Hybrid of Mean Payoff and Total Payoff

Andrzej Pacuk

Warsaw Univerisity

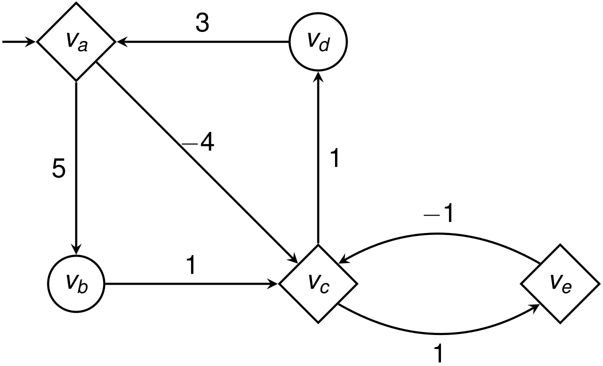
GAMES 2010

Outline

Outline

1. Definitions and Concepts
   - Total Payoff
   - Mean Payoff
   - Motivation
2. Properties
   - Determinacy
   - Counterexamples
3. Algorithm
   - Algorithm steps for minimizing player
   - Algorithm for maximising player
   - Limitations and properties

2-Player-Zero-Sum Infinite Games



Edge cost function $c\colon E \to \mathbb{Z}$

Total Payoff

Definition

Payoff for the infinite play $\pi = v_0 v_1 \ldots$*:*

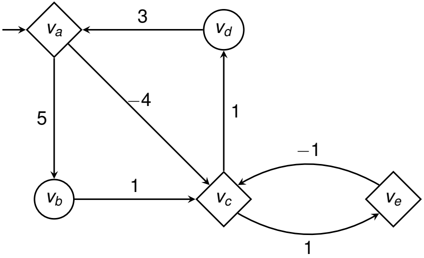
$$tp(\pi) = \liminf_{n \to \infty} \sum_{i=0}^{n-1} c(v_i, v_{i+1})$$

Total Payoff

Definition

Payoff for the infinite play $\pi = v_0 v_1 \ldots$:

$$tp(\pi) = \liminf_{n \to \infty} \sum_{i=0}^{n-1} c(v_i, v_{i+1})$$



- $tp((v_a v_b v_c v_d)^\omega) = +\infty$
- $tp(v_a (v_c v_e)^\omega) = -4$

Mean Payoff

Definition

Payoff for the infinite play $\pi = v_0 v_1 \ldots$:

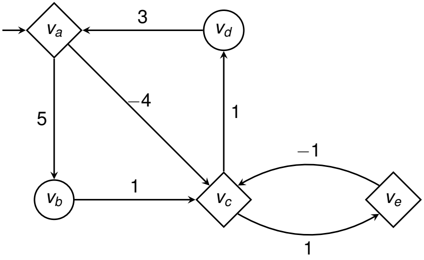
$$mp(\pi) = \liminf_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} c(v_i, v_{i+1})$$

Mean Payoff

Definition

Payoff for the infinite play $\pi = v_0 v_1 \ldots$:

$$mp(\pi) = \liminf_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} c(v_i, v_{i+1})$$



- $mp((v_a v_b v_c v_d)^\omega) = \frac{10}{4}$
- $mp(v_a (v_c v_e)^\omega) = 0$

Properties

Fact

*Both Mean Payoff and Total Payoff Games are positionally determined. (**Ehrenfeucht, Mycielski, Zielonka, Gimbert**)*

Lemma

For any initial position v, let:

$$\begin{aligned}\mu(v) &:= \textit{optimal play value in Mean Payoff Game starting at } v\\ \tau(v) &:= \textit{optimal play value in Total Payoff Game starting at } v\end{aligned}$$

*It holds (**Seidl**):*

$$\begin{aligned}\mu(v) < 0 &\quad \textit{iff} \quad \tau(v) = -\infty\\ \mu(v) = 0 &\quad \textit{iff} \quad \tau(v) \quad \textit{finite}\\ \mu(v) > 0 &\quad \textit{iff} \quad \tau(v) = +\infty\end{aligned}$$

Properties

Fact

*Both Mean Payoff and Total Payoff Games are positionally determined. (**Ehrenfeucht, Mycielski, Zielonka, Gimbert**)*

Lemma

For any initial position v, let:

$$\begin{aligned} \mu(v) &:= \textit{optimal play value in Mean Payoff Game starting at } v \\ \tau(v) &:= \textit{optimal play value in Total Payoff Game starting at } v \end{aligned}$$

*It holds (**Seidl**):*

$$\begin{array}{lll} \mu(v) < 0 & \textit{iff} & \tau(v) = -\infty \\ \mu(v) = 0 & \textit{iff} & \tau(v) \quad \textit{finite} \\ \mu(v) > 0 & \textit{iff} & \tau(v) = +\infty \end{array}$$

Hybrid Payoff

Definition

Hybrid Payoff games use the payoff mapping hp defined as follows:

$$hp(\pi) = \begin{cases} tp(\pi) & \textit{if } mp(\pi) = 0 \\ mp(\pi) & \textit{otherwise} \end{cases}$$

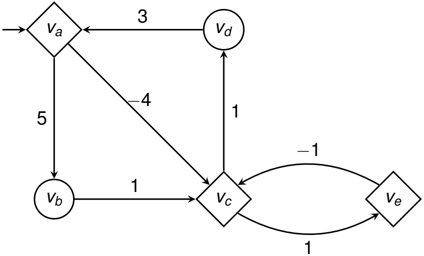
- for play $\pi_1 = (v_a v_b v_c v_d)^\omega$:
  $hp(\pi_1) = mp(\pi_1) = \frac{10}{4}$
- for play $\pi_2 = v_a(v_c v_e)^\omega$:
  $mp(\pi_2) = 0$, so
  $hp(\pi_2) = tp(\pi_2) = -4$

Hybrid Payoff

Definition

Hybrid Payoff games use the payoff mapping hp defined as follows:

$$hp(\pi) = \begin{cases} tp(\pi) & \textit{if } mp(\pi) = 0 \\ mp(\pi) & \textit{otherwise} \end{cases}$$



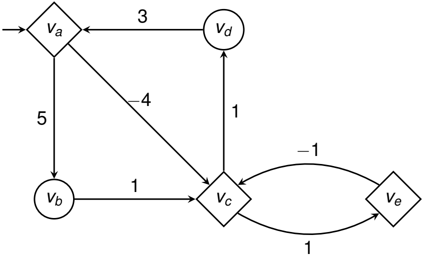
- for play $\pi_1 = (v_a v_b v_c v_d)^\omega$: $hp(\pi_1) = mp(\pi_1) = \frac{10}{4}$
- for play $\pi_2 = v_a(v_c v_e)^\omega$: $mp(\pi_2) = 0$, so $hp(\pi_2) = tp(\pi_2) = -4$

Hybrid Payoff

Definition

Hybrid Payoff games use the payoff mapping hp defined as follows:

$$hp(\pi) = \begin{cases} tp(\pi) & \textit{if } mp(\pi) = 0 \\ mp(\pi) & \textit{otherwise} \end{cases}$$



- for play $\pi_1 = (v_a v_b v_c v_d)^\omega$: $hp(\pi_1) = mp(\pi_1) = \frac{10}{4}$
- for play $\pi_2 = v_a(v_c v_e)^\omega$: $mp(\pi_2) = 0$, so $hp(\pi_2) = tp(\pi_2) = -4$

Outline

Theorem

(Determinacy) *Hybrid Payoff Games are determined.*

Proof.

It follows from the determinacy of Borel games, because function *hp* is Borel measurable (as a combination of Borel measurable functions *tp* and *mp*). □

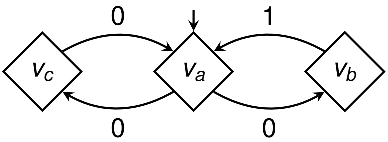
Theorem

(Determinacy) *Hybrid Payoff Games are determined.*

Proof.

It follows from the determinacy of Borel games, because function *hp* is Borel measurable (as a combination of Borel measurable functions *tp* and *mp*). □

The lack of positional determinacy



We have two positional plays starting from v_a:

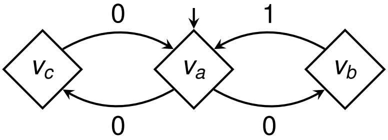
- $\pi_1 = (v_a v_b)^\omega$ assuring $hp(\pi_1) = 1/2$
- $\pi_2 = (v_a v_c)^\omega$ assuring $hp(\pi_1) = 0$

However, non-positional play:

$$\pi = (v_a v_b)(v_a v_c)^1(v_a v_b)(v_a v_c)^2(v_a v_b)(v_a v_c)^3 \ldots$$

gives optimal payoff $hp(\pi) = \infty$.

The lack of positional determinacy



We have two positional plays starting from v_a:

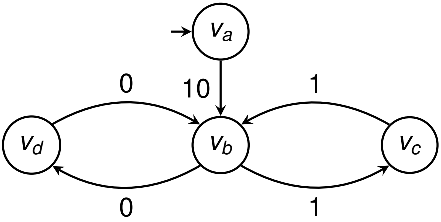
- $\pi_1 = (v_a v_b)^\omega$ assuring $hp(\pi_1) = 1/2$
- $\pi_2 = (v_a v_c)^\omega$ assuring $hp(\pi_1) = 0$

However, non-positional play:

$$\pi = (v_a v_b)(v_a v_c)^1(v_a v_b)(v_a v_c)^2(v_a v_b)(v_a v_c)^3 \dots$$

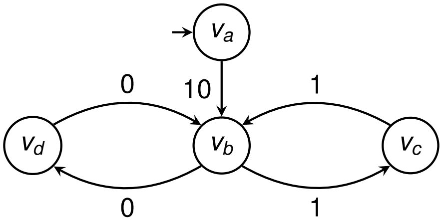
gives optimal payoff $hp(\pi) = \infty$.

The lack of optimal strategies



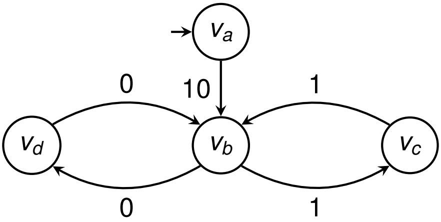
- There is no strategy assuring payoff $= 0$.
- For every positive $k \in \mathbb{N}$, a play $v_a\Big((v_b v_d)^{k-1} v_b v_c\Big)^{\omega}$ assures hybrid payoff (=mean payoff) $= \frac{1}{k}$.
- The optimal value of game $= 0$.

The lack of optimal strategies



- There is no strategy assuring payoff $= 0$.
- For every positive $k \in \mathbb{N}$, a play $v_a\Big((v_b v_d)^{k-1} v_b v_c\Big)^{\omega}$ assures hybrid payoff (=mean payoff) $= \frac{1}{k}$.
- The optimal value of game $= 0$.

The lack of optimal strategies



- There is no strategy assuring payoff $= 0$.
- For every positive $k \in \mathbb{N}$, a play $v_a\Big((v_b v_d)^{k-1} v_b v_c\Big)^{\omega}$ assures hybrid payoff (=mean payoff) $= \frac{1}{k}$.
- The optimal value of game $= 0$.

Outline

Algorithm

- The algorithm computes game values for instances of *Hybrid Payoff* games.
- But only for single player arenas.
- Cases of 0 and 1 player have to be treated separately.

Algorithm steps for minimizing player

1. Divide an arena into strongly connected components (SCC).
2. For each SCC, compute maximum (A) and minimum (B) *mean* of the cycle in that SCC.
3. For each SCC, depending on values (A, B) we compute game values for plays ending in that SCC
   (and finally game value := the lowest one):
   - If $A > 0$ and $B > 0$, then every cycle‘s mean is positive, so for every play π we have $hp(\pi) = mp(\pi)$. We return B.
   - If $A < 0$ and $B < 0$, then we symmetrically return B.

Algorithm steps for minimizing player

1. Divide an arena into strongly connected components (SCC).
2. For each SCC, compute maximum (A) and minimum (B) *mean* of the cycle in that SCC.
3. For each SCC, depending on values (A, B) we compute game values for plays ending in that SCC (and finally game value := the lowest one):
   - If $A > 0$ and $B > 0$, then every cycle's mean is positive, so for every play π we have $hp(\pi) = mp(\pi)$. We return B.
   - If $A < 0$ and $B < 0$, then we symmetrically return B.

Algorithm steps for minimizing player

1. Divide an arena into strongly connected components (SCC).
2. For each SCC, compute maximum (A) and minimum (B) *mean* of the cycle in that SCC.
3. For each SCC, depending on values (A, B) we compute game values for plays ending in that SCC
   (and finally game value := the lowest one):
   - If $A > 0$ and $B > 0$, then every cycle's mean is positive, so for every play π we have $hp(\pi) = mp(\pi)$. We return B.
   - If $A < 0$ and $B < 0$, then we symmetrically return B.

Algorithm steps for minimizing player

1. Divide an arena into strongly connected components (SCC).
2. For each SCC, compute maximum (A) and minimum (B) *mean* of the cycle in that SCC.
3. For each SCC, depending on values (A, B) we compute game values for plays ending in that SCC
(and finally game value := the lowest one):
    - If $A > 0$ and $B > 0$, then every cycle's mean is positive, so for every play π we have $hp(\pi) = mp(\pi)$. We return B.
    - If $A < 0$ and $B < 0$, then we symmetrically return B.

Algorithm steps for minimizing player

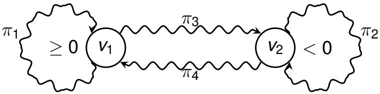
1. Divide an arena into strongly connected components (SCC).
2. For each SCC, compute maximum (A) and minimum (B) *mean* of the cycle in that SCC.
3. For each SCC, depending on values (A, B) we compute game values for plays ending in that SCC
(and finally game value := the lowest one):
    - If $A > 0$ and $B > 0$, then every cycle‘s mean is positive, so for every play π we have $hp(\pi) = mp(\pi)$. We return B.
    - If $A < 0$ and $B < 0$, then we symmetrically return B.

Continuation:

- If $A \geq 0$ and $B < 0$, then:

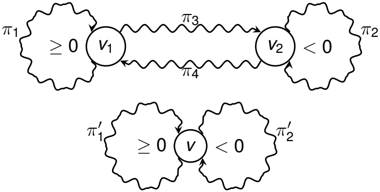
Continuation:

- If $A \geq 0$ and $B < 0$, then:



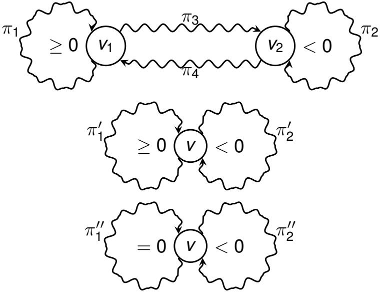
Continuation:

- If $A \geq 0$ and $B < 0$, then:



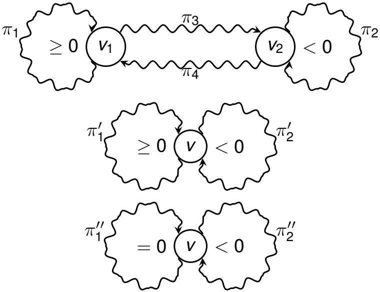
Continuation:

- If $A \geq 0$ and $B < 0$, then:



Continuation:

- If $A \geq 0$ and $B < 0$, then:



so one can generate strategy assuring payoff $= -\infty$.

Continuation:

- If $A \geq 0$ and $B = 0$, then:
  1. We detect all vertices laying on any 0-sum cycle in our SCC (call them *critical*) and compute the shortest paths from the initial vertex to critical vertices (using Bellman-Ford algorithm).
  2. If computing of shortest paths failed, this implies we have a path from initial vertex to our SCC accessing a negative cycle (in any previous SCC), so we can return $-\infty$.
  3. Otherwise, let C be the cost of the shortest path.
  4. If $A > 0$, we return $\min(C, 0)$, because we can approach to 0:
  5. Otherwise, we return C.

Continuation:

- If $A \geq 0$ and $B = 0$, then:
  1. We detect all vertices laying on any 0-sum cycle in our SCC (call them *critical*) and compute the shortest paths from the initial vertex to critical vertices (using Bellman-Ford algorithm).
  2. If computing of shortest paths failed, this implies we have a path from initial vertex to our SCC accessing a negative cycle (in any previous SCC), so we can return $-\infty$.
  3. Otherwise, let C be the cost of the shortest path.
  4. If $A > 0$, we return $\min(C, 0)$, because we can approach to 0:
  5. Otherwise, we return C.

Continuation:

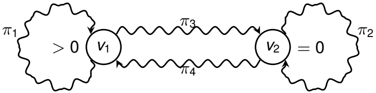
- If $A \geq 0$ and $B = 0$, then:
  1. We detect all vertices laying on any 0-sum cycle in our SCC (call them *critical*) and compute the shortest paths from the initial vertex to critical vertices (using Bellman-Ford algorithm).
  2. If computing of shortest paths failed, this implies we have a path from initial vertex to our SCC accessing a negative cycle (in any previous SCC), so we can return $-\infty$.
  3. Otherwise, let C be the cost of the shortest path.
  4. If $A > 0$, we return $\min(C, 0)$, because we can approach to 0:
  5. Otherwise, we return C.

Continuation:

- If $A \geq 0$ and $B = 0$, then:
  1. We detect all vertices laying on any 0-sum cycle in our SCC (call them *critical*) and compute the shortest paths from the initial vertex to critical vertices (using Bellman-Ford algorithm).
  2. If computing of shortest paths failed, this implies we have a path from initial vertex to our SCC accessing a negative cycle (in any previous SCC), so we can return $-\infty$.
  3. Otherwise, let C be the cost of the shortest path.
  4. If $A > 0$, we return $\min(C, 0)$, because we can approach to 0:
  5. Otherwise, we return C.

Continuation:

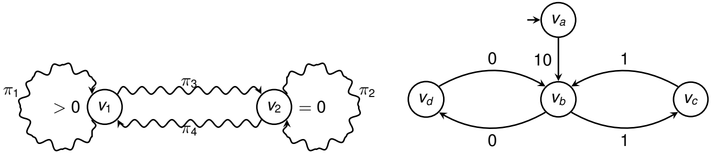
- If $A \geq 0$ and $B = 0$, then:
  1. We detect all vertices laying on any 0-sum cycle in our SCC (call them *critical*) and compute the shortest paths from the initial vertex to critical vertices (using Bellman-Ford algorithm).
  2. If computing of shortest paths failed, this implies we have a path from initial vertex to our SCC accessing a negative cycle (in any previous SCC), so we can return $-\infty$.
  3. Otherwise, let C be the cost of the shortest path.
  4. If $A > 0$, we return $\min(C, 0)$, because we can approach to 0:



  5. Otherwise, we return C.

Continuation:

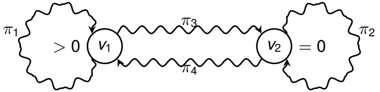
- If $A \geq 0$ and $B = 0$, then:
  1. We detect all vertices laying on any 0-sum cycle in our SCC (call them *critical*) and compute the shortest paths from the initial vertex to critical vertices (using Bellman-Ford algorithm).
  2. If computing of shortest paths failed, this implies we have a path from initial vertex to our SCC accessing a negative cycle (in any previous SCC), so we can return $-\infty$.
  3. Otherwise, let C be the cost of the shortest path.
  4. If $A > 0$, we return $\min(C, 0)$, because we can approach to 0:

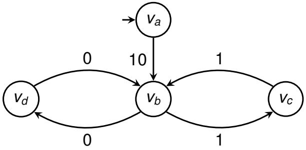


  5. Otherwise, we return C.

Continuation:

- If $A \geq 0$ and $B = 0$, then:
  1. We detect all vertices laying on any 0-sum cycle in our SCC (call them *critical*) and compute the shortest paths from the initial vertex to critical vertices (using Bellman-Ford algorithm).
  2. If computing of shortest paths failed, this implies we have a path from initial vertex to our SCC accessing a negative cycle (in any previous SCC), so we can return $-\infty$.
  3. Otherwise, let C be the cost of the shortest path.
  4. If $A > 0$, we return $\min(C, 0)$, because we can approach to 0:





  5. Otherwise, we return C.

Version for maximising player:

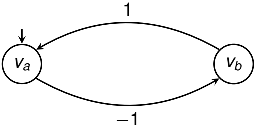
- Simple approach to reverse positions and edge weights, launch an algorithm for the other player and finally revert the returned value, may fail.
- It happens because:

$$\liminf -a_n = -\limsup a_n \neq -\liminf a_n$$

  for divergent sequences.
- This required aplication of modified version of Floyd-Warshall algorithm.

Version for maximising player:

- Simple approach to reverse positions and edge weights, launch an algorithm for the other player and finally revert the returned value, may fail.



- It happens because:

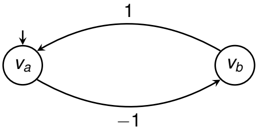
$$\liminf -a_n = -\limsup a_n \neq -\liminf a_n$$

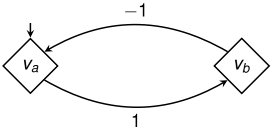
for divergent sequences.

- This required aplication of modified version of Floyd-Warshall algorithm.

Version for maximising player:

- Simple approach to reverse positions and edge weights, launch an algorithm for the other player and finally revert the returned value, may fail.





- It happens because:

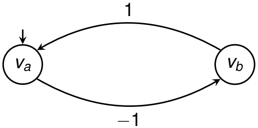
$$\liminf -a_n = -\limsup a_n \neq -\liminf a_n$$

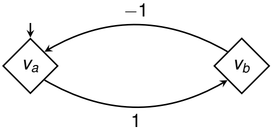
  for divergent sequences.

- This required aplication of modified version of Floyd-Warshall algorithm.

Version for maximising player:

- Simple approach to reverse positions and edge weights, launch an algorithm for the other player and finally revert the returned value, may fail.





- It happens because:

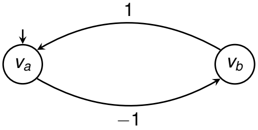
$$\liminf -a_n = -\limsup a_n \neq -\liminf a_n$$

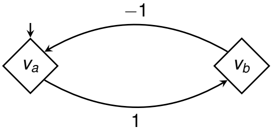
for divergent sequences.

- This required aplication of modified version of Floyd-Warshall algorithm.

Version for maximising player:

- Simple approach to reverse positions and edge weights, launch an algorithm for the other player and finally revert the returned value, may fail.





- It happens because:

$$\liminf -a_n = -\limsup a_n \neq -\liminf a_n$$

for divergent sequences.

- This required aplication of modified version of Floyd-Warshall algorithm.

The Algorithm:

- is efficient. Its time complexity is $O(V^3)$.
- computes games values, but not strategies.
- works only for single player arenas. The case of 2-player arenas remains open and seems to be hard.

The Algorithm:

- is efficient. Its time complexity is $O(V^3)$.
- computes games values, but not strategies.
- works only for single player arenas. The case of 2-player arenas remains open and seems to be hard.

The Algorithm:

- is efficient. Its time complexity is $O(V^3)$.
- computes games values, but not strategies.
- works only for single player arenas. The case of 2-player arenas remains open and seems to be hard.

Related works:

- H. Seidl, *Precise Program Analysis, Strategy Iteration and Games*, tutorial slides, Warsaw (Games 2008)
- K. Chatterjee, T.A. Henzinger, M. Jurdziński, *Mean-payoff parity games*
- D. Fischer, E. Grädel, Ł. Kaiser, *Model Checking Games for the Quantitative μ-Calculus*
- H. Gimbert, W. Zielonka, *Deterministic priority mean-payoff games as limits of discounted games*