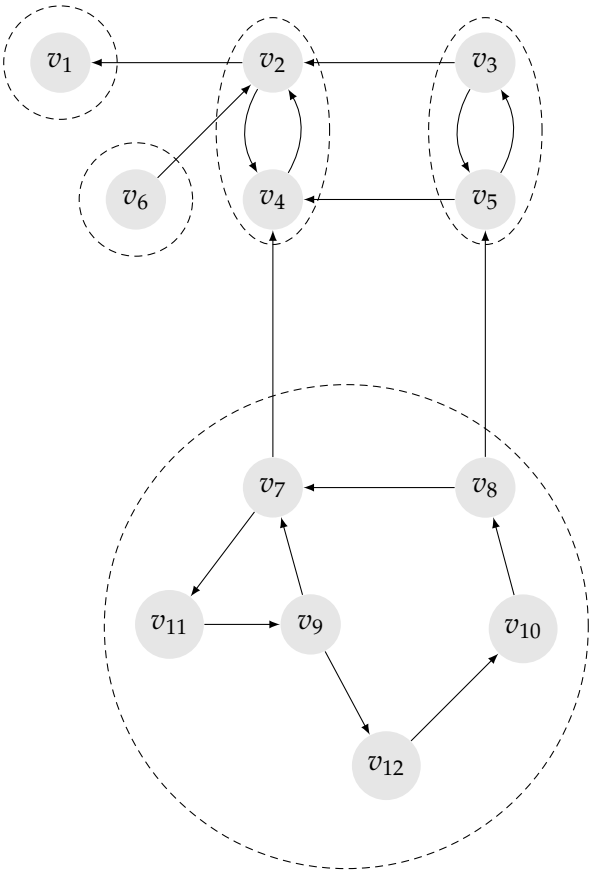

Figure 3.3: G_4 decomposed into it's strongly connected components

As already stated it is a basic decision to which winning set a vertex is allocated if this vertex is a sink. We continue to pursue this idea in the next step while looking for components in the graph that behave like sinks in the sense of not being able to leave. Well known that sinks are assigned to the opponent's winning set and remembering that loops converge and any nodes end up in that circle can be assigned to the correct winning set the purpose is to find *strongly connected components* solve them and look for vertices reaching them. By repeating these steps until all vertices are assigned, our algorithm is complete. Strongly connected components are maximal subgraphs of the graph G for which every vertex can reach all other vertices. The order in which these are found will play an important role. The function *StrConnComp* for finding all strongly connected components of the graph is shown below.

Algorithm 2 Find all strongly connected components of a subgraph X

```
    function StrConnComp(X, E)
2:      s := 0
        i := 0
4:      F =: {(v, w) ∈ E | v, w, ∈ X}
        while Y_1 ∪ Y_s ≠ X do
6:          if i = 0 then
                i := 0
8:              choose any v_1 ∈ X − (Y_1 ∪ ..., Y_s)
                m_1 := 1
10:             j := 1
            end if
12:         while j ≤ i do
                for ((v_j, w) ∈ F do
14:                 F := F − {(v_j, w)}
                    if w = v_k for some k < m_i then
16:                     for l = k + 1, ..., i doml := mk
                        end for
18:                     j := m_k
                    else
20:                     if w ∉ {v_{m_i}, ..., v_i} then
                            i := i + 1
22:                         v_i := w
                            m_i = i
24:                         j := i
                        end if
26:                 end if
                end for
28:         end while
            s := s + 1
30:         Y_s := {v_{m_i}, ..., v_i}
            i := m_i − 1
32:         j := m_i
        end while
34:     return (s, Y_s, ..., Y_1)
    end function
```

It reminds of *Tarjan's algorithm of strongly connected components*.[5]

The search starts in an arbitrary node v_1 and is stored in two arrays v and m, where v lists the already visited vertices v_j and m indicates a list of indices m_i for v with the information which nodes form a circle. For a reachable neighbour w it is checked whether it is already listed or new, i.e. if there is an entry $v_k = w$ in the list v for some $k < m_i$. If w has an entry all successor nodes starting from w form a circle and the corresponding indices are adjusted to be the same. If it is not part of the list w is added and we continue searching, now for the neighbour of w. If a circle is found the algorithm jumps back to the first node of this circle and continues the search for further outgoing edges to possibly expand this component. If there is no more edge leaving the component, it cannot be extended and a maximal circle is found. It is stored as the first component Y_1. The indices are adjusted so that the algorithm returns to the last node before entering the component. From there we keep looking. If no more connections can be found between components, the algorithm looks for a new starting node until all nodes are divided into components. Note that independently of the starting node, components with no outgoing edges are found first. The function returns the s, the number of components, and the components themselves in reverse order. A comprehensive example follows in the section 3.5 at the end of this chapter. The algorithm traverses each node once and considers each edge at most once. The complexity is therefore linear in $\mathcal{O}(|V| + |E|)$.

3.3 Solving subgraphs

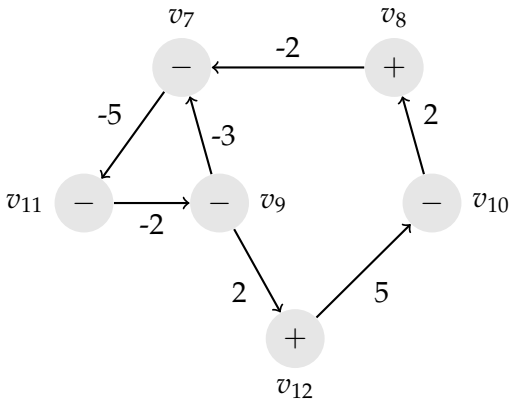

Figure 3.4: Subgraph of graph G_4

Under the assumption the subgraph is a strongly connected component the function *SolveMPG* solves the mean payoff game according to the *Zwick-Pateron's method*.[9]

If the component has no edges, it only contains a sink and can directly be assigned to one of the winning sets Y_+ or Y_-. Otherwise the optimal positional strategy finds a path by computing the characteristic value regarding $e(v)$. *Min* tries to minimize the path and *Max* to maximize it. Depending on whether the characteristic value is greater or less than 0, it belongs to Y_+ if $\chi(v_1) > 0$, to Y_- if $\chi(v_1) < 0$ or, in case of a draw, to Y_0 if $\chi(v_1) = 0$. For the implementation, the interval $[-2Mn, 2Mn]$ is defined instead of the comparison with exactly 0. To approximate the infinite number of steps, the value $4n^2M$ was estimated.

As a result of removing the obsolete edges, the components consist just of circles of equal

signed vertices unless there is a draw and the payoff becomes zero otherwise the circle would have been interrupted by the function *Obsolete*. Considering the component in Figure 3.4 two circles can be determined. The small one $(v_7, v_{11}, v_9, v_7, ...)$ and the extension through all vertices. Notice that this subgraph has no obsolete edges. The small loop leads to a negative payoff while the whole path determines a draw. Otherwise one of the edges would be obsolete. Since player *Min* has two possibilities to move the optimal positional strategy at this point is to choose the edge (v_9, v_7). So this game ends in the small loop and for all vertices the characteristic value $\chi(v) < 0$ applies with v as any starting point. The winning sets are assigned as $Y_- = \{v_7, v_8, v_9, v_{10}, v_{11}, v_{12}\}, Y_+ = \emptyset, Y_0 = \emptyset$.

Algorithm 3 Solving the mean payoff game on the subgraph

function SOLVEMPG(X, V_+, V_-, E, μ)
  $Y_+ = Y_- = Y_0 := \emptyset$
  $C := \{(v, w) \in E \mid v, w, \in X\}$
  if $C = \emptyset$ and $X \subset V_+$ **then**
    $Y_- := X$
  end if
  if $C = \emptyset$ and $X \subset V_-$ **then**
    $Y_+ := X$
  end if
  if $C \neq \emptyset$ **then**
    $n := \#V_+ + V_-$
    $M := max\{|\mu(v, w)| \mid (v, w) \in C\}$
    for $v \in X$ **do**
      $\eta_0(v) := 0$
    end for
    for $k = 1, ..., 4n^2M + 1$ **do**
      for $v \in X$ **do**
        if $v \in V_+$ **then**
          $\eta_k := max\{\mu(v, w) + \eta_{k-1}(w) \mid (v, w) \in C\}$
        end if
        if $v \in V_-$ **then**
          $\eta_k := min\{mu(v, w) + \eta_{k-1}(w) \mid (v, w) \in C\}$
        end if
        if $\eta_k(v) > 2nM$ **then**
          $Y_+ := Y_+ \cup \{v\}$
        end if
        if $\eta_k(v) < -2nM$ **then**
          $Y_+ := Y_- \cup \{v\}$
        end if
      end for
    end for
    $Y_0 := X - (Y_+ \cup Y_-)$
  end if
  return (Y_+, Y_-, Y_0)
end function

3.4 Reachability

If a strongly connected component has no outgoing edges the winning sets Y_+, Y_- and Y_0 of the subgraph correspond to the entire graph's final winning sets W_+, W_- and W_0. Hence, instead of solving all the components at once, the "sink component" is solved first and from there it is searched for nodes whose strategy leads into this set of vertices. The *Reach* function takes as input the target set X of nodes that the nodes outside the set are attempting to reach, the sign e, all vertices in their partition V_+ and V_- and an edge set E. It considers the edges (v, w) with $v \notin X$ and $w \in X$ and checks whether the player at vertex v takes an advantage of choosing this path or if all of it's strategies lead to X. In both of these cases X is expanded with v and the new X is returned at the end of the function. The function *Reach* depicted in

Algorithm 4 Finding all vertices from which player P_e can reach X

```
function REACH(X, e, V+, V−, E)
    for (v, w) ∈ E with v ∉ X and w ∈ X do
        if v ∈ V_e or u ∈ X for all (v, u) ∈ E then
            X := X ∪ {v}
        end if
    end for
    return X
end function
```

the pseudo code is used in two situations.

- $X = W_e$
  The set of reliably determined winning positions is considered. All vertices reaching X and belonging to the same player winning the play in X are added to X. There is no better way for them. For an opponent vertex with only one edge ending in X, there is no other choice to move. In this situation all vertices of the whole graph can be be regarded.

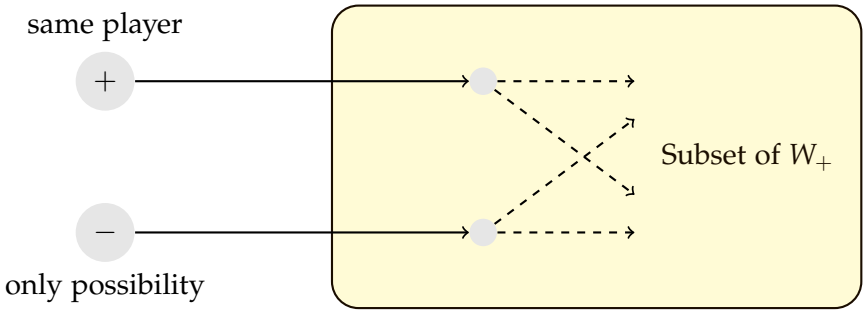

Figure 3.5: Finding additional winning vertices of *Max* via reachability

- $X = W_0$
  The winning set of draw positions is considered. Vertices that have just been assigned to the opponent subgraph's winning set Y_{-e} have the opportunity to escape the opposing set and improve to a draw. At this point only vertices from winning sets Y_e with $e \in \{+, -\}$ from strongly connected components are regarded.

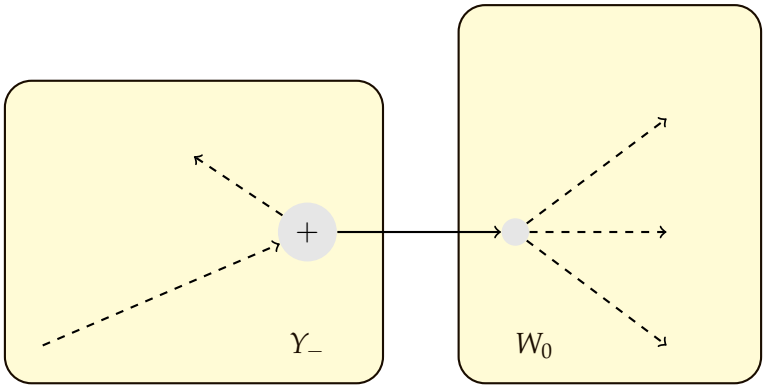

Figure 3.6: Additional draw positions for *Max*

3.5 Composition of the functions

After introducing the four fundamental functions we now combine them into the final algorithm. The idea is to remove firstly all the obsolete edges such that possibly many circles can be interrupted. That reduces large components. Solving a component and search directly for reachable nodes increases the winning sets faster than only solving the components.

The *main* function, listed as pseudo code below, gets a graph G consisting of the vertices sets V_+ and V_- such that $V_+ \cup V_- = V$, an edge set E and a cost function μ as input and returns the final winning sets W_+, W_- and W_0 such that $W_+ \cup W_- \cup W_0 = V$. All obsolete edges are removed until G has no obsolete edges anymore. The empty winning sets, the first component X_1 and the index r for the current component are declared and initialized,

$$W_+ = W_- = W_0 := \emptyset, \qquad X_1 = V \qquad \text{and} \qquad r = \begin{cases} 0 & \text{if } V = \emptyset \\ 1 & \text{if } V \neq \emptyset. \end{cases}$$

We enter into the while loop in line 14 which fills the winning sets until all vertices are assigned to their set. The while loop repeats the following steps:

1. The vertices are decomposed into strongly connected components.
2. For the last component the mean payoff game is solved.
3. Add draws to W_+ and search for additional draws in current winning sets and assign the remaining vertices to W_+ and W_-.
4. For all vertices it is checked whether they reach the winning sets.

Due to the reachability game the components have to be rebuilt and we restart the loop. When there are no more components left, the loop ends and the winning sets are returned.

Algorithm 5 Reduction of a mean payoff game via reachability games

function MAIN(V_+, V_-, E, μ)
  $F := Obsolete(\#V_+\# + V_-, V_+, V_-, E, \mu)$
  while $F \neq \emptyset$ **do**
    $E := E - F$
    $F := Obsolete(\#V_+ + V_-, V_+, V_-, E, \mu)$
  end while
  $W_+ = W_- = W_0 := \emptyset$
  if $V = \emptyset$ **then**
    $r := 0$
  else
    $r := 1$
  end if
  $X_1 := V$
  while $r > 0$ **do**
    $X := X_r - (W_+ \cup W_- \cup W_0)$
    $(s, X_r, ..., X_{r+s-1}) := StrConnComp(X, E)$
    $r := r + s - 1$
    $(Y_+, Y_-, Y_0) := SolveMPG(X_r, V_+, V_-, E, \mu)$
    $W_0 := W_0 \cup Y_0$
    for $e \in \{+, -\}$ **do**
      $F := \{(v, w) \in E \mid v \in Y_e, w \in Y_e \cup W_0$
      $W_0 := Reach(W_0, -e, V_+, V_-, F)$
      if $Y_e \cap W_0 = \emptyset$ **then**
        $W_e := W_e \cup Y_e$
      end if
    end for
    $W_+ := Reach(W_+, +, V_+, V_-, E)$
    $W_- := Reach(W_-, -, V_+, V_-, E)$
    while $r > 0$ and $X_r \subset W_+ \cup W_- \cup W_0$ **do**
      $r := r - 1$
    end while
  end while
  return (W_+, W_-, W_0)
end function

3.6 Example

Consider the following graph G_5 in Figure 3.7 and assume it has no obsolete edges. We start in the *while* loop in line 14.

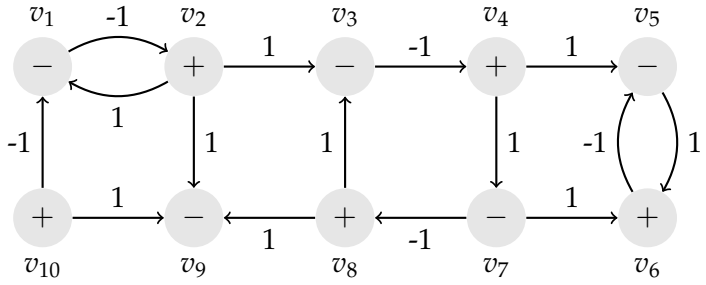

Figure 3.7: Graph G_5

• $(s, X_r, ..., X_{r+s-1}) := StrConnComp(X, E)$
Starting at v_1 a depth first search could be a path $(v_1, v_2, v_1, v_2, v_3, v_4, v_5, v_6, v_5)$ while after recognizing v_1 and v_2 belong to the same component but could eventually be expanded and determine (v_5, v_6) as first component Y_1, we resume the path at v_4: $(v_1, v_2, v_1, v_2, v_3, \mathbf{v_4}, v_7, v_8, v_3)$ and recognize a new component (v_3, v_4, v_7, v_8). Since v_8 has another outgoing edge this component could eventually be expanded thus the search continues at v_8: $(v_1, v_2, v_1, v_2, v_3, v_4, v_7, \mathbf{v_8}, v_9)$. v_9 is a sink and could not be left. Hence a next final component was found and assigned as the second one $Y_2 = (v_9)$. Backwards it is prospected for further edges. From the visited vertices no more edges are leaving so the circles (v_1, v_2) and (v_3, v_4, v_7, v_8) can now be verified as strongly connected components and backwards be assigned to $Y_3 = (v_3, v_4, v_7, v_8)$ and $Y_4 = (v_1, v_2)$. Now a new search has to start from any other node. It remains only v_{10} which forms the last component $Y_5 = (v_{10})$. The function *StrConnComp* returns the set

$$(s, Y_s, ..., Y_1) = (5, (v_{10}), (v_1, v_2), (v_3, v_4, v_7, v_8), (v_9), (v_5, v_6))$$

and in the *main* function we continue with $X_{r+s-1} = X_5 = (v_5, v_6)$.

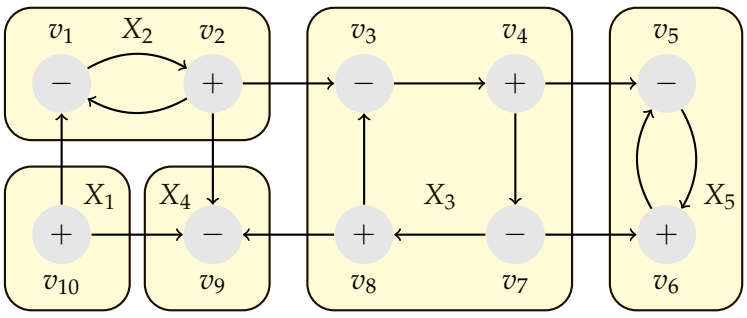

Figure 3.8: Strongly connected components

• $(Y_+, Y_-, Y_0) := SolveMPG(X_r, V_+, V_-, E, \mu)$

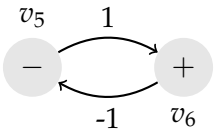

Figure 3.9: Subgraph of G_5

Solving the mean payoff game on the subgraph $X_5 = (v_5, v_6)$ resumes in the characteristic value

$$\chi(v_5) = \chi(v_6) = 0 \quad \Rightarrow \quad Y_+ = Y_- = \emptyset,\ Y_0 = \{v_5, v_6\}.$$

- $W_0 := W_0 \cup Y_0 = \{v_5, v_6\}$
- $X_5 \subset W_+ \cup W_- \cup W_0$ and r is redefined as $r = r - 1 = 4$

Now a new loop run starts. Nothing changes in the remaining components.
- $(s, X_r, ..., X_{r+s-1}) := StrConnComp(X, E)$

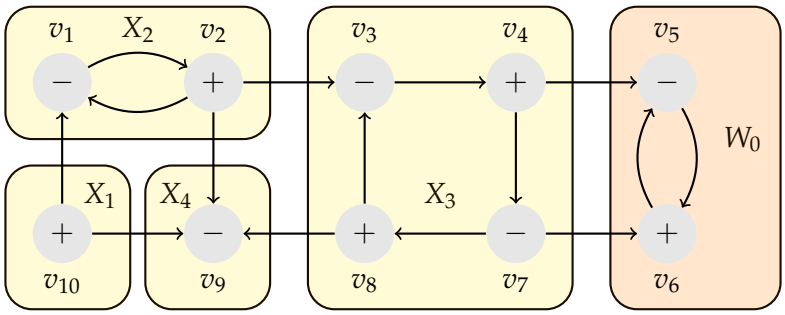

Figure 3.10: v_5 and v_6 assigned to W_0

- $(Y_+, Y_-, Y_0) := SolveMPG(X_r, V_+, V_-, E, \mu)$

Since $v_9 \in V_-$ *Max* wins at this sink and it is assigned to Y_+.

- if $Y_e \cap W_0 = \emptyset$ then $W_e := W_e \cup Y_e$

$\{v_9\} \cap \{v_5, v_6\} = \emptyset \quad \Rightarrow \quad W_+ = \{v_9\}$

- $W_+ := Reach(W_+, +, V_+, V_-, E)$

The vertices v_2, v_8 and v_{10} are considered because they reach v_9 and they all belong to *Max* therefore the winning set of *Max* is extended to $W_+ = \{v_2, v_8, v_9, v_{10}\}$. Also v_1 belongs to W_+ because even if it is *Min*'s turn at this position it's unique outgoing edge leads now to W_+. So it is extended again to $W_+ = \{v_1, v_2, v_8, v_9, v_{10}\}$.

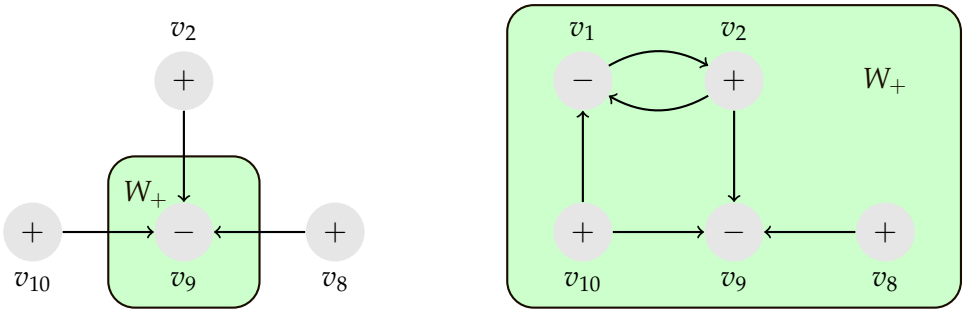

Figure 3.11: Reachability game

- $X_4 \subset W_+ \cup W_- \cup W_0 \quad r = r - 1 = 3$

In the next loop run $X_3 \setminus \{v_8\}$ is recomposed and yields three components of a single vertex.
- $(s, X_r, ..., X_{r+s-1}) = (3, X_3, X_4, X_5) := StrConnComp(X, E)$

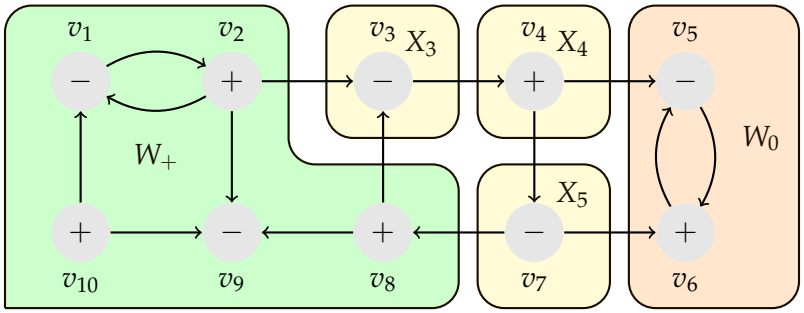

Figure 3.12: Third run of *StrConnComp*

- $(Y_+, Y_-, Y_0) := SolveMPG(X_r, V_+, V_-, E, \mu)$

X_5 is solved by determining v_7 as a sink in V_-, so v_7 is assigned to Y_+.

- $W_0 := Reach(W_0, -e, V_+, V_-, F)$

$v_7 \in Y_+$ reaches W_0 and because a draw is an improvement it becomes part of it.

- $X_5 \subset W_+ \cup W_- \cup W_0 \quad r = r - 1 = 4$
- $(Y_+, Y_-, Y_0) := SolveMPG(X_r, V_+, V_-, E, \mu)$

X_4 is solved by determining v_4 as a sink in V_+, so v_4 is assigned to Y_-.

- $W_0 := Reach(W_0, -e, V_+, V_-, F)$

$v_4 \in Y_-$ improves to a draw.

- $X_4 \subset W_+ \cup W_- \cup W_0 \quad r = r - 1 = 3$

- $(Y_+, Y_-, Y_0) := SolveMPG(X_r, V_+, V_-, E, \mu)$

X_3 is solved by determining v_3 as a sink in V_-, so v_3 is assigned to Y_+.

- $W_0 := Reach(W_0, -e, V_+, V_-, F)$

$v_3 \in Y_-$ improves to a draw.

- $X_3 \subset W_+ \cup W_- \cup W_0 \quad r = r - 1 = 2$
- $X_2 \subset W_+ \cup W_- \cup W_0 \quad r = r - 1 = 1$
- $X_2 \subset W_+ \cup W_- \cup W_0 \quad r = r - 1 = 0$

$\Rightarrow$ All vertices are assigned to the winning sets

$W_+ = \{v_1, v_2, v_8, v_9, v_{10}\}$ $W_0 = \{v_3, v_4, v_5, v_6, v_7\}$ $W_- = \emptyset$

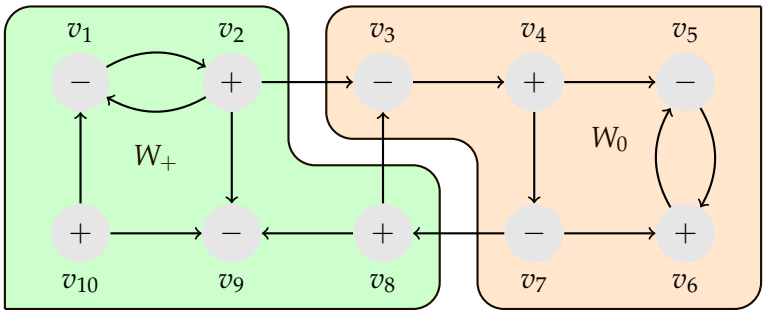

Figure 3.13: Resulting winning sets W_+ and W_0

3.7 Faster Algorithm for Mean-Payoff Games

The "Faster Algorithm for Mean-Payoff Games" developed by J. Chaloupa and L. Brim is an important algorithm to solve mean payoff games and we want to compare it to our algorithm.[3]
In order to reduce our algorithm to this one we adapt the game structure as follows:

1. Add an edge to each vertex to avoid sinks.
2. Redefine the winning sets as $W_- = W_-$ and $W_+ = W_+ \cup W_0$ so that it exists only the two outcomes win or lose.

It then fulfills the properties of the games that are solved by their algorithm. The idea is also to remove unnecessary edges what reminds us of omitting obslete edges. However it is solved in another way. All edges that don't belong to the optimal positional strategies of player *Min* and *Max* are removed so that it remains exactly one outgoing edge at each vertex. For every node it exists just one circle the node can reach. A value d is defined for each node, indicating a winning strategy for *Max*. While traversing the graph *Max* tries to keep this value positive. Obviously this is not possible in our original game because we consider draws. The fact that there are no draws in this game has the advantage that only one strategy has to be considered and since this is a 0-mean game, one player wins and the opponent automatically loses. The complexity of this alternative algorithm is $\mathcal{O}(|V|^2 \cdot |E| \cdot W \cdot \log((|V| \cdot W)$ which is faster than our algorithm for general graphs without the signature of a potential because the *SolveMPG* function alone has already the complexity $\mathcal{O}(|V|^3 \cdot M)$. But for the mean payoff games with a signature of a potential we achieve to provide a linear time of $|V| + |E|$.

4 Implementation and testing results

4.1 Initialization

The implementation of the algorithm is based on the pseudo code and written in C. It consists of a *main* function and the four crucial functions *Obsolete, StrConnComp, SolveMPG* and *Reach* described above. As input it receives a graph G consisting of a vertex set V, divided into $V = V_+ \cup V_-$, an edge set E and a cost function μ. The variable n specifies the number of vertices. The edges are placed in an adjacency matrix E that stores which vertices are connected, i.e. it has a row and a column for each node, resulting in a $n \times n$ matrix. An entry in the $i-th$ row and $j-th$ column indicates whether an edge leads from the $i-th$ to the $j-th$ node. If the entry is "0" there is no edge between v_i and v_j, a "1" instead indicates that an edge exists. Similarly, the weight is stored in an adjacency matrix μ. If $(v, w) \in E$ μ stores the value at the corresponding entry $\mu_{v,w}$ otherwise it is set to $-e(v) \cdot \infty$ where $\infty := 10000$. The array V_+ stores the vertices belonging to player Max and V_- the vertices belonging to player Min. These two arrays consist of n entries every index representing a vertex. A "1" at position i signifies v_i belongs to the player of the array.

4.2 Improvement

Depending on the amount of edges an improvement might be to use a process for storing *sparse matrices*, i.e. matrices with a large number of 0-entries. Sparse matrices have the property that they can be stored efficiently by storing only the positions and values of the non-zero entries. CRS (Compressed Row Storage) and CCS (Compressed Column Storage) are two ways of storing a sparse matrix in a space-saving manner.

That would certainly improve the runtime. In our case, the most time-consuming function is the *Obsolete* function, which allows the entire algorithm to be classified in $\mathcal{O}(n^4)$.

4.3 Graph generator

With the inspiration of the program to create a random graph[2] a specified graph generator was built. This program creates random graphs by assigning random neighbours to each node. The number of vertices and *degrees* can either be specified or arise randomly. In graph theory the degree is the number of edges leading to or leaving a node. The graph is stored in a text file from which the algorithm reads the data. The generator has two more options to specialize in mean payoff games with the signature of a potential. It is possible to check if the generated graph has a signature of a potential and moreover a graph with a signature of a potential can be created.

4.4 Testing results

In this chapter we use our implementation and the graph generator to experimentally test the practical runtime. The tables and graphs summarize the results. Each entry was tested several times and then the mean value was determined. As an abbreviation for signature of a potential, we use s.o.a.p. and #Vertices or #Components stand for the number of nodes or components, max degree means the maximal degree of each vertex.

4.4.1 Experiment 1

In Experiment 1 we increase the number of nodes and the max degree simultaneously. In the table 4.1 results are listed and Figure 4.1 plots the corresponding values. The time is shown on the y-axis, depending on the nodes on the x-axis. The orange graph describes the curve of the game with a signature of a potential and the blue graph the general game. We see that up to 300 nodes the time seems to be really good but then it starts to arise quickly. The graph with the s.o.a.p. has a better runtime than the general game but the general development of both graphs behaves similar. Since the *Obsolete* function is such expensive the results is not unexpected. Regardless of the type of graph it always computes the $n \times n$ matrices for n vertices as discussed in the chapter *Obsolete edges* which is a huge effort.

#Vertices	max degree	Time [s]	
		general	with s.o.a.p.
10	10	0.37	0.33
50	50	0.51	0.36
100	100	2.00	1.45
200	200	43.11	19.75
500	500	2047.86	1238.46

Table 4.1: Simultaneously increasing vertices and degrees

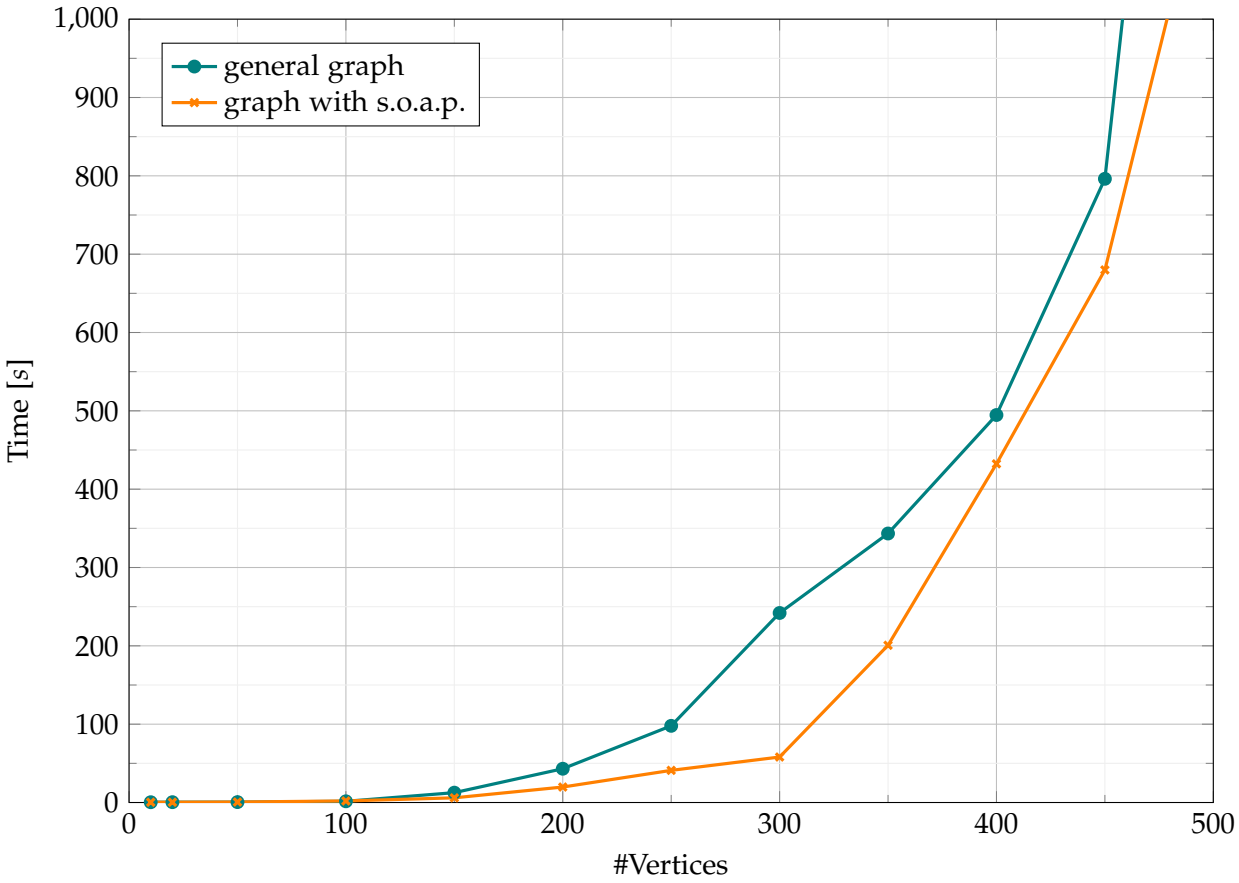

Figure 4.1: Simultaneously increasing vertices and degrees

4.4.2 The degree

In a separate experiment, we decrease the degree while the number of nodes is fixed to 100. This results in a better time for both. We can verify and conclude that the fewer edges the graph contains, the higher is the number of components. It becomes clear from table 4.2 that graphs with 100 vertices have a notable number of components if their degree is 10 or lower.

max degree	#Components	
	general	with s.o.a.p.
100	2	2
50	2	2
30	3	2
20	6	5
10	17	14
5	46	74
2	77	98
1	83	100
0	100	100

Table 4.2: #Components in relation to degree in a graph with 100 vertices

4.4.3 Experiment 2

We repeat our first experiment with a lower degree once in (a) with a constant maximal degree of 5 and secondly in (b) with a degree

$$\text{max degree} = \frac{\#\text{Vertices}}{10}.$$

The plots also includes the results of Experiment 1 to highlight the comparison.

(a)

#Vertices	Time [s]	
	general	with s.o.a.p.
10	0.37	0.33
50	0.43	0.37
100	0.94	0.94
200	13.34	9.12
500	516.48	240.48

Table 4.3: Comparison general graph and graph with the s.o.a.p. with constant max. degree 5

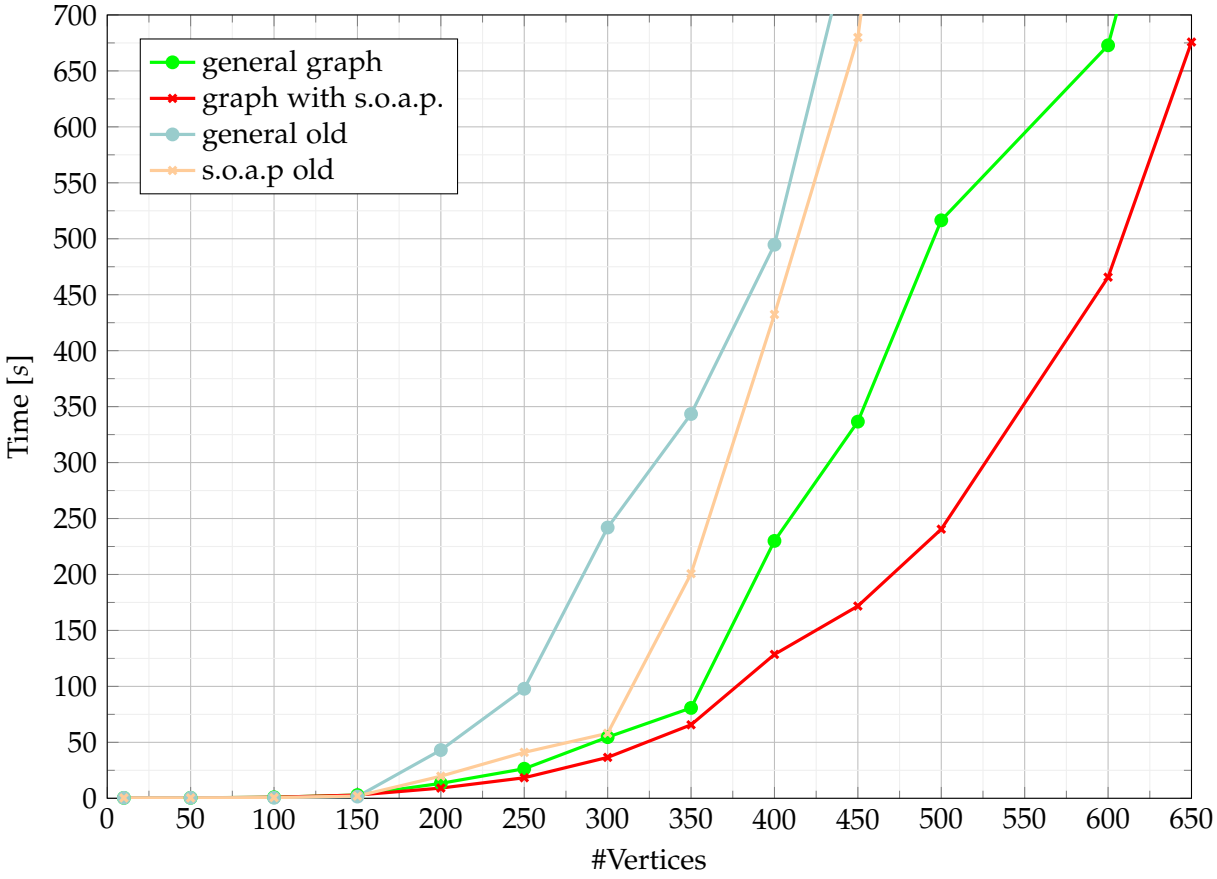

Figure 4.2: Comparison general graph and graph with the s.o.a.p. with constant maximal degree 5

(b)

#Vertices	max degree	Time [s]	
		general	with s.o.a.p.
10	1	0.34	0.33
50	5	0.39	0.38
100	10	1.31	0.97
200	20	10.80	10.17
500	50	315.48	300.57

Table 4.4: Testing results

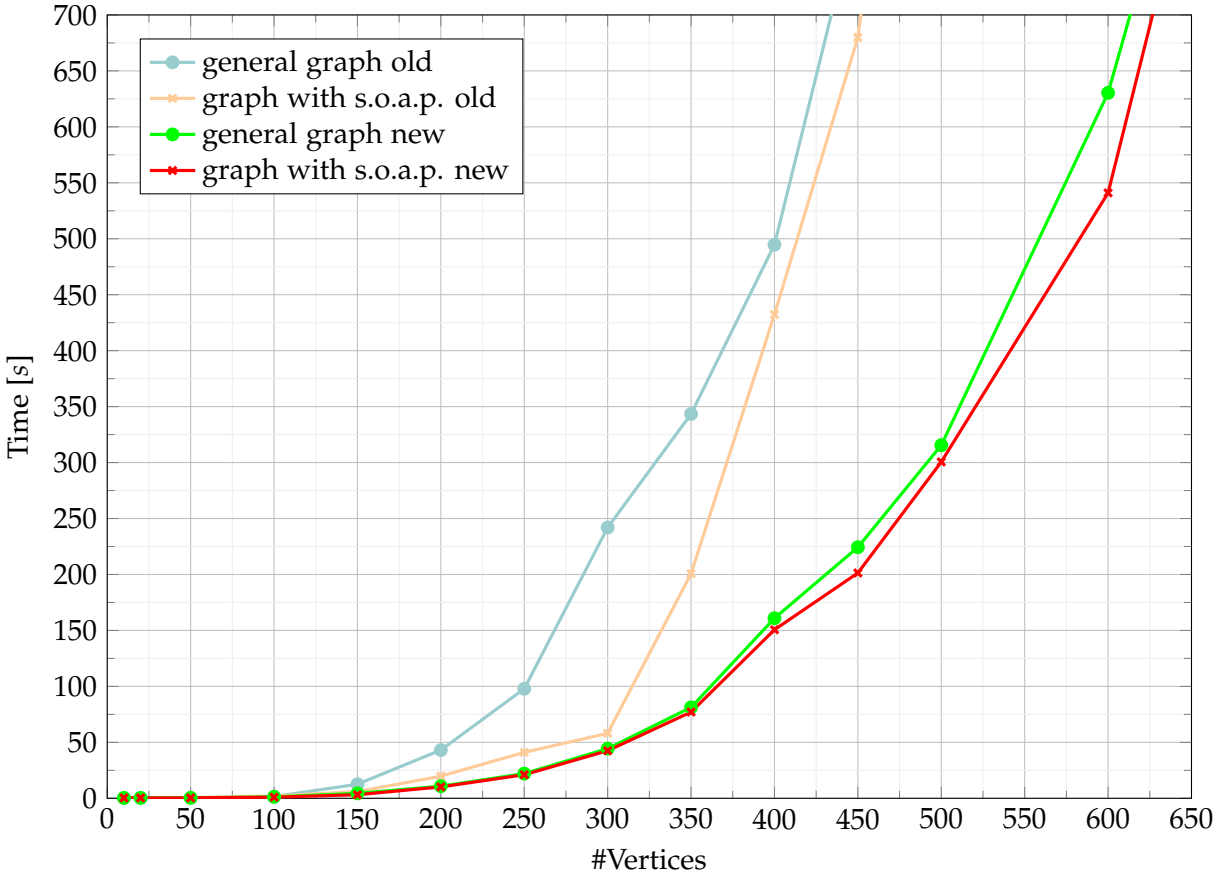

Figure 4.3: Comparison general graph and graph with the s.o.a.p. with degree = $\frac{\#Vertices}{10}$

The degree must be lowered enormously to see an effort and that limits the form of the graph. Even with a max degree

$$\text{max degree } = \frac{\#Vertices}{10} \qquad \text{or} \qquad \text{max degree } = \frac{\#Vertices}{20}$$

we receive at most two components. But when testing with a slightly higher degree it is noticeable that all vertices belong to the same winning set which is understandable because removing all obsolete edges resumes in cycles of a draw or belonging to the same player. Then if a circle consists of the whole graph, there is a big chance that it will end up in the same loop for all the vertices.

What initially appeared to be a disadvantage can perhaps be turned into an advantage and all games above a certain degree are said to be won by exactly one player. So a reduction to graphs in which only one player wins just like in other games.[7]

5 Conclusion

The aim of this thesis was the implementation of the new algorithm for solving mean payoff games with a signature of a potential. First attempts at theoretical runtime verification were made. The implementation solves mean payoff games both in a general form and with the signature of a potential. The graph generator creates random graphs to take as input. So far, tests regarding graph size were performed, so the number of nodes and edges have been changed to observe runtime behavior. The targeted values for the games of a signature of a potential are only achieved in small graphs. Also the expected difference between general graphs and signature of potential graphs was visible but minimal, which is probably related to the function of the obsolete edges. During testing it was noticed that their determination claims most of the time. In chapter 3.1 the obsolete edges were presented and the calculation of the $\eta-$ function was carried out. Note that the calculation takes n^3 steps to calculate a $n \times n$ matrix η for each node. Approaches to improve the runtime would be to check the function of the obsolete edges. During testing, the correlation between the number of degrees and the strongly connected components was also noticeable. Relatively slowly increasing degrees led to graphs with only one or two components which contradicts the idea of the algorithm. On the other hand, this could be a new consideration for finding winning sets. Considering that above a certain degree all vertices lie in the same winning set we would only needed to determine the winning set of one vertex.

Bibliography

[1] M. Akian, S. Gaubert, O. Lorscheid, and M. Mnich. Mean payoff games with signature of a potential. 2021.

[2] M. Bhojasia. URL `https://www.sanfoundry.com/`. access on 27.05.22.

[3] L. Brim, J. Chaloupka, L. Doyen, R. Gentilini, and J.-F. Raskin. Faster algorithms for mean-payoff games. *Formal Methods in System Design*, 38:97–118, 2011.

[4] A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8(2):109–113, 1979. ISSN 1432-1270. doi: 10.1007/BF01768705.

[5] J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly ltl verification more efficient. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 205–219, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24730-2.

[6] M. Kolmar. *Grundlagen der Spieltheorie*, pages 271–298. Springer Berlin Heidelberg, Berlin, Heidelberg, 2021. ISBN 978-3-662-63362-5. doi: 10.1007/978-3-662-63362-5_9. URL `https://doi.org/10.1007/978-3-662-63362-5_9`.

[7] C. Krishnendu, M. Henzinger, S. Krinninger, and D. Nanongkai. Polynomial-time algorithms for energy games with special weight structures. *Algorithmica*, 70(3):457–492, 2014. ISSN 1432-0541. doi: 10.1007/s00453-013-9843-7.

[8] P. Ohlmann. The GKK algorithm is the fastest over simple mean-payoff games. *CoRR*, abs/2110.04533, 2021. URL `https://arxiv.org/abs/2110.04533`.

[9] U. Zwick and M. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1):343–359, 1996. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(95)00188-3. URL `https://www.sciencedirect.com/science/article/pii/0304397595001883`.

List of Figures

List of Tables