SAPIENZA
UNIVERSITÀ DI ROMA

Games on Graphs a.y. 23-24

AFC Report

Alessio Lucciola - lucciola.1823638@studenti.uniroma1.it

November 17, 2023

Contents

1 Introduction

This report aims to briefly describe the work that I carried out for the AFC Games On Graphs course held by prof. Giuseppe Perelli. Games on graphs refer to a category of mathematical and computational models that involve strategic interactions among agents situated on a graph. These games have applications in various fields, including computer science, operations research, and game theory as they provide a framework for studying strategic interactions and decision-making in complex networked systems. The course mainly focused on the theoretical aspects of Game on Graphs and one the topic was reachability games. For this reason, I decided to implement an algorithm to solve this type of game and test its performance. The algorithm will be exposed in the next chapters.

2 Background

Before explaining what Reachability Games are, let's give some background information on how Games on Graphs work. First of all, a **graph G** is a collection of **nodes** (or vertices) V and edges E that connect pairs of nodes. Graphs can be classified into various types based on their characteristics. Some common types of graphs include:

- Undirected Graph: A graph in which edges have no direction.
- Directed Graph: A graph in which edges have a direction.
- Weighted Graph: A graph in which edges have weights.
- Connected Graph: A graph in which there is a path between every pair of nodes.
- Disconnected Graph: A graph with at least two nodes that do not have a path connecting them.

In particular, A (directed) graph is given by a set V of vertices and a set E of edges given by the functions In,Out : E $\rightarrow$ V: for an edge e we write In(e) for the incoming vertex and Out(e) for the outgoing vertex. We say that e is an outgoing edge of In(e) and an incoming edge to Out(e). To introduce an edge, it is convenient to write e = v $\rightarrow$ v_0 to express that v = In(e) and v_0 = Out(e). A path π is a finite or infinite sequence:

$$\pi = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow ...$$

A **Game** is played over a (finite) graph (V, E), whose vertices are under the control of the two agents (or **players**) V = $V_0 \cup V_1$. This is to say that the set of vertices is divided into vertices controlled by each player. We represent vertices in V_0 by circles, and vertices in V_1 by squares.

Let's go into details on how a game is played. A **token** moves along the vertices and is sent to a successor by the controlling player. The interaction between the two players consists in moving the token on the vertices of the arena. The token is initially on some vertex. When the token is in some vertex v, the player who controls the vertex chooses an edge e = v $\rightarrow$ v' and pushes the token along this edge to the next vertex v'. The **outcome** (or play) is an infinite sequence of vertices in the graph. The game is finite meaning that there are finitely many vertices and edges.

A strategy for a player is a full description of his or her moves in all situations. Formally, a strategy is a function mapping finite plays to edges:

$$\sigma : Paths \rightarrow E$$

We say that a play $\pi = v_0 \rightarrow v_1 \rightarrow ...$ is consistent with the strategy σ of Player 0 if for all i such that $v_i \in V_0$ we have $\sigma(\pi \leq i) = v_i \rightarrow v_{i+1}$.

A **winning condition** (or objective) is a subset Obj $\subseteq V_\omega$ of plays that Player 0 wants to occur. In the same way, we can also define the same condition for Player 1. In this case we talk about **qualitative conditions**.

In a game there may be different **objectives** and the one I focused on during the AFC was **Reachability**.

3 Theory on Reachability Games

In order to explain how Reachability works, let's first define the **safety** objective that is the simplest qualitative condition. Let's assume we have Obj $\subseteq$ V the winning condition. Safe(T) can be defined like this:

$$Safe(T) = \pi \in V_\omega : \forall i \in N, \pi[i] \in Obj$$

Assuming we are playing as Player 0, the formula above means that we have to always stay in Obj. So the strategy of Player 0 is to find a path such that they are always in Obj.

The dual of the safety objective is the **Reachability objective**:

$$Reach(T) = \pi \in V_\omega : \exists i \in N, \pi[i] \in Obj$$

Here the strategy of Player 0 is to reach one node in Obj at least once. Let's go more into details:

- Player 0's objective is to reach a set of target nodes Obj. Player 1's objective is to prevent the protagonist from reaching the goal.
- The game progresses with alternating moves between the Player 0 and Player 1. It continues until either Player 0 reaches the goal (proving the reachability) or Player 1 successfully blocks all paths (proving unreachability).

- Player 0 wins if they can reach the specified target nodes Obj despite Player 1's efforts. Otherwise we consider Player 1 the winner of the game.

Let's now define the force function that is used to solve reachability games:

$$force_0(X) = \{v \in V_0 : E(v) \cap X \neq \emptyset\} \cup \{v \in V_1 : E(v) \subseteq X\}$$

The force function defines the set of vertices from which Player 0 can enforce the token to move into the subset X of vertices, while Player 1 cannot avoid entering X. The first part of the function selects vertices from V_0 for which the set of edges incident to the vertex intersects with X. In other words, Player 0 can force the token to move into X from these vertices. The second part selects vertices from V_1 for which the set of edges incident to the vertex is completely contained within X. This means that Player 1 cannot avoid entering X from these vertices.

Let's try to explain inductively how this force function can be used to solve reachability games. We define:

$$Reach^{\mathrm{n}}(Obj) = \text{Player 0 can reach Obj in at most n moves}$$

Inductive proof:

- Base Case (n=0):

  $$Reach_0(Obj) = Obj$$

  This states that if n=0, then the set of vertices that Player 0 can reach in at most 0 moves is Obj, meaning Player 0 has to be in Obj already.

- Inductive Step (n>0):

  $$Reach_n(Obj) = Reach_{n-1}(Obj) \cup force_0(Reach_{n-1}(Obj))$$

  This states that if n>0, then the set of vertices that Player 0 can reach in at most n moves is the union of the set of vertices that Player 0 can reach in at most n-1 moves and the set of vertices from which Player 0 can enforce movement into.

Finally, to solve reachability we have to define the union of the sets of vertices that Player 0 can reach in at most 0 moves, 1 move, 2 moves, and so on:

$$Win_0(G) = Reach(Obj) = React_0(Obj) \cup React_1(Obj) \cup ...$$

It can be proved that every 2-player turn-based reachability game is **determined** meaning that V = $\mathrm{Win}_0(\mathrm{G}) \cup \mathrm{Win}_1(\mathrm{G})$. So, once computed the winning region of Player 0, we can easily compute its losing region (or the winning region of Player 1) as $\mathrm{Win}_1(\mathrm{G}) = \mathrm{V} - \mathrm{Win}_0(\mathrm{G})$.

4 Implementation of a Reachability Games Solver

Let's finally see the implementation of the Reachability games solver. The implementation of the algorithm was done in Python.

4.1 Graph Generation Algorithm

In order to solve reachability games, a game has to be generated first. In order to do so, I created a Python class to represent the graph. This class takes these values in input:

- n_nodes: A natural number that represents the number of vertices in the game.
- edge_probability: The probability for an edge to be generated from a vertex to another.
- n_winning: The percentage of winning nodes (Obj set) in the graph with respect to the total vertices.

So, I create a graph with n_nodes vertices and immediately add an outgoing edge from each vertex. This is to avoid that there is a sink node in the graph. Then, for each vertex in the graph I randomly add an edge to all the other edges with an edge_probability. For example, assuming that edge_probability=0.3 and I have to decide if an edge from v_1 to v_2 has to be added, I randomly select a number k in the range [0, 1]. If k $\leq$ edge_probability, I add the edge from the vertex v_1 to v_2, otherwise I don't.
In order to slightly simplify the graph creation, I assign nodes with an even index to Player 0 and nodes with an odd index to Player 1. Then, I select the winning condition Obj of Player 0 (the nodes to be reached) as a n_winning percentage with respect to the total vertices. So, for example, if Player 0 has 20 nodes and n_winning = 0.2 then if means that $\|Obj\| = 20 * 0.2 = 4$ winning nodes that are assigned randomly among the vertices of Player 0. Moreover, vertices of Player 0 are represented with circles while vertices of Player 1 with squares and winning nodes are red (instead of blue). Here the partial code that creates the graph:

```
1  import math
2  import random
3  import networkx as nx
4  import matplotlib.pyplot as plt
5
6  class Graph():
7      def __init__(self, n_nodes, edge_probability, n_winning, mode):
8          super(Graph, self).__init__
9          self.graph = nx.DiGraph()
10         self.mode = mode
11
```

```
        # Create the nodes
        for i in range(n_nodes):
            self.graph.add_node(i)

        # Add at least an edge that leaves from each node
        for i in range(1, n_nodes):
            random_node = random.choice(range(i))
            self.graph.add_edge(i, random_node)

        # Add the other edges randomly
        for i in range(n_nodes):
            for j in range(i+1, n_nodes):
                if random.random() < edge_probability and (not self.graph.has_edge(i, j) and
                ↪  not self.graph.has_edge(j, i)):
                    self.graph.add_edge(i, j)

        # Split nodes between the two players
        self.player0_nodes = [node for node in range(n_nodes) if node % 2 == 0]
        self.player1_nodes = [node for node in range(n_nodes) if node not in
        ↪  self.player0_nodes]

        # Define the winning nodes
        n_winning = max(1, math.floor(self.player0_nodes*n_winning)) # Add winning nodes with
        ↪  "n_winning probability"
        winning_nodes = random.sample(self.player0_nodes, n_winning)
        attributes = {}
        for n in self.graph.nodes:
            if n in winning_nodes:
                attributes[n] = {'winning': True}
            else:
                attributes[n] = {'winning': False}
        nx.set_node_attributes(self.graph, attributes)

        # Color the winning node in the graph
        if mode == 'reachability':
            node_colors = {node: 'red' if node in winning_nodes else 'blue' for node in
            ↪  self.player0_nodes}
            nx.set_node_attributes(self.graph, node_colors, 'color')

        # Create a dictionary for labeling the nodes with their numbers
        self.node_labels = {node: str(node) for node in self.graph.nodes}
```

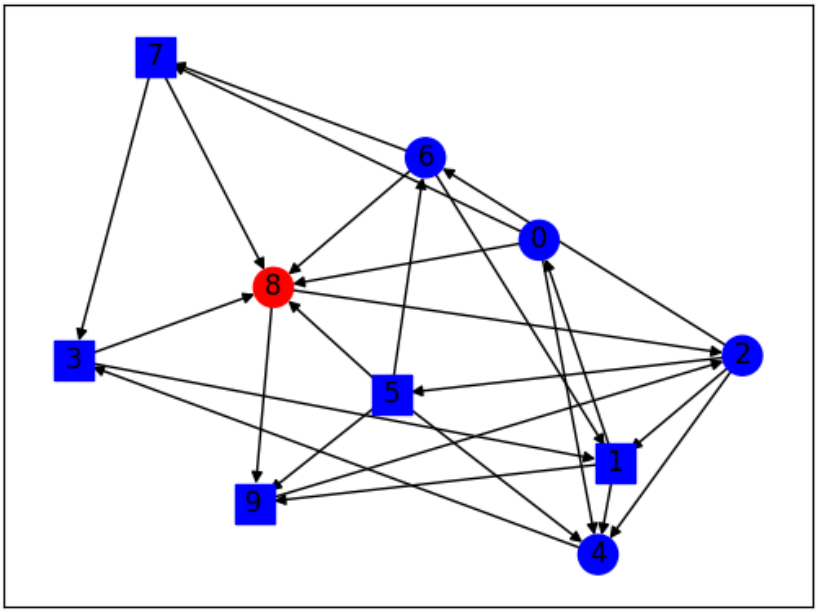

Figure 1: An example of graph created by the class defined above. The configuration used is: n_nodes=10, edge_probability=0.5, n_winning=0.1

4.2 Reachability Solver

The Reachability Solver algorithm first uses the code defined above to generate a game. This is the graph that is going to be used for solving the game. Some variables are defined and are used throughout the game to store useful data:

- queue: Set that stores the vertices to be visited recursively.
- strategy: Dictionary to store the Player's 0 strategy. Both keys and values are an index of a node. Basically if we have a pair key=v_0, value=v_1 it means that if Player 0 is in node v_0, they should move to v_1 to eventually reach a winning node in Obj.
- win_region: The force set in which Player 0 can enforce the token to move into the subset X of vertices, while Player 1 cannot avoid entering X.
- lose_region: Simply computed as the difference between the vertices in the graph and the win region.
- recursion_count: Counter for the recursions done by the algorithm.

The algorithm works like this:

1. For each winning node w in Obj, add w to the queue;
2. While the queue contains at least an element, retrieve the first one added and remove it from the queue. Let's call this element n;
3. For each predecessor of n (we call predecessor a vertex n_{pred} that has a direct edge from n_{pred} to n), compute the force function:
   (a) If n_{pred} is a vertex of Player 0 and it is not in the winning region (here the winning region is X in the force function formula defined above) then add it to the winning region and define a strategy strategy $[n_{pred}] = n$. Also add n_{pred} to the queue so that we will process it in the next epochs.
   (b) If n_{pred} is not a vertex of Player 0 and it is not in the winning region, we have to check that all its successors are in the winning region (this is the second condition of the force function). If so then add it to the winning region and define a strategy strategy $[n_{pred}] = n$. Also add n_{pred} to the queue so that we will process it in the next epochs.

This is an example output of the game presented in Figure 1:

```
Winning nodes: [8]
Strategy: {0: 8, 6: 8, 2: 6, 9: 2}
Winning region: {0, 2, 6, 8, 9}
Losing region: {1, 3, 4, 5, 7}
```

Let's finally present the algorithm of the Reachability Games Solver:

```python
import graph_creation as gc

def reachability_game(n_nodes=10, edge_probability=0.5, n_winning=0.1, visualize=True,
    print_info=True):
    G = gc.Graph(n_nodes, edge_probability, n_winning, mode='reachability')
    W = G.get_winning_nodes()
    if print_info:
        print("Winning nodes: " + str(W))
    strategy, win_region, recursion_count = solve_reachability(G, W, print_info)
    if visualize:
        G.visualize_graph()

    return strategy, win_region, recursion_count
```

```
13
14 """
15     This function takes a direct graph and a set terminal nodes and solve reachability games.
16
17     :param G: A direct graph.
18     :param W: A (non-empty) set of terminal (winning) nodes.
19     :return: strategy: A dictionary where the key is the current node and the value is a
   ↪  successor that, if chosen, allows player 0 to stay in the winning region.
20     :return: win_region: A set of nodes from which player 0 can force player 1 to a winning
   ↪  node.
21     :return: recursion_count: The number of recursions done by the algorithm to terminate.
22 """
23 def solve_reachability(G, W, print_info):
24     queue = set() # Add nodes to be visited yet at execution time
25     strategy = {} # Keep track of a possible strategy for player 0 to reach a winning node
26     win_region = set(W) # Immediatly set the winning node of player 0 in the winning region
27     lose_region = set() # Initialize the losing region as an empty set
28     recursion_count = 0
29
30     for w in W: # For each winnining node
31         queue.add(w) # Add a winning node to the queue
32
33         while queue:
34             recursion_count += 1  # Increment the recursion count
35             n = queue.pop() #Retrieve the first element from the queue
36             for pred in G.get_predecessors(n): # If the predecessor is of player 0 then ->
               ↪  add pred in win_region X
37                 if pred in G.player0_nodes:
38                     if (pred not in win_region):
39                         queue.add(pred)
40                         win_region.add(pred)
41                         strategy[pred] = n
42                 else: # If the predecessor is of player 1 (opponent) and all the successors
                   ↪  are in the win_region X then -> add pred in win_region X
43                     if (pred not in win_region) and all(succ in win_region for succ in
                       ↪  G.get_successors(pred)):
44                         queue.add(pred)
45                         win_region.add(pred)
46                         strategy[pred] = n
47
48     lose_region = set(range(len(G.graph.nodes))) - win_region
49
50     if print_info:
```

```
51          print("Strategy: " + str(strategy))
52          print("Winning region: " + str(win_region))
53          print("Losing region: " + str(lose_region))
54
55      return(strategy, win_region, recursion_count)
```

4.3 Optimizations

The algorithm shown above received some optimizations in order to reduce both the executing time and complexity. One of the optimizations introduced in the algorithm was to substitute lists with sets. At first, Python lists were used to store the queue, the winning region and the losing region. Python sets are not necessarily better than lists but in this case they were better given the high number of membership tests in the algorithm. In particular, lists take O(1) to append an element, O(n) to delete (actually O(1) because the pop operation was used) and O(n) to find an element in the list. The sets are implemented with hashing tables so they take O(1) on average to insert/delete and element while checking if an item is in the set takes O(1) on average that can become O(n) in the worst case. In general, since few insertions and deletions were made and the most common operation was the membership check then using the set brought a huge decrease in the executing time.

4.4 Performance Evaluation

In order to test how the algorithm performs with different configurations, a performance evaluation algorithm was created. This algorithm runs the reachability solver presented above with different configurations of n_nodes, edge_probability and n_winning using a grid search mechanism and measures the time and recursion number needed to solve the game. The results are then saved in a JSON file. The algorithm is the following:

```
1  from datetime import datetime
2  from tqdm import tqdm
3  import reachability
4  import time
5  import json
6  import os
7
8  def evaluate_reachability(edge_probabilities, n_winnings, n_nodes):
9      all_combinations = [(ep, nw, nn) for ep in edge_probabilities for nw in n_winnings for nn
       ↪  in n_nodes]
10     results = []
11
12     for ep, nw, nn in tqdm(all_combinations):
13         print("----Testing the following configuration:----")
```

```
        print(f"Number of nodes: {nn}")
        print(f"Edge probability among nodes: {ep*100}%")
        print(f"Number of winning nodes as a percentage of the total nodes: {nw*100}%")
        start_time = time.time()

        _, _, recursion_count = reachability.reachability_game(n_nodes=nn,
        ↪  edge_probability=ep, n_winning=nw, print_info=False)

        time.sleep(1)
        elapsed_time = (time.time() - start_time) # In seconds

        combination_data = {
            "n_nodes": nn,
            "edge_probability": ep,
            "n_winning": nw,
            "time": elapsed_time,
            "recursion_count": recursion_count
        }

        results.append(combination_data)
        print("---End of the configuration test---")

    if os.path.exists("./Evaluations/") == False:
        os.mkdir("./Evaluations/")

    current_datetime = datetime.now()
    current_datetime_str = current_datetime.strftime("%Y-%m-%d_%H-%M-%S")
    with open(f"./Evaluations/evaluation_reachability_{current_datetime_str}.json", 'w') as
    ↪  json_file:
        json.dump(results, json_file, indent=2)
```

In the results presented above I fixed the percentage of winning nodes at 15%. The other parameters are changes at each configuration to see how performance changes accordingly. The configurations tested are:

```
edge_probabilities = [0.02, 0.05, 0.1, 0.2, 0.35, 0.5]
n_winnings = [0.15]
n_nodes = [10, 50, 100, 1000, 3000, 10000]
```

In order to increase the robustness of the evaluation, 20 tests were made for each configuration and then the average of the results was computed. Finally, the results are presented here:

edge_probability=0.02		
n_nodes	**Avg Time (in sec)**	**Avg Recursion Counts**
10	1.0009354115	3.6
50	1.0018834472	34.55
100	1.0046828628	67.65
1000	1.1064425349	507.1
3000	1.9314035535	1509.25
5000	4.3123691201	2512.85

Table 1: Performance evaluation results with edge_probability=0.02

edge_probability=0.05		
n_nodes	**Avg Time (in sec)**	**Avg Recursion Counts**
10	1.0026243687	3.25
50	1.0024163604	34.3
100	1.0058501124	60.55
1000	1.1541385651	507.85
3000	2.7848705888	1506.55
5000	7.9370730162	2504.65

Table 2: Performance evaluation results with edge_probability=0.05

edge_probability=0.1		
n_nodes	**Avg Time (in sec)**	**Avg Recursion Counts**
10	1.0098279119	5.45
50	1.0028044343	32.45
100	1.0069970489	55.65
1000	1.2298417807	502.05
3000	4.2788440943	1502.95
5000	14.3395955086	2501.55

Table 3: Performance evaluation results with edge_probability=0.1

edge_probability=0.2		
n_nodes	**Avg Time (in sec)**	**Avg Recursion Counts**
10	1.0178026438	4.4
50	1.0033811092	28.9
100	1.0091432929	52.5
1000	1.3827366233	501.0
3000	7.4811639428	1500.9
5000	28.2848608851	2501.1

Table 4: Performance evaluation results with edge_probability=0.2

edge_probability=0.35		
n_nodes	**Avg Time (in sec)**	**Avg Recursion Counts**
10	1.0317748189	5.65
50	1.0037991881	26.15
100	1.0134184599	50.65
1000	1.6399856687	501.3
3000	12.9254255772	1500.35
5000	51.6776298761	2500.6

Table 5: Performance evaluation results with edge_probability=0.35

edge_probability=0.5		
n_nodes	**Avg Time (in sec)**	**Avg Recursion Counts**
10	1.0447369576	4.55
50	1.0034193158	26.05
100	1.0189020634	50.3
1000	1.9018971562	500.4
3000	18.9525875688	1501.35
5000	78.5371464133	2500.05

Table 6: Performance evaluation results with edge_probability=0.5

From the table above it is possible to notice that:

- The time increases as the number of nodes increases. The seconds needed to solve a game with less than 1k vertices remains under two seconds regardless of the edge probability and number of winning nodes. If even more nodes are introduced in the graph, then the time needed increases a lot. In particular, the more dense the graph is in terms of edges, the higher the time needed to solve the game.
- The recursions needed to solve the game are linked with the number of nodes in the graph and generally they are about half of the nodes in the graph. This behavior may also be linked to the fact that we assign the even nodes to Player 0 and odd nodes to Player 1 meaning that the nodes are splitted among the two players.

5 Conclusions

This project aimed to create an algorithm to solve reachability games along with a graph creation algorithm to test it. The solver was successfully implemented in Python and its performance evaluation showed that it is possible to solve a reachability game within a few seconds if the number of nodes is low, and a few minutes if the number of nodes drastically increases.

References

[1] Games on Graphs, Nathanaël Fijalkow

[2] Reachability Games and Parity Games, Volker Diekert and Manfred Kufleitne

[3] Further material presented throughout the AFC seminars