SAPIENZA
UNIVERSITÀ DI ROMA

Games on Graphs a.y. 23-24

Energy Game Solver

Domiziano Scarcelli - scarcelli.1872664@studenti.uniroma1.it

May 13, 2024

SAPIENZA
UNIVERSITÀ DI ROMA

Contents

1 Introduction

In this report I discuss the methods used in order to implement an Energy Game solver in the context of the course ”Games on Graphs” held by Prof. Perelli at Sapienza University of Rome. The report is structured as follows:

- In Section 2 I introduce the theory related to the field of *games on graphs* that is needed in order to understand the problem and provide a valid solution;
- In Section 3, given the general theory behind the *games on graphs*, I introduce the particular case of *energy games*, including how is the problem formulated and what does it mean to solve such problem. I will also provide two high-level game solving algorithms, analyzing their time complexity.
- In Section 4 I discuss the technical implementation of the arena generation and game solving algorithms using Python.
- In Section 5 I present a performance evaluation over the two arena generation algorithms and the two game solving algorithms, with different arena sizes and densities.
- In Section 6 I draw some conclusions about the work.

2 Games on Graphs

Games on graphs is the field concerned with games whose rules and evolution are represented by a graph [1], and is a part of a larger research topic called *game theory*.

Let's now define which are the essential elements that constitute a game.

2.1 Arena

The arena is the place where the game is played, they have also been called game structures or game graphs. [1]

In turn based games, the set of vertices of the graph is divided into vertices controlled by each player. Since in *energy games* we have 2 players (Min and Max), we define the set of vertices $V = V_{\min} \cup V_{\max}$.

The set V of edges can be finite or infinite, in the first case we will have a *finite arena*, while in the second case an *infinite arena*.

2.2 Perfect vs Imperfect information

A game can be classified as a perfect or imperfect information. The fist means that all the players have a complete knowledge of the entire state of the game at all times, while in the latter they may lack some parts of the game state. Chess may be an example of a perfect information game, while Poker of an imperfect one.

2.3 Play

The game evolves through the interaction between the players, which consists in moving a token on the vertices of the arena. The player p who owns the vertex v where the token is can push the token along an edge $e = v \to v'$.

Initially, the token has to be on a certain vertex v. The continuous movement of the token on the vertices done by the player is a *play* (noted as π), which determines the path that the token travels.

Plays can be finite or infinite, but not empty.

2.4 Strategies

A *strategy* (also called *policy*) is a function that maps finite plays to edges, noted as $\sigma : \text{Paths} \to E$ where $\text{Paths} = \{\pi_0, \pi_1, \dots, \pi_n\}$ is a set of n plays, and defines a full description of the move that a player has to do for each possible setting of the game.

2.5 Conditions

A winning condition is what a player wants to achieve, which creates the purpose of the game. There are two types of conditions:

- *Qualitative*: it separates a set of winning plays $W \subseteq \text{Paths}$ from losing ones. If the play is winning then it *satisfies* W.
- *Quantitative*: it's a function $f : \text{Paths} \to \mathbb{R} \cup \{\pm\infty\}$ which assigns a real value (or $\pm\infty$) to each play. The value can be interpreted as a score.

2.6 Winning in qualitative games

A strategy σ for the player Eve is called *winning* from a vertex v in a qualitative game $\mathcal{G}$ if every play that starts from v consistent with σ is winning.

We call a *winning region* $W_{\text{Eve}}(\mathcal{G})$ the set of vertices v such that Eve wins $\mathcal{G}$ from v. A strategy that is winning from all vertices in $W_{\text{Eve}}(\mathcal{G})$ is called *optimal.* Note that winning regions are disjoint, meaning that for all qualitative games $\mathcal{G}$ we have $W_{\text{Eve}}(\mathcal{G}) \cap W_{\text{Adam}}(\mathcal{G}) = \emptyset$

2.7 Computational problems for qualitative games

For qualitative games, we identify different computational problems, which are the followings:

1. *Solving the game*: the input to the problem is a qualitative game $\mathcal{G}$ and a vertex v. The output is a boolean that tells us if Eve has a winning strategy from v.
2. *Computing the winning regions*: the input is a qualitative game $\mathcal{G}$, the output is the winning regions for Eve and Adam, respectively $W_{\text{Eve}}(\mathcal{G})$ and $W_{\text{Adam}}(\mathcal{G})$.
3. *Constructing a winning strategy*: the input is a qualitative game $\mathcal{G}$ and a vertex v, the output is a winning strategy σ for Eve from the vertex v.

2.8 Winning in quantitative games

Given a threshold $x \in \mathbb{R}$, a vertex v, a quantitative condition f and a quantitative game $\mathcal{G}$, we say that a strategy σ for the player Max *ensures* x from v if every play π starting from v consistent with σ satisfies $f(\pi) \geq x$.

Analogously, we say that a strategy τ for the player Min *ensures* x from v if every play π starting from v consistent with τ satisfies $f(\pi) \leq x$.

Let $\text{val}^{\mathcal{G}}_{\text{Max}}(v)$ denote the quantity

$$\sup_{\sigma} \inf_{\tau} f(\pi^{v}_{\sigma,\tau}) \in \mathbb{R} \cup \{\pm\infty\} \tag{1}$$

We define the value $\text{val}^{\sigma}_{\text{Max}}(v) = \inf_{\tau} f(\pi^{v}_{\sigma,\tau})$, which represents the best outcome that the player Max can ensure against any strategy of Min.

A strategy σ such that $\text{val}^{\sigma}_{\text{Max}}(v) = \text{val}^{\mathcal{G}}_{\text{Max}}(v)$ is called *optimal* from v. If it holds for all the vertices, then it's simply called *optimal.*

Equivalently, the strategy σ is called *optimal* from v if for every play π consistent with σ starting from v we have $f(\pi) \geq \text{val}^{\mathcal{G}}_{\text{Max}}(v)$

Note that there may not exist optimal strategies. There always exist an ϵ-optimal strategy for an $\epsilon > 0$ such that $\text{val}^{\sigma}_{\text{Max}}(v) = \text{val}^{\mathcal{G}}_{\text{Max}}(v) - \epsilon$.

2.9 Computational problems for quantitative games

For qualitative games, we identify different computational problems, which are:

1. *Solving the game*: the input to the problem is a quantitative game $\mathcal{G}$, a vertex v and a threshold $x \in \mathbb{R} \cup \{\pm\infty\}$. The output is a boolean that tells us if Max has a strategy ensuring x from v.
2. *Solving the value problem*: the input is the same as the *solving* problem, but the output is a boolean that tells us if $\text{val}^{\mathcal{G}} \geq x$. This differs from the *solving* problem when we drop the assumption that optimal strategies exist.
3. *Computing the value*: the input is a quantitative game $\mathcal{G}$ and a vertex v. The output is the value $\text{val}^{\mathcal{G}}$.
4. *Computing the value function*: the input is a quantitative game $\mathcal{G}$. The output is the value function $\text{val}^{\mathcal{G}} : V \to \mathbb{R} \cup \{\pm\infty\}$
5. *Constructing an optimal strategy*: the input is a quantitative game $\mathcal{G}$ and a vertex v, the output is the optimal strategy from v.

In this report we will explore two different types of the *value function computation* problem solutions on a type of quantitative games called *energy games*. Since the algorithm computes the value function, we will be able to get the solution of the game from each vertex $v \in V$. Given these elements, we can now define an *energy game* in a formal manner.

3 Energy Games

An *energy game* is a type of quantitative game played by 2 players usually called Min and Max, where the arena is defined as a tuple $\mathcal{A} = (G, V_{\min}, V_{\max})$ where $G = (V, E, In, Out)$ is a graph with vertices V, edges E and the functions $In, Out : E \to V$ that define *incoming* and *outgoing* vertices of edges.

The edge weight $w \in [-W, W]$ can be interpreted as an energy loss (in the case of *negative* weight) or energy gain (in the case of *positive* weight) for the player that moves the token on that particular edge.

The solution is the minimum amount of initial energy for the player Min in order to navigate the arena infinitely while maintaining a non-negative energy. Note that the game is a sum-zero game, meaning that the Max player objective is to maximize Min energy loss, while the Min objective is to minimize it.

Note that in order for a game to be solvable for Min, meaning that it can have an energy different from $+\infty$ in order to maintain a non-negative energy forever, the arena must not contain negative cycles, which are cycles where the sum of the edge weights is negative. The reason for this is trivial, since Max could enforce Min to stay in the negative cycle forever, and so it will require an infinite amount of energy. If the arena contains negative cycles, then Max will win each time.

3.1 Value iteration algorithm

Let $\mu : V \to \mathbb{R} \cup \{\pm\infty\}$ be a function that maps vertices to values, and δ the function $\delta(l, w) = \max(l - w, 0)$. Let also define an operator $\mathbb{O}^{\mathcal{G}}$ as following:

$$\mathbb{O}^{\mathcal{G}}(\mu)(u) = \begin{cases} \min\{\delta(\mu(v), w) : u \xrightarrow{w} v \in E\} & \text{if } u \in V_{\text{Min}}, \\ \max\{\delta(\mu(v), w) : u \xrightarrow{w} v \in E\} & \text{if } u \in V_{\text{Max}}. \end{cases} \tag{2}$$

Then we can define a value iteration algorithm that computes the value function μ for each vertex v in the arena.

Algorithm 1 The value iteration algorithm for energy games

```
1: Data: An energy game
2: for $u \in V$ do
3:     $\mu(u) \leftarrow 0$
4: repeat
5:     $\mu \leftarrow \mathbb{O}^{\mathcal{G}}(\mu)$
6: until $\mu = \mathbb{O}^{\mathcal{G}}(\mu)$
7: return $\mu$
```

Since for each vertex its value can only increase, and at each iteration at least on vertex increases, the total number of iterations in the `while` loop before reaching the fixed point is at most $O(nW)$, where n is the number of vertices and W is the max weight in the game.

Each computation of the $\mathbb{O}^{\mathcal{G}}(u)$ operator has a complexity of $O(|Out(u)| + |In(u)|)$, where $|Out(u)|$ and $|In(u)|$ define the number of outgoing and ingoing edges of u, which are controlled during the graph generation via the `edge_probability` variable (that we will call p) that determines the probability of two nodes having an edge between. In the worst case scenario we will have $p = 1$, meaning that $|Out(u)| = |In(u)| = n$, and a complexity of $O(n)$ per iteration. In the general case the complexity per iteration is $O(np)$.

We need to iterate this computation over all the n nodes, hence the complexity of each computation of the $\mathbb{O}^{\mathcal{G}}$ over all the nodes is $O(n^2p)$, which can also be written as $O(m)$ (the number of edges m is obtained by examining all the n^2 combinations of nodes, and creating an edge with a probability p).

Having a number nW of maximum steps of the *repeat*, the final complexity is $O(nmW)$.

3.2 Refined value iteration algorithm

We can be more efficent by not iterating the computation of the $\mathbb{O}^{\mathcal{G}}$ operator over all the n nodes at each iteration, but just on a smaller subset of *incorrect* vertices.

We define the following elements:

- A value Y for each vertex, representing the value function $\mu : V \to \mathbb{R}$
- A set of `Incorrect` vertices, and a temporary set of `Incorrect'` vertices.
- A table `Count`, which stores a number for each vertex of the player Min.

We call an edge $u \xrightarrow{w} v$ *incorrect* if $\mu(u) < \delta(\mu(v), w)$. A vertex belonging to Max is *incorrect* if it is has an outgoing edge which is incorrect. On the other hand, a vertex belonging to Min is *incorrect* if all of its outgoing edges are incorrect.

The pseudocode in algorithm 2 implements three functions called inside of the `Main` function.

- The `Init` function's purpose is to initialize the `Count` table and the `Incorrect` set by checking each vertex $v \in V_{\text{Min}}$ and $v \in V_{\text{Max}}$ with the condition to see if it's incorrect.
- The `Treat` function has the purpose to update the vertex value with the according formula.
- The `Update` function updates the `Count` table and the `Incorrect` set with the new nodes, since the values change after the `Treat` function call.

Even if the asymptotic complexity is the same, empirically the number of incorrect vertices is always less than n, which results in computing the same result as the *naive value iteration* algorithm, but each iteration will require fewer steps. (See performance evaluation results in Section 5).

Algorithm 2 The refined value iteration algorithm for energy games

```
function INIT()
    for $u \in V$ do
        $\mu(u) \leftarrow 0$
    for $u \in V_{\text{Min}}$ do
        for $u \xrightarrow{w} v \in E$ do
            if $incorrect : \mu(u) < \delta(\mu(v), w)$ then
                $\text{Count}(u) \leftarrow \text{Count}(u) + 1$
        if Count(u) = Degree(u) then
            Add $u$ to $Incorrect$
    for $u \in V_{\text{Max}}$ do
        for $u \xrightarrow{w} v \in E$ do
            if $incorrect : \mu(u) < \delta(\mu(v), w)$ then
                Add $u$ to Incorrect
function TREAT($u$)
    if $u \in V_{\text{Max}}$ then
        $\mu(u) \leftarrow \max \left\{ \delta(\mu(v), w) : u \xrightarrow{w} v \in E \right\}$
    if $u \in V_{\text{Min}}$ then
        $\mu(u) \leftarrow \min \left\{ \delta(\mu(v), w) : u \xrightarrow{w} v \in E \right\}$
function UPDATE($u$)
    if $u \in V_{\text{Min}}$ then
        Count(u) $\leftarrow 0$
    for $v \xrightarrow{w} u \in E$ which is incorrect do
        if $v \in V_{\text{Min}}$ then
            Count(v) $\leftarrow$ Count(v) + 1
            if Count(v) = Degree(v) then
                Add $v$ to Incorrect'
function MAIN()
    Init()
    for $i = 0, 1, 2, \ldots$ do
        Incorrect' $\leftarrow 0$
        for $u \in$ Incorrect do
            Treat(u)
            Update(u)
        if Incorrect' $= \emptyset$ then return $\mu$
        else Incorrect $\leftarrow$ Incorrect'
```

4 Implementation

The arena generation and the implementation of the two energy games algorithm was done using Python and **gravis**[2] for the graph visualization.

4.1 Arena Generation

We recall that a solvable *energy game* arena must not contain negative cycles. Because of this we need an algorithm that is able to construct such arena in a reasonable time even for an high number of nodes and edges, in order to later evaluate the algorithm on small, big, dense and less dense arenas.

The `Arena` class is responsible for the Arena modeling and generation. It contains the following attributes:

- `num_nodes`: used in the generation, it controls how many nodes the final graph should contain.
- `edge_probability`: used in the generation, it controls what is the probability that a node is connected to another node via an edge.
- `max_weight`: used in the generation, it controls the range $(-W, W)$ of edge weights.
- `nodes`: the set of nodes in the graph, represented as a set of integers.
- `edges`: the set of edges in the graph, represented as a tuple $(\text{origin}, \text{destination}, \text{weight})$.
- `value_mapping`: a dictionary that maps nodes to their value. It models the value function μ.
- `edges_mapping`: a dictionary that maps the origin of an edge to a list of its outgoing edges.
- `player_mapping`: a dictionary that maps nodes to their player, which can be `Player.MIN` or `Player.MAX`.
- `ingoing_edges`: a dictionary that maps nodes to their set of ingoing edges. It's built each time a new edge is added in the graph generation in order to be used efficiently in the *refined value iteration* algorithm.

The idea behind the generation of the arena without negative cycles is to iterate over all the possible tuples of nodes (u, v), and with a probability of `edge_probability` add an edge (u, v, w) with weight w sampled from $(-W, W)$ only if the edge (u, v, w) won't create a negative cycle, which we can check with the *Bellman Ford* algorithm.

The problem with the *Bellman Ford* algorithm taken as-it-is is the fact that it returns a boolean that indicates if the graph contains a negative cycle from any starting node, which requires to visit all the nodes in the graph and each edge. This is expensive, especially with graph with many nodes and connections, since we need to run this each time we wanto to add a new edge.

Let's analyze the *Bellman Ford* algorithm to see where we can make more efficient choices.

```python
def bellman_ford(self):
    """
    Detect negative cycles using Bellman-Ford algorithm.
    """
    distances = {node: 0 for node in nodes}
    # Relax edges repeatedly
    for _ in range(len(nodes) - 1):
        for edge in edges:
            if distances[edge[0]] + edge[2] < distances.get(edge[1], float('inf')):
                distances[edge[1]] = distances[edge[0]] + edge[2]
    # Check for negative cycles
    for edge in edges:
        if distances[edge[0]] + edge[2] < distances.get(edge[1], float('inf')):
            return True  # Negative cycle found
    return False
```

We can see that the algorithm initializes a **`distances`** dictionary to track the shortest distance for the node (key in the dictionary) and every other node in the graph. This operation has a complexity of $O(mn)$, since we need to iterate over all the nodes and over all the edges. Then the algorithm iterates over all the edges in order to find negative cycles, which has a complexity of $O(m)$. The overall complexity of the algorithm is $O(mn) + O(m) = O(mn)$.

Since each time we run the *Bellman Ford* algorithm we have only a new edge, we can check for negative cycles only related to the nodes and edges that are related to the new added edge, since that particular edge is the only cause for a plausible new negative cycle.

For this, we define a new class attribute that is **`fast_edges`**, which has a nested dictionary structure for efficiency, since it's going to be used mainly as a lookup table.

The keys of the dictionary are nodes, and the values represent a dictionary where the keys are nodes that are connected to their keys with an edge, which weight is stored in the associated value.

Here is an example of the structure for a graph with the follwing edges:

```
    # An edge is modelled as a tuple (origin: int, dest: int, weight: float)
    edges = [(0, 1, 0.3), (0, 2, 0.4), (2, 3, 0.9), (3, 2, -0.1)]
    fast_edges = {
        0: {1: 0.3, 2: 0.4},
        2: {3: 0.9},
        3: {2: -0.1}
    }
```

The algorithm is faster since it remembers the state of the **`distances`** array of the previous bellman ford calls and instead of relaxing all the edges n times, it just relaxes n times just the new edge. We then can iterate over all the outgoing edges from the nodes involved in the new edge (that we can lookup with the **`fast_edges`** data structure) and only check if a negative cycle is created on that subgraph.

```
def bellman_ford_incremental(self, new_edge: Tuple[int, int, float]):
    """
    A very efficient implementation of the Bellman-Ford algorithm that only checks
    for negative cycles related to the new edge.
    """

    # Add the new edge
    self.edges.add(new_edge)
    self.fast_edges[new_edge[0]][new_edge[1]] = new_edge[2]

    new_distance_0 = self.distances.get(new_edge[0], 0) + new_edge[2]
    new_distance_1 = self.distances.get(new_edge[1], float('inf'))

    previous_distance_1 = self.distances.get(new_edge[1], None)

    # Relax edges related to the new edge
    if new_distance_0 < new_distance_1:
        new_distance_1 = new_distance_0
    self.distances[new_edge[1]] = new_distance_1

    # This is a dictionary that maps the destination to the weight
    origin_to = self.fast_edges[new_edge[0]]

    # This is a dictionary that maps the origin to the weight
    dest_to = self.fast_edges[new_edge[1]]

    edges = {(new_edge[0], dest, weight) for dest, weight in origin_to.items()} |
            {(new_edge[1], origin, weight) for origin, weight in dest_to.items()}

    for edge in edges:
        if (self.distances[edge[0]] + edge[2] \
        < self.distances.get(edge[1], float('inf'))):

            self.edges.remove(new_edge)
            self.fast_edges[new_edge[0]].pop(new_edge[1])
            self.distances[new_edge[1]] = previous_distance_1
            return True
    return False  # No negative cycle found
```

The overall complexity of the algorithm is $O(|Out(v)| + |In(v)|) = O(np)$ (p is the probability of having an edge between two nodes), since the only for loop iterates over all the ingoing and outgoing edges of the origin node of the freshly added edge (input to the Bellman Ford algorithm).

Note that the result is not equivalent to the standard Bellman-Ford result, meaning that this faster algorithm can have some false positives on the negative cycle detection, but as far as what we need the algorithm for, it's not a problem if it means a much faster execution time.

In the performance evaluation (Section 5) we will see some examples in arena generation times with the standard *Bellman ford* algorithm and the optimized one.

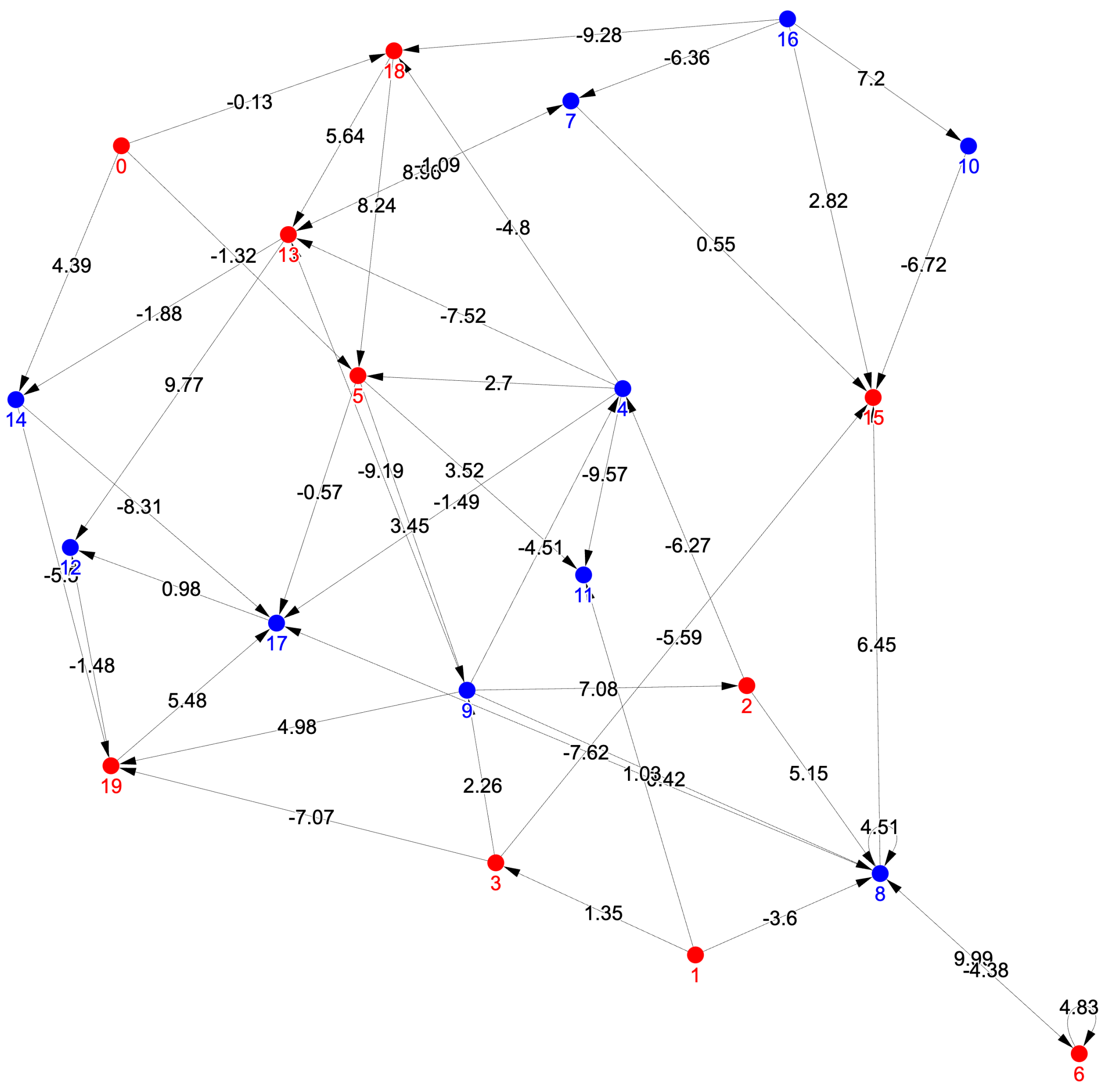

Figure 1: An example of an Arena with 20 edges and `edge_probability` of 0.2.

4.2 Naive value iteration algorithm

Regarding the value iteration algorithm, which pseudocode can be found at Algorithm 1, it's implemented as following:

We define a **Solver** class which is constructed based on an **Arena** instance.

```
import logging
from typing import Dict, Set
from tqdm import tqdm
from Graph import Arena, Player

def _delta(self,l, w):
    return max(l-w, 0)

def _O(self, node: int):
    """
    The O^G function which returns the max value between all the outgoing
    edges from the node (if player is Max), or the min value (if player is Min).
    """
    values = (self._delta(self.arena.value_mapping[v], w)
        for (u, v, w) in self.arena.get_outgoing_edges(node))

    if self.arena.player_mapping[node] == Player.MAX:
        return max(values, default=0)
    else:  # player is MIN
        return min(values, default=0)

def value_iteration(self):
    """
    The naive value iteration algorithm to compute the value function.
    """
    threshold = 0.000001
    steps = 0
    max_steps = 50_000
    pbar = tqdm(total=max_steps, desc="Value iteration")

    converged = {node: False for node in self.arena.nodes}
    while not all(converged.values()):
        converged = {node: False for node in self.arena.nodes}
        steps += 1
        if steps > max_steps:
            break
        pbar.update(1)
        for node in self.arena.nodes:
            old_value = self.arena.value_mapping[node]
            self.arena.value_mapping[node] = self._O(node)
            if abs(self.arena.value_mapping[node] - old_value) < threshold:
                converged[node] = True
    pbar.close()
```

```
        if steps > max_steps:
            logging.info(f"Did not converge after {steps} steps")
        else:
            logging.info(f"Converged after {steps} steps")
        return steps
```

The algorithm updates the node values until all of them converge, meaning that the difference of values before the update and after update is below a certain `threshold`, which is set to `1e-6`. Note that if the graph has at least one negative cycle, then the while loop will never stop. For this reason a `max_steps` of `50000` is added.

4.3 Refined value iteration algorithm

The *refined value iteration algorithm*, which pseudocode can be found at Algorithm 2 is implemented as following:

```
def optimized_value_iteration(self):
    incorrect: Set[int] = set()
    incorrect_prime: Set[int] = set()
    count: Dict[int, int] = {node: 0 for node in self.arena.nodes}

    def init():
        self.arena.value_mapping = {node: 0 for node in self.arena.nodes}
        pbar = tqdm(total=len(self.arena.nodes), desc="Opt Value Iteration - Init")
        # For each MIN node
        min_nodes = (n for n in self.arena.nodes
                     if self.arena.player_mapping[n] == Player.MIN)
        for node in min_nodes:
            pbar.update(1)
            for (u, v, w) in self.arena.get_outgoing_edges(node):
                # Node u is incorrect if each outgoing edge is incorrect
                if self.arena.value_mapping[u] < self._delta(self.arena.value_mapping[v
    ], w):
                    count[u] += 1
            # If count == degree of node
            if count[node] == self.arena.get_node_degree(node):
                incorrect.add(node)
        # For each MAX node
        max_nodes = (n for n in self.arena.nodes
                     if self.arena.player_mapping[n] == Player.MAX)
        for node in max_nodes:
            pbar.update(1)
            for (u, v, w) in self.arena.get_outgoing_edges(node):
                # Node u is incorrect
                if self.arena.value_mapping[u] < self._delta(self.arena.value_mapping[v
    ], w):
                    incorrect.add(u)

    def treat(u: int):
        self.arena.value_mapping[u] = self._O(u)
    def update(u: int):
        if self.arena.player_mapping[u] == Player.MIN:
            count[u] = 0
        for (v, _, w) in self.arena.ingoing_edges.get(u, set()):
            # Only consider nodes that are incorrect
            if not (self.arena.value_mapping[v] < self._delta(self.arena.value_mapping[u
], w)):
                continue

            if self.arena.player_mapping[v] == Player.MIN:
                count[v] += 1
                if count[v] == self.arena.get_node_degree(v):
                    incorrect_prime.add(v)
            if self.arena.player_mapping[v] == Player.MAX:
                incorrect_prime.add(v)

    init()
    n = len(self.arena.nodes)
    W = self.arena.max_weight
    max_steps = n * W
    steps = 0
    for i in tqdm(range(max_steps)):
        steps += 1
        incorrect_prime = set()
        for u in incorrect:
            # Complexity is O(m) where m is the number of edges
            treat(u)
            update(u)
        if incorrect_prime == set():
            print(f"Converged after {i} steps")
            return steps
        incorrect = incorrect_prime
    return steps
```

5 Performance Evaluation and Results

In this section we will evaluate the performance of the following tasks:

- Simple arena generation;
- Arena generation without negative cycles with standard Bellman Ford algorithm;
- Arena generation without negative cycles with the incremental version of the Bellman Ford algorithm;
- Energy games solver with the Naive Value Iteration algorithm;
- Energy games solver with the Refined Value Iteration algorithm;

All tests will be performed using different parameters, to see how the performances of the different algorithms change when increasing the graph complexity.

5.1 Evaluation of Arena Generation

For each setting (number of nodes and edge probability), a total of 5 tests were conducted to obtain an average time of execution.

The table below contains:

- The number of nodes
- The edge probability
- The time to generate an arena without checks on the negative cycles;
- The time to generate an arena checking negative cycles with Bellman Ford
- The time to generate an arena checking negative cycles with the Optimized Bellman Ford algorithm.

Arena Generation				
Number of nodes	Edge probability	Time (no checks)	Time (BF)	Time (Optimized BF)
100	0.1	8.80 ms	27.43 ms	15.13 ms
100	0.2	7.39 ms	42.31 ms	14.38 ms
100	0.3	9.29 ms	57.22 ms	19.54 ms
100	0.4	8.31 ms	72.40 ms	25.35 ms
100	0.5	11.57 ms	88.39 ms	34.94 ms
200	0.1	10.53 ms	133.68 ms	20.90 ms
200	0.2	12.71 ms	252.16 ms	44.88 ms
200	0.3	15.09 ms	365.00 ms	70.19 ms
200	0.4	17.11 ms	490.65 ms	108.65 ms
200	0.5	19.67 ms	603.05 ms	158.93 ms
300	0.1	18.54 ms	413.68 ms	46.43 ms
300	0.2	22.73 ms	824.86 ms	113.85 ms
300	0.3	30.61 ms	1.21 sec	202.74 ms
300	0.4	32.01 ms	1.62 sec	333.74 ms
300	0.5	45.70 ms	2.00 sec	489.24 ms
400	0.1	28.15 ms	1.03 sec	92.53 ms
400	0.2	36.78 ms	1.98 sec	236.87 ms
400	0.3	55.51 ms	2.90 sec	446.53 ms
400	0.4	59.05 ms	3.98 sec	749.61 ms
400	0.5	68.06 ms	4.82 sec	1.13 sec
500	0.1	41.80 ms	1.95 sec	154.65 ms
500	0.2	62.84 ms	3.84 sec	463.09 ms
500	0.3	76.73 ms	5.58 sec	847.26 ms
500	0.4	87.64 ms	7.49 sec	1.39 sec
500	0.5	98.93 ms	9.38 sec	2.18 sec
1000	0.1	161.31 ms	15.41 sec	913.22 ms
1000	0.2	214.72 ms	29.93 sec	3.06 sec
1000	0.3	281.77 ms	44.60 sec	5.88 sec
1000	0.4	333.74 ms	59.64 sec	10.57 sec
1000	0.5	392.45 ms	1.24 min	18.48 sec
2000	0.1	616.58 ms	2.11 min	6.18 sec
2000	0.2	871.34 ms	4.01 min	22.53 sec
2000	0.3	1.10 sec	6.00 min	45.92 sec
2000	0.4	1.32 sec	8.03 min	1.36 min
2000	0.5	1.57 sec	9.93 min	2.14 min
5000	0.1	3.98 sec	32.20 min	1.51 min
5000	0.2	5.49 sec	62.57 min	5.44 min
5000	0.3	7.33 sec	94.75 min	11.73 min
5000	0.4	8.90 sec	125.82 min	20.78 min
5000	0.5	11.14 sec	155.92 min	33.99 min

From this table we can see how the *Incremental Bellman Ford* algorithm is much faster than the original version, and how for a large number of nodes and edges the generation times increase exponentially, due to the many checks that the algorithm has to do in order to not insert edges that will create negative cycles.

5.2 Evaluation of the value iteration algorithm

The table below contains the comparison of time to solve the game with the Naive and Refined value iteration algorithms.

Solving the game			
Number of nodes	Edge probability	Time (naive)	Time (optimized)
100	0.1	3.65 ms	3.28 ms
100	0.2	4.49 ms	4.53 ms
100	0.3	6.68 ms	4.83 ms
100	0.4	7.15 ms	7.75 ms
100	0.5	8.58 ms	7.09 ms
200	0.1	7.42 ms	5.46 ms
200	0.2	14.09 ms	10.01 ms
200	0.3	21.12 ms	16.05 ms
200	0.4	32.17 ms	26.29 ms
200	0.5	39.86 ms	32.95 ms
300	0.1	17.05 ms	15.46 ms
300	0.2	31.63 ms	35.43 ms
300	0.3	58.42 ms	45.91 ms
300	0.4	99.49 ms	82.58 ms
300	0.5	161.07 ms	113.33 ms
400	0.1	33.19 ms	26.94 ms
400	0.2	81.50 ms	64.60 ms
400	0.3	151.79 ms	110.84 ms
400	0.4	230.66 ms	172.51 ms
400	0.5	361.59 ms	285.20 ms
500	0.1	58.26 ms	43.85 ms
500	0.2	142.15 ms	111.85 ms
500	0.3	365.51 ms	291.57 ms
500	0.4	513.59 ms	429.98 ms
500	0.5	638.67 ms	505.26 ms
500	1.0	1.85 sec	1.37 sec
1000	0.1	309.54 ms	225.22 ms
1000	0.2	1.03 sec	886.58 ms
1000	0.3	1.94 sec	1.47 sec
1000	0.4	3.79 sec	2.63 sec
1000	0.5	4.95 sec	4.21 sec
2000	0.1	2.35 sec	1.73 sec
2000	0.2	8.46 sec	7.04 sec
2000	0.3	15.62 sec	14.01 sec
2000	0.4	26.91 sec	23.65 sec
2000	0.5	37.48 sec	32.75 sec
5000	0.1	39.75 sec	33.81 sec
5000	0.2	2.24 min	2.15 min
5000	0.3	4.48 min	4.42 min
5000	0.4	8.72 min	7.65 min
5000	0.5	15.11 min	13.17 min

6 Conclusions

In this document we've reviewed the games on graphs theory that is necessary to understand in order to develop an energy game solver. We've seen that it's possible to generate an arena that doesn't contain negative cycles in a reasonable time if the number of nodes and the density is not too high, and that we can solve the game finding the initial value needed for the player Min to maintain a non-negative energy forever starting from each vertex.

References

[1] Games on Graphs

[2] gravis github.com/robert-haas/gravis