UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RICCO VASCONCELLOS CONSTANTE SOARES

Coevolutionary Genetic Algorithm and Graph Neural Networks for a Risk-like Board Game: Policy Predictions for Population Initialization

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Anderson Rocha Tavares

Porto Alegre
January 2025

“Man is, at one and the same time, a solitary being and a social being.”

— Albert Einstein

ABSTRACT

Over the last decades, various Artificial Intelligence techniques have been employed in Risk implementations to address the unique challenges posed by this complex strategy board game, such as its high branching factor and the variable topology of different maps. Modern state-of-the-art research has gravitated towards the use of Graph Neural Networks and algorithms that learn the game *tabula rasa*, aligning with the latest trends in AI for games like chess, shogi, and Go. Results indicate that this methodology has not yet surpassed the performance of incorporating hand-crafted logic into agents, as in other domains. Recently, Genetic Algorithms have been introduced to the cutting-edge performance toolbox as an alternative to tree-based search methods, showing promising improvements. In this approach, player moves are treated as individuals in a population, which is evolved to identify optimal solutions. As originally proposed (BAUER, 2024), populations are initialized by sampling random moves. This research explores and evaluates the effectiveness of incorporating policy prediction to initialize populations, as opposed to random initialization. In our experiments, the GA with policy initialization significantly outperformed the standard GA, achieving a 9.05% higher score and winning 60.6% of head-to-head matchups.

Keywords: Artificial Intelligence. Reinforcement Learning. Evolutionary Algorithm. Risk. Graph Neural Networks.

Algoritmo Genético Coevolucionário e Redes Neurais de Grafos para um Jogo de Tabuleiro similar a Risk: Predição de Política na Inicialização das Populações

RESUMO

Nas últimas décadas, diferentes técnicas de Inteligência Artificial foram aplicadas em implementações de Risk para lidar com os desafios únicos proporcionados por esse complexo e estratégico jogo de tabuleiro, tais como o alto fator de ramificação do jogo, além das diferentes topologias de cada mapa. A pesquisa moderna que atinge o estado da arte convergiu para a utilização de Redes Neurais de Grafos e algoritmos que aprendem a jogar sem conhecimento prévio (*tabula rasa*), em conformidade com as últimas tendências em IA aplicada a jogos como xadrez, shogi e Go. Resultados indicam que essa linha metodológica ainda não foi capaz de superar agentes que incorporam lógicas específicas do jogo (feitas à mão), conforme acontece em outros domínios. Recentemente, Algoritmos Genéticos foram adicionados à caixa de ferramentas das técnicas de melhor performance, constituindo uma alternativa aos métodos baseados em busca em árvore, apresentando resultados contundentes. Essa classe de algoritmos trata as ações de um turno de um jogador como indivíduos de uma população, que é evoluída para a identificação das melhores soluções. Tipicamente, os indivíduos de uma população serão inicializados randomicamente, isto é, por amostragem de jogadas aleatórias. O presente trabalho visa experimentar o uso de predição de política na inicialização das populações, ao invés de recorrer à inicialização randômica. Nos nossos experimentos, o algoritmo genético com inicialização por predição de política desempenhou significativamente melhor que o algoritmo genético conforme originalmente proposto (BAUER, 2024), obtendo uma pontuação 9.05% maior, e vencendo 60.6% dos confrontos diretos.

Palavras-chave: Inteligência Artificial, Aprendizado por Reforço, Algoritmo Evolucionário, Risk, Redes Neurais de Grafos.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
ML	Machine Learning
CNN	Convolutionl Neural Network
GNN	Graph Neural Network
GCN	Graph Convolutional Network
MCTS	Monte Carlo Tree Search
SOTA	State Of The Art

CONTENTS

1 INTRODUCTION

The game of Risk is a complex multi-player strategy board game with many variations and implementations where each player controls one army trying to conquer the world. Each turn of the game is composed by deploying or units into territories, transferring from one to another, and performing attacks, resulting in numerous possible next states when played in any reasonably large map. Aspects like these impose serious challenges to the task of implementing an AI agent that plays the game on a competitive level, and even well established algorithms for general videogame playing can perform poorly when transposed to Risk-like games.

Over the years, researchers have proposed various approaches to tackle this problem. Early methods heavily relied on translating human heuristics into algorithms, allowing AI to address specific game scenarios. However, these approaches were limited in adaptability and scope. With advancements in machine learning, especially reinforcement learning and deep neural networks, new paradigms have emerged that aim to learn game strategies from scratch (*tabula rasa*). A major example of this is the AlphaZero (SILVER et al., 2017), whch combines Neural Networks (NNs) with Monte Carlo Tree Search (MCTS). Despite their successes in other domains like chess, Go, and shogi, these methodologies have often underperformed when applied to Risk-like games, particularly on larger maps or against rule-based agents.

Genetic Algorithms (GAs) are a class of algorithms that searches for optimal solutions of a problem in a iterative process mimicking biological evolution. They evolve one or more populations through multiple generations, where each individual represents a valid solution to the problem of interest. The solutions are evaluated by a fitness score, indicating the quality of our solution. In terms of the natural selection analogy, its quality of an individual equivalent to its ability to survive. A recent work (BAUER, 2024) proposed a GA agent to Warzone (a Risk-like game), achieving convincing results in a tournament against the most notable techniques. It defines one population for each player, where each individual corresponds to a player's move, and evolve them simultaneously in a competitive co-evolution. The fitness of an individual will be determined by a GNN. The results suggest that this approach addresses the high branching factor of the game more efficiently, in comparison to previously documented agents.

Despite of its impressive results, the initial population of the GG-net is composed by random solutions, a characteristic that may guide the co-evolution to a sub-optimal

solution. In this work, we propose the use of a policy and value GNN in the GA agent, as in AlphaZero, to extract initial populations with policy predictions. In order to verify the effect of starting the evolution with high-quality individuals, we organized tournaments with time control, including the following agents;

- **Vanilla MCTS**: Baseline agent without GNN guidance.
- **GNN-Guided MCTS**: An AlphaZero-like agent.
- **GA Agent**: Standard genetic algorithm, similar to GG-Net.
- **GA with Policy Initialization**: GA agent with populations initialized using the GNN's policy predictions.

Our experiments indicated that the policy initialization may be a valuable addition to GG-Net. Out of the 1050 games, the GA with policy scored 9.05% more points than the standard GA. It presented a particularly strong performance when paired against the standard version, achieving 60.6% in the head-to-head machups.

Additionally, this work's contributions include a public implementation derived from GG-Net's work, which was our starting point (further comments in 5.1).

The remainder of this work is organized as follows: Chapters 2 and 3 will explain some crucial algorithms and provide an overview of the literature in AI agents for Risk-like games. Chapters 4 and 5 provide explanations on the methods utilized, and describe the tests. Finally, Chapter 6 is dedicated to future work and conclusion.

2 BACKGROUND

2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) (BROWNE et al., 2012) is a general heuristic search algorithm for decision-making in vast and complex state spaces, such as board games, planning problems, and real-time strategy scenarios. Its success lies in its ability to efficiently balance exploration and exploitation during tree search without requiring explicit domain-specific heuristics. This feature has made MCTS a cornerstone of modern AI applied to games and other sequential decision-making tasks.

The algorithm is structured around four key phases, which are iteratively performed to build and refine a search tree, described in the subsections ahead.

2.1.1 Selection

In the selection phase, the algorithm traverses the existing tree starting from the root node, selecting child nodes based on a balance between exploration (visiting less-explored nodes) and exploitation (focusing on nodes with high expected rewards). This trade-off is often managed using the Upper Confidence Bound for Trees (UCT) formula (KOCSIS; SZEPESVáRI, 2006), which balances these aspects:

$$UCT = \frac{Q_i}{N_i} + c\sqrt{\frac{\ln N_p}{N_i}},$$

where:

- Q_i: Cumulative reward of child node i.
- N_i: Number of times child node i has been visited.
- N_p: Number of visits to the parent node.
- c: Exploration constant, which controls the degree of exploration.

Nodes with high UCT values are prioritized, ensuring a balance between exploring promising nodes and exploring new ones. This strategy is particularly effective in large and sparse search spaces.

2.1.2 Expansion

Once the selection phase reaches a leaf node (a node without children), the expansion phase adds new child nodes to the tree. These nodes represent unexplored actions or states that can be taken from the current position. Expansion increases the breadth of the tree, allowing subsequent simulations to consider new paths.

Typically, a single child node is added during expansion, although this can vary depending on the implementation and domain.

2.1.3 Simulation (or Rollout)

In the simulation phase, a "random playout" is conducted from the newly added node until the game ends or a predefined horizon is reached. These playouts involve selecting actions at random or using a simple policy (e.g., heuristic-based rules). The outcome of the simulation provides an estimate of the value of the expanded node.

The randomness of this phase enables MCTS to sample diverse trajectories, making it effective in exploring large state spaces.

2.1.4 Backpropagation

The backpropagation phase propagates the results of the simulation back up the tree, updating statistics for each node along the visited path. These updates refine the estimates of each node's value (Q_i) and visitation count (N_i):

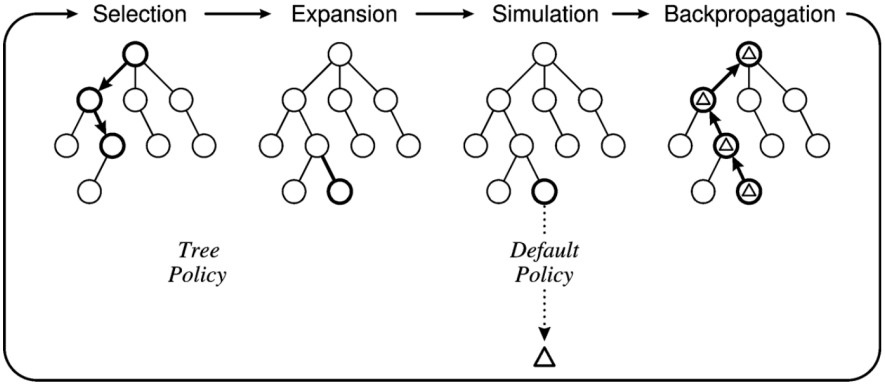
- Increment N_i for each visited node.
- Update Q_i based on the simulation outcome.

This iterative refinement allows MCTS to converge toward optimal decisions as more simulations are performed.

2.1.5 Summary

A typical MCTS iteration can be visualized as shown in Figure 2.1:

Figure 2.1: Visualization of the four key phases of MCTS: selection, expansion, simulation, and backpropagation. The process iteratively refines the search tree.



Source: (BROWN; SANDHOLM; MACHINE, 2017)

2.2 Machine Learning

Machine learning (ML) is a subfield of artificial intelligence (AI) that focuses on developing algorithms and models capable of learning patterns from data and making decisions without explicit programming. A machine learning algorithm can be defined as an algorithm capable of learning from data (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.2.1 Learning Paradigms

Machine learning encompasses three primary paradigms, each suited to specific types of problems:

- **Supervised Learning**: The model learns from labeled data, where each input sample has an associated target label. The goal is to map inputs X to outputs Y by minimizing the error between predictions and the true labels. Examples include image classification and regression tasks.
- **Unsupervised Learning**: The model identifies patterns or structures in unlabeled data. Common tasks include clustering, dimensionality reduction, and anomaly detection. Examples include grouping similar customer profiles or reducing the dimensions of large datasets for visualization.
- **Reinforcement Learning (RL)**: The model interacts with an environment and learns to make decisions by receiving rewards or penalties for its actions. RL is commonly used in robotics, autonomous systems, and game AI, as discussed in

Section 2.7.

2.2.2 The ML Pipeline

The typical machine learning workflow involves several stages:

1. **Data Collection**: Gathering relevant data for the problem at hand.
2. **Data Preprocessing**: Cleaning, transforming, and normalizing the data to make it suitable for model training.
3. **Model Selection**: Choosing an appropriate model architecture or algorithm for the task (e.g., linear regression, neural networks).
4. **Training**: Fitting the model to the training data by optimizing a loss function.
5. **Evaluation**: Measuring the model's performance on unseen test data using metrics such as accuracy, precision, or mean squared error.

2.3 Neural Networks

Neural networks are a class of machine learning models designed to learn and represent complex relationships in data. Inspired by the structure and functioning of biological neural systems, they have become a cornerstone of modern artificial intelligence, powering applications from image recognition to natural language processing and strategic decision-making.

2.3.1 Fundamental Concepts

A neural network consists of interconnected layers of nodes, called neurons. These layers are categorized into three types:

- **Input Layer**: The first layer receives raw data and passes it to subsequent layers.
- **Hidden Layers**: Intermediate layers process the data using learned weights and biases, transforming it into higher-level representations.
- **Output Layer**: The final layer produces predictions or classifications based on the processed data.

Each neuron in a layer connects to neurons in the subsequent layer through weighted edges. The output of a neuron is determined by an activation function, which introduces non-linearity, enabling the network to model complex relationships.

2.3.2 Mathematical Representation

For a neuron, the output h_i is computed as:

$$h_i = \sigma\left(\sum_j w_{ij} x_j + b_i\right),$$

where:

- w_{ij} represents the weight of the connection between neuron j in the previous layer and neuron i in the current layer.
- x_j is the input from the previous neuron.
- b_i is the bias term.
- σ is the activation function, such as ReLU (Rectified Linear Unit), sigmoid, or tanh.

2.3.3 Training Neural Networks

Neural networks learn by adjusting weights and biases to minimize a loss function, which quantifies the error between predicted and actual outputs. The training process involves:

1. **Forward Propagation**: Input data passes through the network, generating predictions.
2. **Loss Computation**: The loss function calculates the error.
3. **Backward Propagation**: Gradients of the loss with respect to weights are computed using the chain rule of calculus.
4. **Weight Update**: Weights and biases are updated using optimization algorithms like Stochastic Gradient Descent (SGD) or Adam (KINGMA; BA, 2017).

2.3.4 Applications in Decision-Making

Neural networks are particularly effective in decision-making tasks. By learning patterns from data, they can:

- **Predict Probabilities**: For example, estimating the likelihood of a specific outcome (e.g., winning a game).
- **Optimize Actions**: Generating policies that maximize rewards in strategic environments, such as games or real-world planning problems.

These capabilities make neural networks foundational in more advanced architectures like Graph Neural Networks (GNNs) and decision-making frameworks like AlphaZero (SILVER et al., 2017).

2.4 Logits, Loss Functions, and Neural Networks in Decision-Making

2.4.1 Logits

Logits are the unnormalized raw outputs of a neural network layer, typically the final layer, before being converted into probabilities. For classification tasks, logits are transformed using activation functions, such as the *softmax* function, to produce probabilities. These probabilities indicate the likelihood of each class or decision being the optimal choice. In the context of games like Risk, logits are particularly useful for representing the relative priorities of moves or actions.

Example of Logits in Policy Predictions: Consider a neural network outputting logits for three possible moves:

- Logits: $[2.0, 1.0, 0.1]$
- Applying *softmax*:
$$P(\text{class}_i) = \frac{e^{\text{logit}_i}}{\sum_j e^{\text{logit}_j}}$$
- Resulting probabilities: $[0.65, 0.24, 0.11]$

Here, the first move has the highest probability of being optimal, based on the logits.

2.4.2 Cross-Entropy Loss

Cross-entropy loss (GOOD, 1952) is widely used for classification tasks to measure the difference between the predicted probability distribution and the true distribution. For a target label y and predicted probabilities $\hat{y}$, the loss is defined as:

$$\mathcal{L}_{\text{Cross-Entropy}} = -\sum_{i} y_i \log(\hat{y}_i)$$

In policy optimization, this loss is used to train the model to assign higher probabilities to actions that align with expert data or optimal policies. Cross-entropy ensures the network's predictions move closer to the desired target distribution over time.

2.4.3 Mean Squared Error (MSE)

Mean Squared Error (MSE) is commonly used for regression tasks, including value predictions in AlphaZero-like frameworks. For predicted values $\hat{v}$ and ground truth v, MSE is defined as:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(\hat{v}_i - v_i)^2$$

In the context of games, MSE helps the GNN refine its evaluation of game states, enabling accurate predictions of the likelihood of winning from a given state.

2.5 Graph Neural Networks

Graph Neural Networks (SCARSELLI et al., 2009) are a class of neural networks that operate on graph-structured data. Unlike traditional neural networks that are designed for grid-like data such as images (2D grids of pixels) or sequences (1D grids of words), GNNs are capable of modeling complex relationships and dependencies through nodes and edges in graphs. This makes them suitable for various applications like social network analysis, recommendation systems, natural language processing, and even biological networks.

2.5.1 Foundations of Graph Neural Networks

A graph G can be represented as $G = (V, E)$, where V is the set of nodes (or vertices) and E is the set of edges connecting these nodes. Each node $v \in V$ can have associated features x_v, and each edge $(u, v) \in E$ can also carry features e_{uv}.

The core idea behind GNNs is the iterative, feature aggregation mechanism where each node updates its representation by aggregating feature information from its immediate neighbors. This process can be summarized as follows:

1. Message Passing: Nodes collect messages from their neighbors.
2. Aggregation: The collected messages are aggregated.
3. Update: Nodes update their embeddings based on the aggregated messages.

2.5.1.1 Message Passing

In each layer of a GNN, every node v gathers messages from its neighbors $\mathcal{N}(v)$:

$$m_v^{(k)} = \text{AGGREGATE}^{(k)}(\{h_u^{(k-1)}, \forall u \in \mathcal{N}(v)\})$$

Here, $h_u^{(k-1)}$ is the representation of node u from the previous layer $k-1$, and $\text{AGGREGATE}^{(k)}$ is a permutation invariant function, such as sum, mean, or max.

2.5.1.2 Node Update

Once the messages are aggregated, the node updates its representation using:

$$h_v^{(k)} = \text{UPDATE}^{(k)}(h_v^{(k-1)}, m_v^{(k)})$$

where $\text{UPDATE}^{(k)}$ can be a neural network that combines the previous node state and the aggregated messages.

2.5.2 Graph Convolutional Networks (GCNs)

Graph Convolutional Networks are a popular variant of GNNs introduced by (KIPF; WELLING, 2017). GCNs extend traditional convolutional operations to graph data, utilizing a propagation rule based on spectral graph theory. The update rule for a node v in a GCN layer can be written as:

$$H^{(k)} = \sigma\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(k-1)}W^{(k)}\right)$$

where $\tilde{A}$ is the adjacency matrix with added self-loops, $\tilde{D}$ is the degree matrix, $W^{(k)}$ are learned weight matrices, $H^{(k-1)}$ are the node embeddings from the previous layer, and σ is an activation function like ReLU.

2.5.3 Graph Attention Networks (GATs)

Graph Attention Networks (VELIčKOVIć et al., 2018) introduce an attention mechanism to GNNs, which allows nodes to assign different importance weights to their neighbors. The attention coefficient $\alpha_{uv}^{(k)}$ for edge (u, v) is computed as:

$$\alpha_{uv}^{(k)} = \frac{\exp\left(\text{LeakyReLU}\left(a^T[W^{(k)}h_u^{(k-1)} \| W^{(k)}h_v^{(k-1)}]\right)\right)}{\sum_{j\in\mathcal{N}(v)}\exp\left(\text{LeakyReLU}\left(a^T[W^{(k)}h_u^{(k-1)} \| W^{(k)}h_j^{(k-1)}]\right)\right)}$$

where a is a learned attention vector, $h_u^{(k-1)}$ and $h_v^{(k-1)}$ are the node features, and $\|$ denotes concatenation. The node update is then:

$$h_v^{(k)} = \sigma\left(\sum_{u\in\mathcal{N}(v)}\alpha_{uv}^{(k)}W^{(k)}h_u^{(k-1)}\right)$$

2.5.4 Transformer Convolution Layers

The Transformer Convolution Layer (`TransformerConv`) (SHI et al., 2021a) extends traditional Graph Neural Networks by incorporating concepts from transformer architectures, enabling enhanced message passing on graph-structured data. Introduced in the Unified Message Passing (UniMP) framework (SHI et al., 2021b), TransformerConv is particularly effective in semi-supervised classification tasks, combining both node feature propagation and label propagation during training and inference.

2.5.4.1 Definition and Parameters

The `TransformerConv` layer operates by leveraging multi-head dot-product attention mechanisms on graph data. Its forward operation integrates features from neighboring nodes, optionally incorporating edge features, as shown in the following equation:

$$h_v^{(k+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{uv}\left(W_h^{(k)} h_u + W_e^{(k)} e_{uv}\right)\right),$$

where:

- $h_v^{(k)}$ and $h_u^{(k)}$: Node feature vectors of nodes v and u at layer k.
- $\mathcal{N}(v)$: The set of neighbors of node v.
- e_{uv}: Edge features for edge (u, v).
- $W_h^{(k)}$ and $W_e^{(k)}$: Trainable weight matrices for node and edge feature transformations.
- α_{uv}: Attention coefficient computed as:

$$\alpha_{uv} = \frac{\exp\left(\text{LeakyReLU}\left(q_v^T k_u + e_{uv}^T W_a\right)\right)}{\sum_{j \in \mathcal{N}(v)} \exp\left(\text{LeakyReLU}\left(q_v^T k_j + e_{vj}^T W_a\right)\right)},$$

  where q_v and k_u are query and key vectors, respectively.

2.6 Policy and Value Predictions

In decision-making frameworks, such as reinforcement learning, policy and value predictions play crucial roles in guiding an agent's actions and assessing game states.

2.6.1 Policy Predictions

A policy prediction represents a probability distribution over all possible actions in a given state. Formally, a policy $\pi(a \mid s)$ is a function that maps a state s to probabilities of actions a:

$$\pi(a \mid s) = \frac{\exp(\text{logit}_a)}{\sum_b \exp(\text{logit}_b)},$$

where logit_a is the neural network's raw output for action a. The softmax function normalizes these logits into probabilities. Policy predictions guide the agent in selecting actions based on their likelihood of success.

2.6.2 Value Predictions

A value prediction estimates the expected cumulative reward from a given state s. The value function $V(s)$ is defined as:

$$V(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s\right],$$

where γ is the discount factor and r_t is the reward at time step t. Value predictions help the agent assess the long-term potential of states, complementing the short-term focus of policy predictions.

2.6.3 Optimization

Policy predictions are optimized using cross-entropy loss, which aligns the predicted probabilities with the optimal actions:

$$\mathcal{L}_{\text{policy}} = -\sum_i y_i \log(\hat{y}_i),$$

where y_i is the true probability distribution, and $\hat{y}_i$ is the predicted distribution.

Value predictions are optimized using mean squared error (MSE), which minimizes the difference between predicted and actual rewards:

$$\mathcal{L}_{\text{value}} = \frac{1}{N}\sum_{i=1}^{N}(\hat{v}_i - v_i)^2.$$

Together, these predictions form the foundation of advanced decision-making frameworks like AlphaZero, integrating strategic planning and state evaluation.

2.7 Reinforcement Learning

Reinforcement Learning (RL) is a computational approach to learning through interaction, where an agent learns to make decisions by optimizing a numerical reward signal over time. The agent interacts with an environment, observes its state, takes actions, and receives feedback in the form of rewards. Unlike supervised learning, RL does not require labeled examples of correct actions but instead relies on the agent exploring and exploiting the environment to maximize cumulative rewards (SUTTON; BARTO, 2018).

2.7.1 Key Concepts in Reinforcement Learning

A reinforcement learning system is composed of the following elements:

- **Agent and Environment**: The agent is the learner and decision-maker, while the environment encompasses everything the agent interacts with.
- **Policy**: The policy $\pi(s)$ defines the agent's behavior, mapping states s to actions a. Policies can be deterministic or stochastic.
- **Reward Signal**: A numerical value r the agent receives from the environment after taking an action, indicating the immediate desirability of the state or action.
- **Value Function**: The value of a state $V(s)$ represents the expected cumulative reward starting from that state and following a policy. Similarly, the action-value function $Q(s, a)$ quantifies the value of taking action a in state s.
- **Model (Optional)**: A model predicts the environment's next state and reward given the current state and action. Methods that use models are referred to as *model-based*, while those that do not are *model-free*.

2.7.2 Exploration vs. Exploitation

A central challenge in RL is balancing exploration (trying new actions to discover their effects) and exploitation (choosing actions known to yield high rewards). Effective RL strategies aim to optimize this trade-off, ensuring that the agent gathers sufficient knowledge of the environment while maximizing rewards.

2.7.3 Markov Decision Processes

Reinforcement learning problems are often formalized as Markov Decision Processes (MDPs), defined by the tuple (S, A, P, R, γ):

- S: Set of states.
- A: Set of actions.
- $P(s'|s, a)$: Transition probabilities, denoting the probability of moving to state s' after taking action a in state s.
- $R(s, a)$: Reward function, specifying the immediate reward received for taking action a in state s.
- γ: Discount factor, $0 \leq \gamma \leq 1$, controlling the weight of future rewards.

The agent's objective is to find an optimal policy π^* that maximizes the expected cumulative discounted reward:

$$G_t = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k})\right].$$

2.7.4 RL Algorithms

Reinforcement learning encompasses several algorithms, including:

- **Q-Learning**: A model-free method that learns the action-value function $Q(s, a)$ through temporal difference updates.
- **Policy Gradients**: Optimize the policy directly by computing gradients of the expected reward with respect to policy parameters.
- **Actor-Critic Methods**: Combine policy-based and value-based methods, where an actor updates the policy and a critic evaluates the policy using a value function.

2.8 AlphaZero: Combining RL and MCTS

AlphaZero (SILVER et al., 2017) is a cutting-edge reinforcement learning algorithm that combines deep learning and Monte Carlo Tree Search (MCTS) to achieve superhuman performance in games like chess, Go, and shogi. Unlike traditional RL ap-

proaches, AlphaZero integrates neural network predictions with search-based planning, enabling efficient decision-making in large state spaces.

2.8.1 Key Features of AlphaZero

- **Unified Policy and Value Network**: A single neural network predicts both the policy (probability distribution over moves) and the value (expected outcome of the game).
- **Self-Play Training**: AlphaZero learns through self-play, iteratively improving its policy and value network by playing against itself and learning from its own moves.
- **MCTS Guidance**: The neural network guides MCTS by providing priors for move probabilities and state evaluations, reducing the computational cost of exhaustive simulations.

2.8.2 Training Workflow

The AlphaZero training process involves:

1. **Self-Play**: The agent generates games by playing against itself, using MCTS guided by the current neural network.
2. **Data Collection**: States, actions, and outcomes from self-play games are stored in a training dataset.
3. **Neural Network Training**: The policy and value network is updated to minimize the loss:

$$\mathcal{L} = \alpha \cdot \text{Cross-Entropy Loss (Policy)} + \beta \cdot \text{Mean Squared Error (Value)},$$

   where α and β control the weight of each loss component.
4. **Policy Improvement**: The updated network is used in subsequent MCTS runs, iteratively improving decision-making.

2.9 Genetic Algorithms and RHEA

Genetic Algorithms (GAs) are optimization heuristics inspired by the process of natural selection (GOLDBERG, 1989), (HOLLAND, 1992). They mimic biological evolution to explore complex search spaces, making them suitable for solving combinatorial and continuous optimization problems. GAs operate on populations of candidate solutions (referred to as individuals or chromosomes), applying evolutionary operators such as selection, crossover, and mutation to evolve towards better solutions over generations.

2.9.1 Core Components of Genetic Algorithms

The typical workflow of a GA involves the following steps:

1. **Initialization**: Generate an initial population of candidate solutions, often randomly. Each individual encodes a potential solution to the problem.
2. **Fitness Evaluation**: Assign a fitness score to each individual, reflecting its quality or how well it solves the problem. The fitness function is a crucial component that guides the evolution process.
3. **Selection**: Select individuals for reproduction based on their fitness scores. Common selection methods include tournament selection, roulette wheel selection, and rank-based selection.
4. **Crossover (Recombination)**: Combine the genetic material of two selected individuals (parents) to produce offspring. Crossover methods vary, including one-point, two-point, or uniform crossover.
5. **Mutation**: Introduce small random changes in the offspring's genetic material to maintain diversity and explore new areas of the search space.
6. **Replacement**: Form the next generation by replacing less-fit individuals in the population with the newly generated offspring.

The algorithm iterates through these steps until a stopping criterion is met, such as reaching a maximum number of generations or achieving a satisfactory fitness score.

2.9.2 Fitness Evaluation in Genetic Algorithms

Fitness evaluation is a critical step in GAs as it determines how well each individual performs in solving the problem. The fitness function varies depending on the application and is often domain-specific. For example:

- In optimization problems, fitness may be defined as the objective function to be maximized or minimized.
- In game AI, fitness often involves simulating gameplay to measure an individual's performance, such as the score achieved, win/loss outcome, or strategic effectiveness.
- In pathfinding or planning tasks, fitness could involve minimizing the distance traveled, energy consumed, or time taken.

In multi-agent systems or games, fitness evaluation often involves interactions between agents. For instance, an individual's fitness may depend on its performance against other individuals or predefined opponents. This dynamic evaluation process introduces co-evolutionary dynamics, where the success of one individual affects the performance of others.

2.9.3 Rolling Horizon Evolutionary Algorithms (RHEA)

Rolling Horizon Evolutionary Algorithms (RHEA) (GAINA et al., 2022) extend the principles of GAs to sequential decision-making tasks, such as games. Instead of evolving a single solution, RHEA evolves sequences of actions over a finite horizon (e.g., game ticks or turns). Each individual in the population represents a potential sequence of actions, and the fitness evaluation simulates the outcome of executing the sequence from the current state.

Advantages of RHEA:

- It can handle large and continuous action spaces by focusing on evolving feasible action sequences.
- RHEA allows flexible adaptation to changing environments or states due to its rolling horizon nature.
- It can serve as a strong alternative to MCTS in decision-making tasks, particularly

in single-agent or deterministic scenarios.

Challenges in Multi-Agent Games: Applying RHEA to multi-agent games introduces unique challenges:

- **Opponent Modeling**: Vanilla RHEA requires handcrafted heuristics to simulate opponents' actions, which may not generalize well across different strategies.
- **Co-Evolutionary Dynamics**: To address the lack of explicit opponent modeling, co-evolutionary approaches evolve separate populations for each agent. For example, Liu, Pérez-Liébana and Lucas (2016) demonstrate a method where two populations are evolved simultaneously—one for each player's sequence of actions. This allows RHEA to adapt to dynamic strategies in competitive settings.

2.10 Risk and Warzone

Risk and War are classic strategy board games where players compete to conquer territories on a map through strategic deployment and combat. Originating in 1957 as "La Conquête du Monde" (The Conquest of the World) by Albert Lamorisse, Risk has undergone various adaptations, including the Brazilian War. Over time, these games have inspired numerous digital variations, one of them being Warzone.

2.10.1 Risk: The Board Game

In the classic Risk game, players take turns drafting territories, deploying armies, and engaging in battles. The goal is to dominate the map by eliminating opponents or achieving specific objectives, depending on the variant. Battles are resolved through dice rolls, introducing an element of randomness that can influence outcomes, making Risk both strategic and luck-dependent.

The game phases include:

1. **Drafting Territories:** Players take turns selecting territories or receive them randomly in certain versions.
2. **Deployment:** Players deploy armies based on their controlled territories and bonuses for entire continents.
3. **Combat:** Players attack adjacent territories to expand their control, resolved through

dice rolls with a balance of offense and defense.

4. **Fortification:** Players can reposition armies to strengthen their positions.

2.10.2 Warzone: A Strategic Evolution

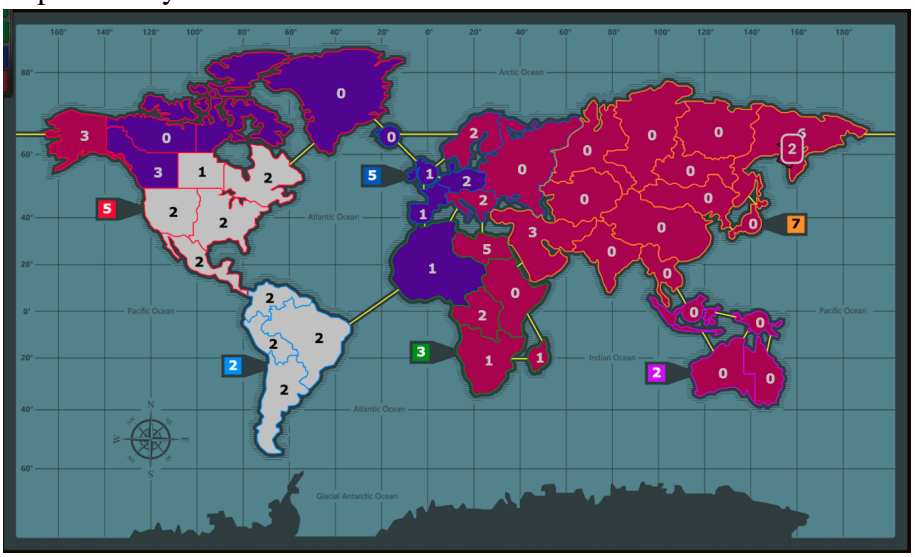
Warzone is a modern digital adaptation of Risk that enhances strategic depth while minimizing reliance on randomness. Figure 2.2 illustrates an ongoing Warzone game. The game retains the core elements of territorial conquest and army deployment but introduces several key differences:

- **Simultaneous Turns:** Players submit their moves simultaneously, unlike the sequential turns in Risk. This adds complexity, as players must anticipate their opponent's actions. After players submitting their moves, the game combines all actions and process them in a **randomized order**. This randomized processing of moves introduces an additional layer of unpredictability and strategic depth to the gameplay.
- **Deterministic Combat:** Unlike Risk's dice rolls, Warzone offers an option to calculate combat outcomes deterministically using fixed kill rates (60% for attackers and 70% for defenders). This setting eliminates randomness, ensuring outcomes depend solely on strategy.
- **No Draft Phase:** Territories are pre-assigned at the start of the game, removing the drafting phase common in Risk.
- **Flexible Configurations:** Warzone supports extensive customizations, such as adjusting bonuses, enabling fog of war, or playing with various team compositions and map designs.

2.10.3 Key Features for Two-Player Matches

For this work, we focus on Warzone configured for deterministic, two-player matches without hidden information. The game begins in a randomly generated state, a notion that is better explained and justified in our context in Chapter 5. Players deploy armies, transfer forces, and attack territories in simultaneous turns, emphasizing prediction and counter-strategy.

Figure 2.2: A Warzone game in progress. Each territory contains a number representing the number of armies stationed there and is colored (pink or purple) to indicate which player owns it. Neutral territories are gray and may contain neutral troops, requiring an attack to be conquered. Colored boxes outline bonus regions (in this case, continents) and indicate the additional income provided for controlling the entire region. The bonus income is specified by the number inside each box.



Source: The Author.

The deterministic combat model is particularly relevant, as it ensures reproducibility and provides a controlled environment for evaluating strategies. Players gain additional armies based on controlled territories and bonuses, which must be strategically deployed to maintain a balance between offense and defense.

2.10.4 Strategic Implications

Warzone's simultaneous turns and deterministic combat shift the focus from chance-based tactics to precise strategic planning. Players must not only allocate resources effectively but also predict and adapt to their opponent's moves. These features make Warzone an excellent platform for studying decision-making algorithms, as the game's high branching factor and complex interactions provide a challenging and dynamic environment.

3 RELATED WORK

The first notable works that proposed AI agents for RISK and similar games relied mostly on translating human knowledge from the game to algorithms to enable high-level play while overcoming the high branching factor for the game. More recently, with the advancement of Machine Learning techniques, the research focused on agents that learn to play the game with no previous knowledge. In 2005, MARS (JOHANSSON; OLSSON, 2005) took a multi-agent approach where an agent is put in each territory, letting them negotiate to decide what actions to prioritize in a bid system. In 2010, Gibson et al. (GIBSON; DESAI; ZHAO, 2010) applied Monte Carlo Tree Search combined with an evaluation function trained under supervised learning to shorten the length of UCT simulations. It only focused on the country drafting phase of the game, but it neatly carries the intuition of combining extensive simulations with the usage of a board critic, pivotal ideas in the more modern approaches.

Later, more cutting edge algorithms were employed to this task; (HEREDIA; CAZENAVE, 2022) represents the era where agents are trained to learn tabula rasa through self-play and Graph Neural Networks replace conventional Convolutional Neural Networks. This work dealt with the computational challenges of using MCTS applying Expert Iteration algorithm to approximate the MCTS policy without performing the actual search. The neural networks also are utilized as a prior bias to restrict the search towards more promising nodes (moves). To sum up, one iteration of the Expert Iteration training process would consist of the usage of the neural network guiding the search of the MCTS and/or evaluating game states at the leaves. Then, the new policy is learned by the neural network. Still, the experiments were made in defined small synthetic maps, and they also just found evidence of learning for the country drafting phase. In 2020, Carr (CARR, 2020) Created an agent using temporal difference reinforcement learning to train a Deep Neural Network including a Graph Convolutional Network to evaluate player positions. It also tackles the non-determinism in Risk with in the attack phase of the game introducing a new method, pre-calculating the many possible results of an attack and performing re-syncs after each attack. This allows opting for an agent with more conservative/pessimistic or more optimistic/aggressive strategies in regard to dealing with the randomness. The first work to experiment with Genetic Algorithms in Risk-like games was GG-Net (BAUER, 2024), evolving two populations simultaneously (each players moves), with the fitness of each individual being determined by a GNN that eval-

uates the winning probability of the player. It also achieves generalization across different maps. The chosen platform for this work was Warzone, dealing with deterministic attacks, where units lost in defending/attacking players are defined by a fixed ratio. Bauer (2024) implemented the most dominant techniques as agents to test the proposed GG-Net in a tournament in different maps, estimating their performance to find that this GA algorithm surpassed its contestants. In 2015, a Warzone AI tournament had an agent named Cowzow as the winner. In general, its competitors implemented approaches that combined tree seach with handcrafted evaluation functions, or rule-based heuristics. The winner used the Ford-Fulkerson algorithm (FORD; FULKERSON, 1956) to efficiently distribute armies. Cowzow was also included on GG-Net's tournament, where it reached second position. Despite losing to GG-Net, which is also a *tabula rasa* approach, Cowzow managed to outperform a MCTS-GNN approach, that should be similar to AlphaZero. This result outlines that the intrinsic characteristics of the game can make it challenging even to well established techniques in other domains.

4 METHODS

In this chapter, we describe the core methods related to the GNN, and Genetic Algorithm used in this work. Section 4.1 describes how GNNs are applied in Risk, which will be a critical component of both MCTS and GA AIs, while Section 4.2 elaborates on the Genetic Algorithm, covering its fundamental concepts, and justifying design choices adopted along this work.

4.1 Graph Neural Network

The GNN plays a central role in both the MCTS and GA agents. In MCTS, policy predictions guide tree expansions by prioritizing promising moves, while its value predictions replace expensive rollout simulations. In evolutionary agents, policy predictions initialize populations with high-quality individuals, while value predictions are critical to the fitness assignment during the competitive coevolutionary process. This section covers how Risk elements, such as the board and actions, are represented as graph-related entities, and how the GNN interacts with these entities to learn the game.

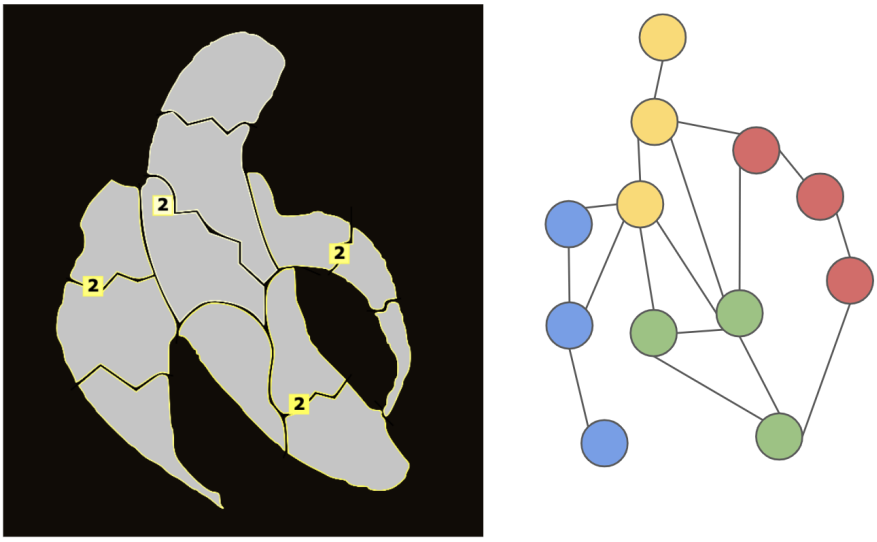
4.1.1 Map and Vertex Features

The game maps are modeled as graphs, where each territory corresponds to a vertex, and edges represent adjacency between territories. Figure 4.1 illustrates a Warzone map and its graph representation. The left side highlights four bonus regions with yellow boxes denoting the income bonus awarded for controlling a region, while the right side displays territories grouped into the different regions. On more complex maps however, bonus regions often share territories, causing them to overlap.

Territories are considered adjacent if they can be directly accessed from one another, reflecting physical connectivity. However, visual proximity on the map does not always correspond to adjacency. For example, two territories sharing a border may be impassable due to natural barriers such as rivers or mountains.

Total armies for each player and their income are incorporated to the value and prediction pipeline as global features. Vertex features encode the state of each territory, including:

Figure 4.1: From left to right, a Warzone map, and its graph representation. On the left, the yellow boxes indicate the income bonus offered by its four bonus regions. On the right, each color group denotes the territories composing each region



Source: The Author.

- **Ownership**: One-hot encoded to indicate control by the player, the opponent, or neutrality.
- **Army Count**: Scalars representing the number of stationed armies, separated by ownership type.
- **Bonus Membership**: A binary vector denoting whether the territory belongs to one or more bonuses.

Figures 4.2 and 4.3 illustrate how ownership and army count are encoded as vertex features.

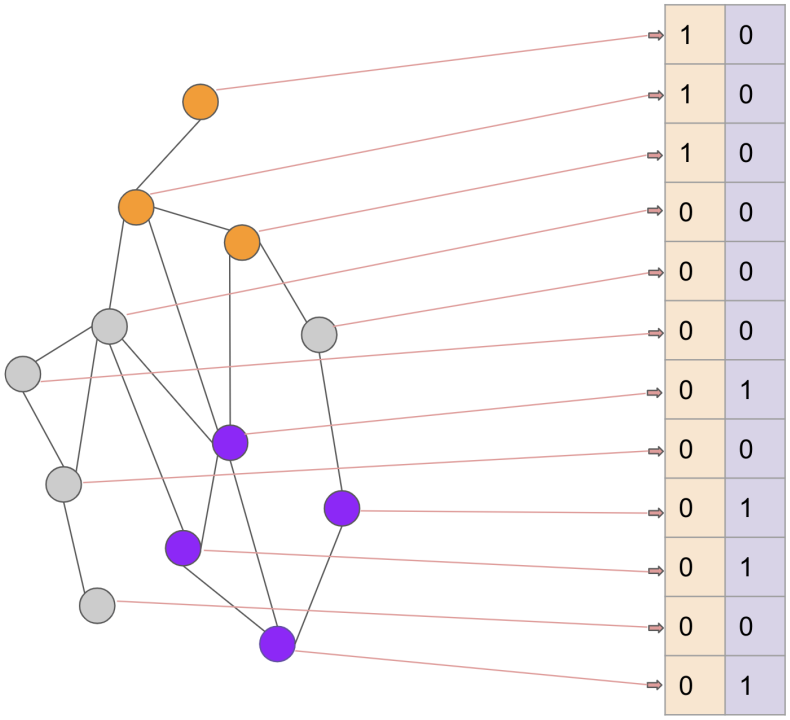
4.1.2 Policy and Value Predictions

The GNN outputs two predictions for a given game state:

- **Value Prediction** v: A single scalar estimating the player's winning likelihood from the current state.
- **Policy Prediction** π: A probability distribution over a set of *candidate moves*, where each move can consist of attack, transfer, or deployment orders.

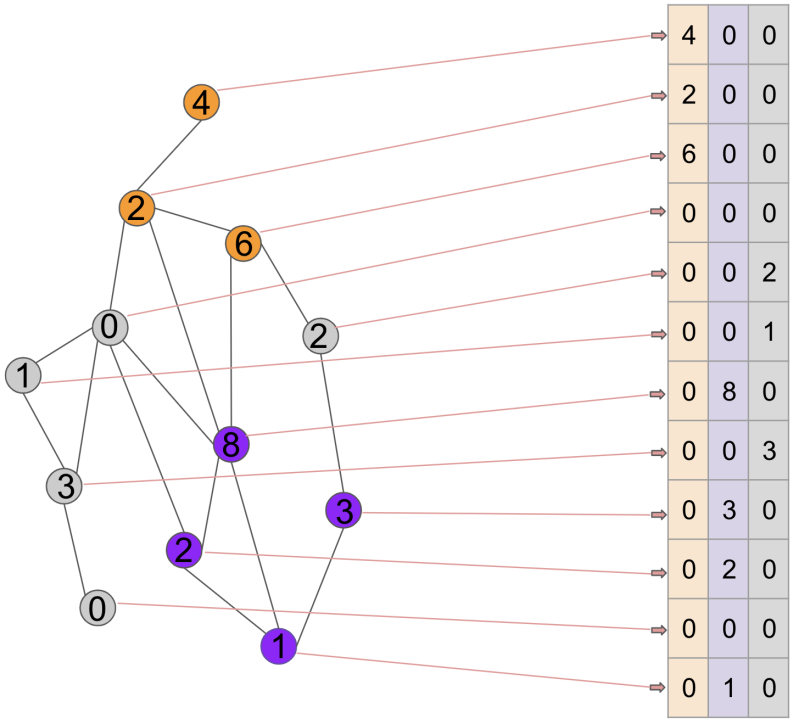
Rather than automatically enumerating all possible actions, the GNN receives a *list of candidate moves* to evaluate. It treats each candidate move (or order) by extracting

Figure 4.2: Ownership of territories as vertex feature. Orange and purple nodes represent territories controlled by a player. Neutral territories are gray.



Source: The Author.

Figure 4.3: Armies encoded as vertex feature.



Source: The Author.

the relevant node embeddings for source and destination territories, as well as the number of armies involved. These features pass through specialized dense layers to generate a logit for each candidate. The logits are then pooled to form a distribution over the input moves. This approach provides flexibility: for instance, MCTS can supply child moves for scoring, while a GA might submit randomly generated or evolved moves for evaluation. Each order is encoded by its source and destination territories, plus the number of armies. This encoding is described in Section 4.1.4.

4.1.3 Architecture Overview

The GNN architecture employs TransformerConv layers (SHI et al., 2021a) and dense layers to process graph-level and bonus-specific information. Figure 4.4 outlines the following steps:

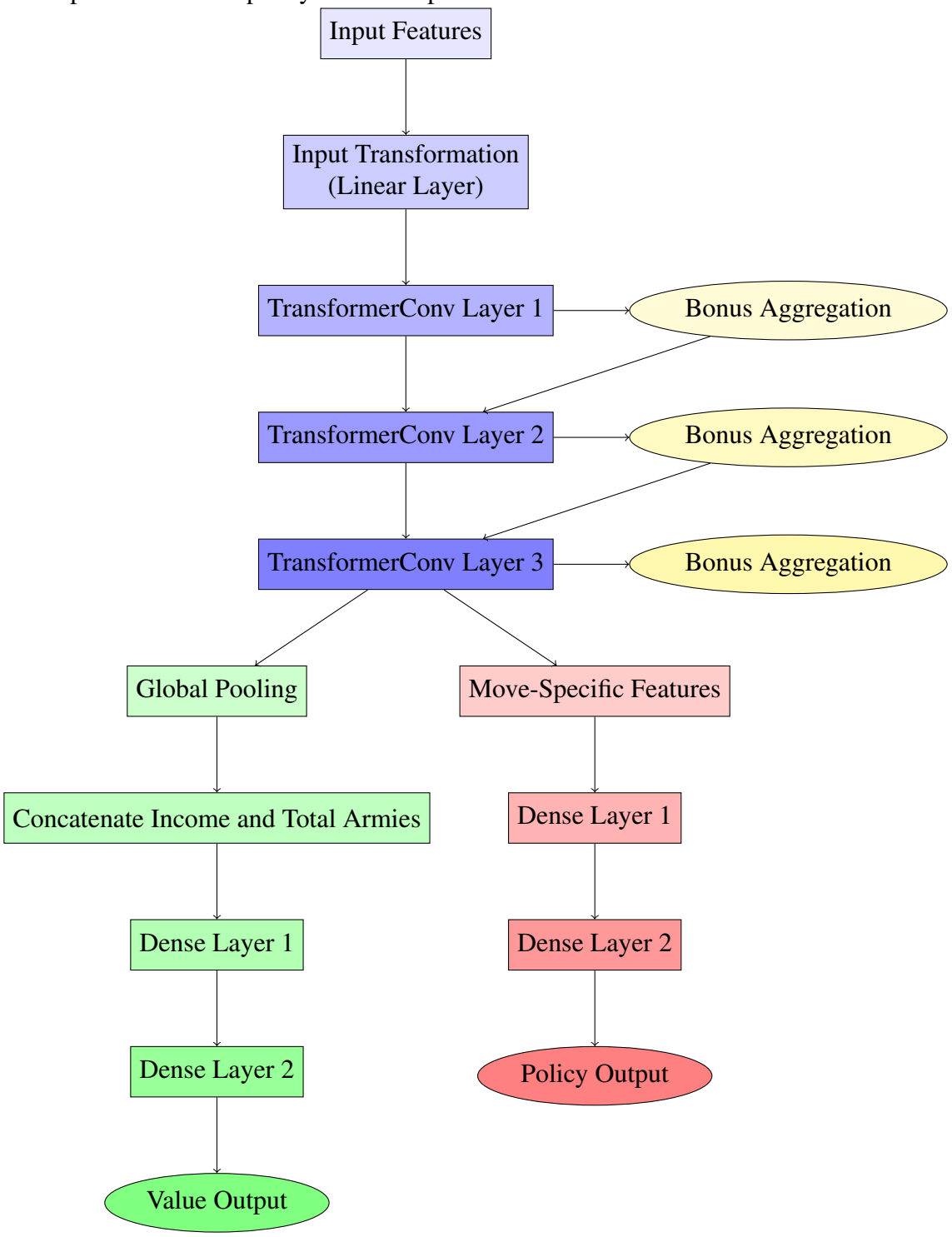
1. **Input Transformation**: A dense layer projects raw input features into the initial embedding space.
2. **Graph Processing Layers**: Three TransformerConv iterations enhance node embeddings via adjacency and bonus-specific connections. At each step, bonus-level features are aggregated from node embeddings and fed back into the graph representation.
3. **Policy Head**: Move-specific features (attack, transfer, deploy) are processed by separate modules. The outputs are pooled and normalized to yield logits for each move passed to the network.
4. **Value Head**: Node embeddings are globally pooled to form a single graph-level vector, which is concatenated with global features (e.g., incomes, total armies) and passed through dense layers to predict the value.

4.1.4 Actions Encoding

This subsection further explains how each candidate action is encoded for the *policy head*. As introduced in Section 2.6, the GNN receives a batch of candidate moves. Each move is represented by:

1. **Source and Destination Territories**: Interpreted as vertex indices in the game's

Figure 4.4: Graph Neural Network architecture for policy and value predictions. The architecture includes TransformerConv layers for graph processing, bonus integration, and separate heads for policy and value predictions.



Source: The Author.

graph. If source and destination are the same territory, it is a deploy action; if they differ and are both owned by the same player, it is a transfer; if ownership differs, it is an attack.

2. **Number of Armies**: An integer specifying how many troops are assigned to the order.

Figure 4.5 illustrates a pair of territories $t1$ and $t2$. An oriented edge from $t1$ to $t2$ plus an integer troop count collectively describes a single order. By combining these node embeddings (output from the TransformerConv layers) with the integer troop count, the policy head produces a logit for that order.

Figure 4.5: Two adjacent territories interpreted as vertices. The oriented edge denotes that one game order mobilizes troops from $t1$ to $t2$. The GNN's policy head encodes this order by concatenating node embeddings for $t1$ and $t2$ plus an integer representing troop count.

Source: The Author.

Hence, the GNN can handle any list of candidate moves if they are encoded in this consistent (source, destination, armies) manner. Internally, ownership information in the node embeddings determines whether an action is an attack, transfer, or deployment.

4.1.5 GNN Training

The training process for the Graph Neural Network (GNN) was inspired by the AlphaZero framework (SILVER et al., 2017), incorporating iterative self-play, experience collection, and parameter updates. In each iteration, a guided MCTS agent, utilizing the current GNN model, played multiple games against itself to generate self-play experiences. These experiences, including game states, actions, and rewards, were stored in a replay buffer. Each iteration consists of:

1. **Self-Play**: A guided MCTS agent played games to populate the replay buffer with new experiences.

2. **Parameter Updates**: The replay buffer was sampled to train the GNN using policy and value loss functions.
3. **Evaluation**: The updated model was evaluated on a set of games to assess improvements in performance.

Network parameters were updated using the Adam optimizer (KINGMA; BA, 2017), with policy loss computed using Cross-Entropy (GOOD, 1952) and value loss using Mean Squared Error (DODGE, 2008). This iterative process ensured that the GNN refined its policy and value predictions over time, closely following the principles of the AlphaZero training paradigm. Training was performed over multiple epochs, with data stored in a replay buffer for batch updates.

At each evaluation step, the trained model plays a number of games against two agents; A MCTS-based agent guided by the model from previous iteration, and a vanilla MCTS agent (not guided by GNN). The performance against the previous model indicates the learning at each iteration, while the vanilla agent is a fixed strength baseline for comparison, indicating the learning across iterations.

The training scope is designed using principles of curriculum learning (SOVIANY et al., 2022), which observe that focusing on learning easier tasks first often increases the accuracy and convergence speed of ML models. The first iterations of training included games in a set of three small maps, while the last iteration included a larger map, which is more complex to learn. This process could be repeated indefinitely.

Details of the specific training iterations, including the maps used and the evolution of the training process, are provided in Chapter 5.

4.2 Genetic Algorithm

This section explains in detail the fundamental notions of the GA, starting with an overview in Subsection 4.2.1. Next, Subsection 4.2.2 explains how individuals are evaluated during co-evolution. Then, Subsection 4.2.3 describes the life cycle of this evolution process. Subsection 4.2.4 concentrates on how actions are encoded into genes, a needed understanding to discuss the handling of mutation and crossover steps, found in 4.2.5. Lastly, we describe how GNN policy predictions are leveraged for population initialization (Section 4.3).

4.2.1 Overview

The idea of GA is to simulate the natural evolution of populations, analogous to biological evolution. The real world analogy includes entities representing genes, individuals, and populations to be evolved. In our context, the genes of the individuals will be possible actions of a player from a given board state (deploy, attack, or transfer). Multiple genes will compose an individual, which by extension is defined as a set of actions for a player's turn, or a move. Following the analogy, a population is defined as multiple individuals, or multiple possible moves for a player on a board state. Each individual will have a fitness value representing the ability of this individual to survive. This fitness in our context should represent how strong a move is.

The evolution process aims to simulate multiple steps of crossovers and mutations, also called generations, maintaining only the fittest individuals in the population. At the end of the evolution, the fittest move of the population is selected to be played. Since the players play each turn simultaneously, the evolutionary algorithm will define a population for each player, and they will be co-evolved. At the end of this co-evolution, the best move of the population that represents the moves of the GA agent is selected to be played.

4.2.2 Fitness Assignment

The fitness of an individual depends on its average performance against individuals of the reciprocate population. Suppose that x is an individual that belongs to the population X, and that y is an individual of population Y, with s being the initial board state of the turn. After simulating the moves x and y, reaching the resulting board state s', a position evaluation v is assigned to s', that expresses how advantageous this resulting board state is for each player. This performance of x against y, evaluated as v will be referred to as the relational fitness of the pair (x, y). By obtaining value predictions for the relational fitness of all pairs (x_n, y_n) for populations X and Y, the fitness of the individual x will be given by its average relational fitness against individuals of Y. This average is the average of values $v_n = RelationalFitness(x, y_n)$, for every y_n in Y. Algorithm 1 contains the pseudocode for the fitness assignment of individuals from the populations X and Y that is done at each populations evaluation step of the evolutionary algorithm in this competitive co-evolution. Value predictions for board states will be estimated by a trained GNN.

Algorithm 1 Pseudocode to evaluate two populations X and Y of length n, assigning a fitness value for their individuals.

Require: Population X with n individuals for player $P1$
Require: Population Y with n individuals for player $P2$
Require: Function $SimulateMoves(x, y)$:returns resulting board state s'
Require: Function $EvaluateBoardState(s')$:returns evaluation v for $P1$'s score
Initialize matrix $RelationalFitnessTable[n][n]$ with zeros
Initialize array $FitnessX[n]$ with zeros
Initialize array $FitnessY[n]$ with zeros
for each individual x in X indexed by i **do**
    for each individual y in Y indexed by j **do**
        $s' \leftarrow SimulateMoves(x, y)$
        $v \leftarrow EvaluateBoardState(s')$
        $RelationalFitnessTable[i][j] \leftarrow v$ {Evaluation for $P1$}
    end for
end for
for each individual x in X indexed by i **do**
    $fitness \leftarrow 0$
    for each individual y in Y indexed by j **do**
        $fitness \leftarrow fitness + RelationalFitnessTable[i][j]$
    end for
    $FitnessX[i] \leftarrow fitness/n$ {Average relational fitness of x over all individuals in Y}
end for
for each individual y in Y indexed by j **do**
    $fitness \leftarrow 0$
    for each individual x in X indexed by i **do**
        $fitness \leftarrow fitness + RelationalFitnessTable[i][j]$
    end for
    $FitnessY[j] \leftarrow -fitness/n$ {Average relational fitness of y over all individuals in X. Signal correction is needed for $P2$}
end for

4.2.3 Life Cycle

Each generation has steps of evaluation, crossover, mutation, and elitism. Figure 4.6 displays the life cycle of the competitive coevolution, showing how these steps are coordinated with fitness assignment steps over the evolution. The first step of each evolution instance is initializing the populations. Typically, the default initialization is done by sampling random individuals.

Crossover and mutation are the basic biological operations that will produce new individuals that share genetic information with previously existing ones. Considering that the population size is constant, and that the less adapted individuals are selected out of the population, it is expected that the moves become progressively stronger across generations. The balance between exploring new moves and preserving genetic information from the best-evaluated moves is typically determined by hyperparameters like mutation rate, crossover rate, the portion of genes to be mutated in mutated individuals, and the size of the elites. Mutation rate dictates the chance of an individual suffering mutations, while crossover rate quantifies how many new individuals will be generated by mating parents from the current population. Elitism is a resource to carry the fittest individuals to the next generation, preventing them to be replaced or selected out of the population at any step. The size of the elites is also controlled by a hyperparameter.

As implemented, no step will change the size of the populations. To assert this, the mutation operation replaces the original individual in the population by the mutated one. Similarly, the offspring generated in crossover also replace their parents. Different strategies of selecting individuals to mate and to be mutated are viable, and in the present work, all mating parents and individuals to be mutated are selected randomly. At each crossover step, two parents will be selected and two new individuals will be generated by crossover until the offspring set reaches the size of the population. Elites are copied at the beginning of each generation to be re-introduced in the population at the end when the elitism is applied, as shown in Figure 4.6. As a result of these design choices, elitism is the only way for one individual to reach the next generation, and all other elements will be replaced by the byproduct of crossover, which might contain part of their genes. Elites still participate in the crossover operation, as they were not removed from the population at the elites definition step. The mutation step will select random elements of this offspring population based on the mutation rate parameter. A selected individual will have a random chance of having each one of its genes mutated.

At the end of a generation, populations will be evaluated, and elitism will be applied. The evaluation consists of a fitness assignment, as already covered in detail in algorithm 1, followed by sorting both populations, ranking their individuals by their fitness. Applying elitism reintroduces the elites in the population, excluding a number of worst solutions equal to the size of the elite.

While the fitness evaluation method in this algorithm ensures that each individual is tested against all individuals from the opposing population, there exist simpler selection mechanisms such as tournament selection. In tournament selection, a small, randomly sampled subset of individuals competes, and the individual with the highest fitness among them is selected for mating. This method balances selective pressure and diversity and is thoroughly described in the literature (BLICKLE; THIELE, 1995).

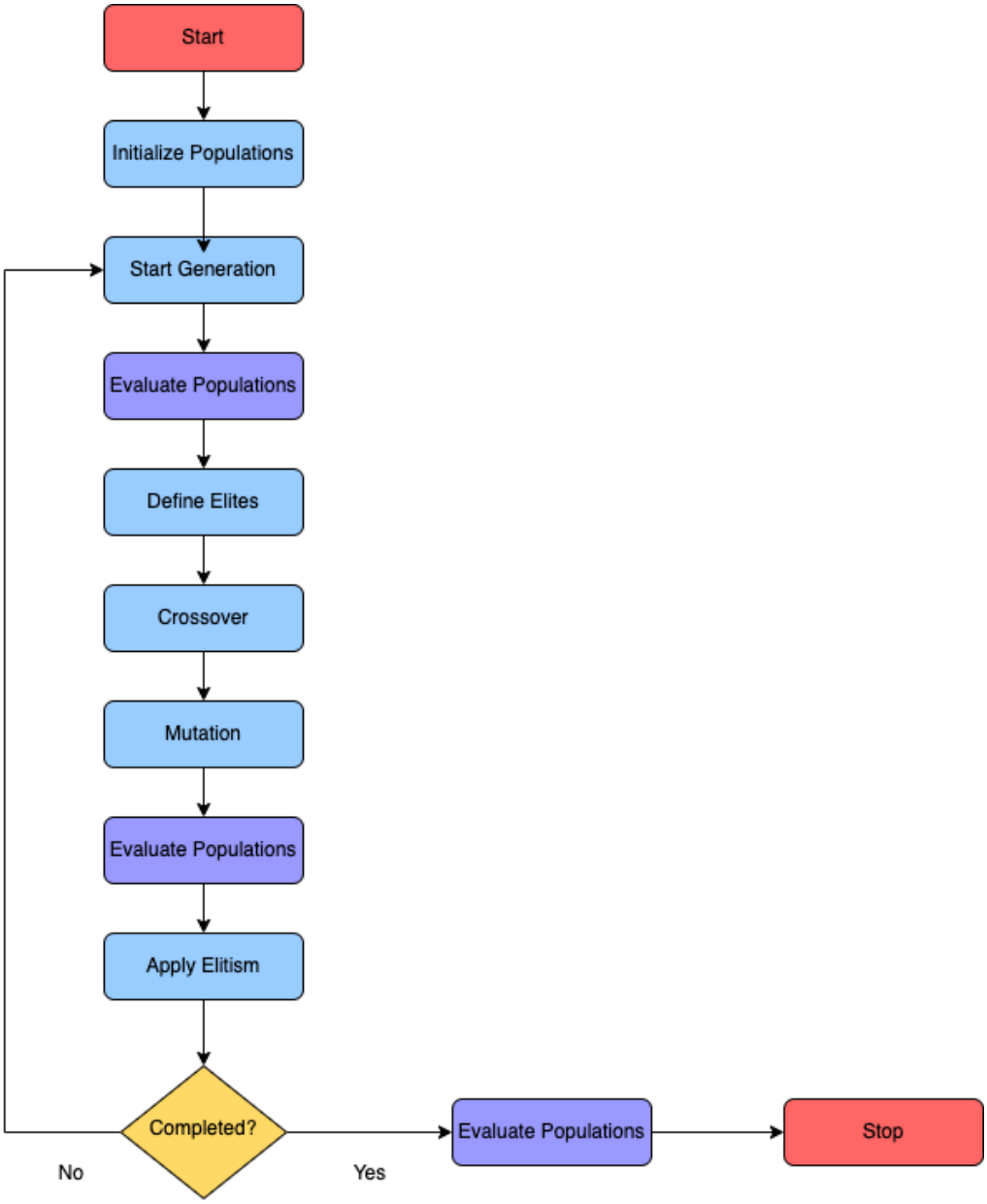
The conditions that will set the evolution as complete are if the desired number of generations are performed, or if the playing agent has reached its time limit. In practice, checks are added between each step within a generation to check for the time elapsed. In case the agent has exceeded the given time budget, the evolution will be interrupted, and the populations will be evaluated and sorted, so that the rank is preserved and the best solution can still be retrieved to be played.

4.2.4 Genes Encoding

The evolutionary algorithm interprets individuals as an array of integers, where each integer is a gene. To encode and decode actions of the game into genes, it is necessary to establish a bijective function. The possible actions of a player for a given board state are mapped one-to-one to a gene, which may be part of an individual of its population. To demonstrate how this occurs, it is important to recap some graph notions. Section 4.1 argues that the orders of the game can be mapped to a number of armies a associated with a pair or vertices $(source, destination)$. Let G be the graph of a Risk map, and E be a set that contains all edges in G, considering that every vertex in G is connected to itself. To encode an order of a player that mobilizes a troops into an unambiguous gene of an individual I of length $|E|$, we consider that each position of the array I corresponds to one different element of E, and attribute a to $I[index]$, where $index$ corresponds to the desired connection $(source, destination)$ of E. Algorithm 2 illustrates this process, expanding it to show the encoding of a move M into an individual I.

Decoding individuals back to game moves is possible by performing the inverse

Figure 4.6: Competitive co-evolution flowchart.



Source: The Author.

procedure. For a current board configuration, each gene in the genetic sequence of an individual can correspond to only one type of action (attack, deployment, or transfer). We have established that each index of the genetic sequence is mapped to one pair of vertices of the graph. Then, for each gene, if the integer is greater than zero, an order will be added to the resulting move. The order type is inferred from the ownership of the vertices of the pair associated with the index in the current board state. Algorithm 3 demonstrates how an individual I can be decoded into a move M, a list of orders. The resulting move is ready to be simulated in a game manager implementation.

Algorithm 2 Pseudocode to encode move M, containing n orders, into individual I. Each order is an integer a over a pair of territories $(source, destination)$.

Require: List of orders M containing n orders
Initialize G as the graph representing a Risk map.
Initialize set E with all pairs of connected vertices in G.
Initialize set V with all vertices in G.
Initialize array I of size $|E| + |V|$ with zeros.
for each v in V **do**
    Add (v, v) to E
end for
for each $(\text{t1}, \text{t2})$ in E **do**
    Map $(\text{t1}, \text{t2})$ to $index$
end for
for each $order_n$ in M **do**
    $I[\text{index from } (source_n, destination_n)] = a_n$
end for

Algorithm 3 Pseudocode to decode individual I into move M, a list of orders. Each order is an integer a over a pair of territories $(source, destination)$.

Require: a graph G representing a Risk map.
Require: Individual I, which is an array of integer
Require: Function $Map(index)$: maps each index of I to a pair $(src, dest)$ of G
Require: Function $Interpret(order)$: interprets the order type of $order$ for current board state
Initialize empty list M of orders.
for each gene g in I indexed by $index$ **do**
    if $g > 0$ **then**
        $(src, dest) \leftarrow Map(index)$
        $order \leftarrow ((src, dest), g)$
        $interpretedOrder \leftarrow Interpret(order)$
        Add $interpretedOrder$ to M
    end if
end for

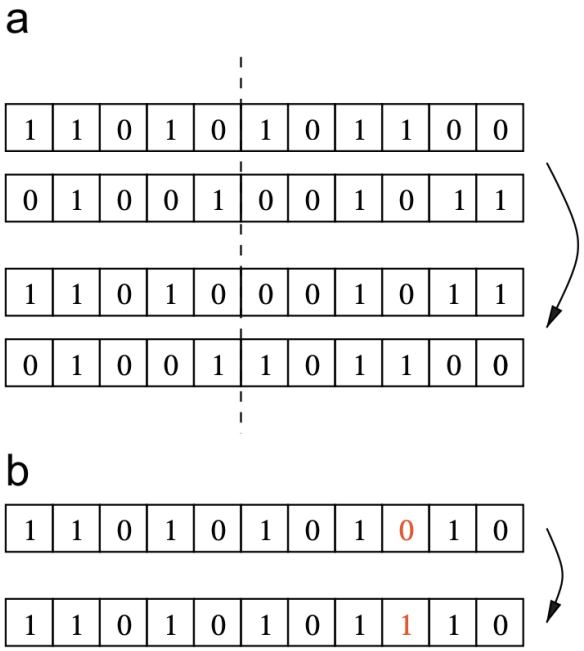
4.2.5 Mutation and Crossover

Knowing how possible moves for a turn are encoded into array of integers, the mutation operation can be defined as changing one or more integers of an individual's genetic sequence. In analogy, a crossover operation can be defined as a function that takes two individuals as inputs and produces one or more new individuals. The genes of these offspring individuals will be a combination of the genetic sequence of their parents. Figure 4.7 displays both operations at a gene level. The crossover in the image corresponds to a one-point crossover, or single-point crossover (WORDEN; STASZEWSKI; HENSMAN, 2011), where each parent suffers an imaginary cut at the same single point, and new individuals are generated by swapping the genetic sequences from the point of the cut. For the mutation, Fig. 4.7 presents a simple mutation in which one gene of an individual is changed from 0 to 1. The single-point crossover corresponds to the crossover operation performed at the evolution of the GA agents tested in this work, while the mutation implemented has a chance of switching each gene of the mutated individual to a random number. This chance is controlled by a hyperparameter.

Frequently, the domain of the problem restricts the possible values for the combination of genes. In these cases, operations such as mutation and crossover can lead to invalid solutions. In our domain, these operations can produce moves in which the number of deployed armies exceeds player's income, or that contain deployments to unowned territories. Considering the evolution cycle described in 4.2.3, checks for these constraints and eventual corrections only need to be applied after mutation, before fitness assignment. For this reason, correction will be considered as part of the mutation process.

After randomly mutating genes of an individual that initially encodes a valid action, genes corresponding to deployment orders over enemy territories are set to zero. In addition, exceeding deployments are removed; if the sum of armies deployed exceeds the player's income, armies are randomly subtracted from deployment orders until the condition is satisfied. Finally, since there is no benefit in not deploying available troops (the income cannot be accumulated), valid deployments are randomly added in case there is income left.

Figure 4.7: Genetic operations (a) one-point crossover, and (b) mutation.



Source: (WORDEN; STASZEWSKI; HENSMAN, 2011)

4.3 Policy Predictions for Population Initialization

The default initialization strategy in Genetic Algorithms is to sample individuals randomly. However, we leverage the policy predictions from our GNN to seed the initial populations with higher-quality moves. Concretely:

1. **Generate a large pool of random moves.** For each player's population, we create an oversize set of random moves (e.g., three times the desired population size).
2. **Score each move using the GNN policy head.** The GNN takes as input the board state plus these candidate moves and outputs policy logits for each candidate.
3. **Select the top-n moves to form the initial population.** By sorting candidates according to their GNN-assigned scores (logits), we choose the best moves as individuals.

This heuristic ensures that each population begins with a set of moves the GNN already considers more promising, rather than starting purely at random Section 5 contains experiments to test if the policy-based initialization provides a beneficial "head start" in the competitive coevolution.

5 EXPERIMENTS

This chapter presents the experiments conducted in this work, outlining the methodologies for training and evaluation, as well as the results obtained. Section 5.1 provides a brief implementation overview, including the codebase, the maps used in the experiments, and the rules enforced in the matches. Section 5.2 details the training process and performance of the GNN. Section 5.3 evaluates different agents through tournaments, focusing on the impact of using GNN policy predictions for Genetic Algorithm (GA) initialization.

5.1 Risk Implementation

5.1.1 Code Base

The experiments utilized a custom implementation based on the Warzone simulation framework from (Bauer, Andrew, 2023), with modifications to meet the needs of this study. The updated codebase is available at (Soares, Ricco, 2024). Key modifications include:

- **Local Play Support**: Hand-crafted maps replaced external dependencies due to Warzone's restricted access to official maps without paid membership.
- **Competitive Coevolution**: A dedicated module was designed to support competitive co-evolution, replacing PyGAD (GAD, 2021), which does not officially support coevolution. The new module offers better compatibility and flexibility.
- **GNN Training and Evaluation**: Scripts for policy and value GNN training were developed and integrated into the codebase.

5.1.2 Maps

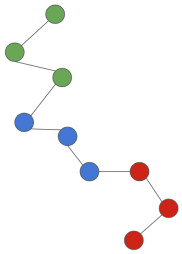
Four maps were used in the experiments (Figure 5.1). The maps vary in size, topology, and complexity:

- **Small Custom Map**: A simple sequential map with 9 territories and 3 bonus regions.
- **Owl Island and Banana Maps**: Both contain 12 territories and 4 disjoint bonuses,

differing only in topology.

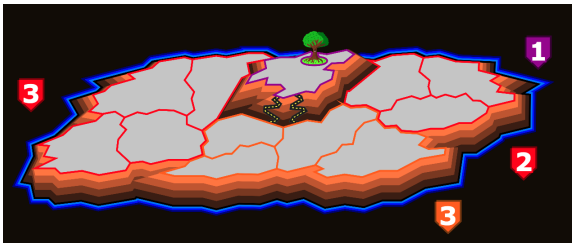
- **Italy Map**: A larger map with 20 territories and 10 bonus regions, including more complex non-disjoint bonuses.

Figure 5.1: Maps used in the experiments.



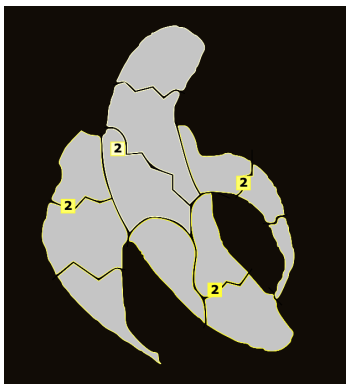
(a) Small Custom Map: 9 territories and 3 bonuses. Each region offers a bonus of 2 units.

Source: The Author.



(b) Owl Island Map: 12 territories, 4 bonuses.

Source: <https://www.warzone.com/SinglePlayer?PreviewMap=56763>



(c) Banana Map: 12 territories, 4 bonuses.

Source: <https://www.warzone.com/SinglePlayer?PreviewMap=29633>

(d) Italy Map: 20 territories, 10 bonuses.

Source: <https://www.warzone.com/SinglePlayer?PreviewMap=3448>

5.1.3 Match Rules

Each game started in a random board state, where territories and armies were distributed randomly. Despite introducing a luck factor into the evaluation, approaches

like this are common in many ML agents for games. In the agent evaluation step, the random positions prevent early-game exploits. This practice ensures agents' performance correlates more to a general understanding of the game as a whole, than to hand-crafted heuristics for the initial phase of the game or map characteristics exploits. Similarly, in GNN training, it increases diversity in data. In TCEC (Top Chess Engine Championship), competitors play from predetermined positions, playing a game as white and as black for each position. In our evaluation matches, luck is indeed a factor that produces noise over the results, since players do not switch perspective. However, this should be mitigated as numerous games are played.

To avoid prolonged matches or stalemates, two tie conditions were enforced:

- The game was considered a tie after 50 turns if both players had winning confidence between 49% and 51%.
- A game exceeding 100 turns was also declared a tie.

In order to save computational resources, players will agree on the winner in clear winning positions. This happens in cases where one player has a winning confidence of at least 99% while the other has 1% or less, and it is considered that the losing player surrenders. Match scoring followed a chess-like convention, where wins awarded 1 point, and ties awarded 0.5 points to each player.

5.2 GNN Training Results

The network training followed an AlphaZero-inspired framework described in Subsection 4.1.5. This process involved three iterations of self-play for data collection, parameter updates, and evaluation. Results for each iteration are presented in this section, followed by a brief discussion.

After updating parameters, learning was evaluated through matches against the previous model and a fixed strength baseline. In each match, 30 games are played on each map. The updated agent subsequently played against itself to collect more data, which was accumulated in a replay buffer. The final dataset contained about 89,254 turns. The remainder of this section details each iteration and its results.

5.2.1 Setup

The training setup included a baseline vanilla MCTS agent (without GNN guidance) and a GNN-guided MCTS agent. Their hyperparameters are listed in Table 5.1. Under these specifications, our baseline agent took about 30s to make each move, while the guided agent required about 15.5 seconds.

Rather than using randomly initialized weights for the first training iteration, the initial dataset was generated from played by the baseline agent. Subsequent iterations expand this dataset with experiences from the guided agent using the updated network, optimizing its outputs using MCTS with 150 iterations. These experiences served as policy and value estimators for parameter updates in the next iteration, following a methodology similar to (SILVER et al., 2017). The GNN training hyperparameters are detailed in Table 5.2.

Table 5.1: MCTS Parameters for Training

	Baseline	GNN Guided
Iterations	300	150
Max Rollout Depth	20	20
Policy Trust	N/A	0.75
Time Limit	∞	∞

Table 5.2: Neural Network Optimization Hyperparameters

Optimizer	Adam
Policy Loss	Cross Entropy
Value Loss	Mean Squared Error
Learning Rate	0.001
Batch Size	32
Epochs	200

5.2.2 Results

This subsection presents the results obtained during the training of the GNN for each iteration. The policy and value losses across all three iterations are depicted in Figure 5.5, offering a comparative visualization on a fixed scale.

5.2.2.1 Iteration 1

The first iteration utilized 26,928 turns of data from three small maps (Owl Island, Banana, and Custom Map). This dataset, generated via self-play by the baseline agent, is summarized in Table 5.3. The loss curves (Figure 5.2) exhibit smooth convergence, with value loss stabilizing faster than policy loss. Table 5.4 highlights evidence of learning during evaluation.

The trained agent achieved 75.6% of match points against the untrained model and demonstrated competitive performance against the baseline despite using half as many iterations. The uneven data distribution may explain the agent's relatively lower performance on Owl Island, which had significantly fewer data points in the dataset.

Table 5.3: Data Distribution Across Maps For Iteration 1

Map	Number of Turns
Custom Simple	11,980
Banana	8,744
Owl Island	6,204
Total	26,928

Table 5.4: Evaluation Results for Training Iteration 1.

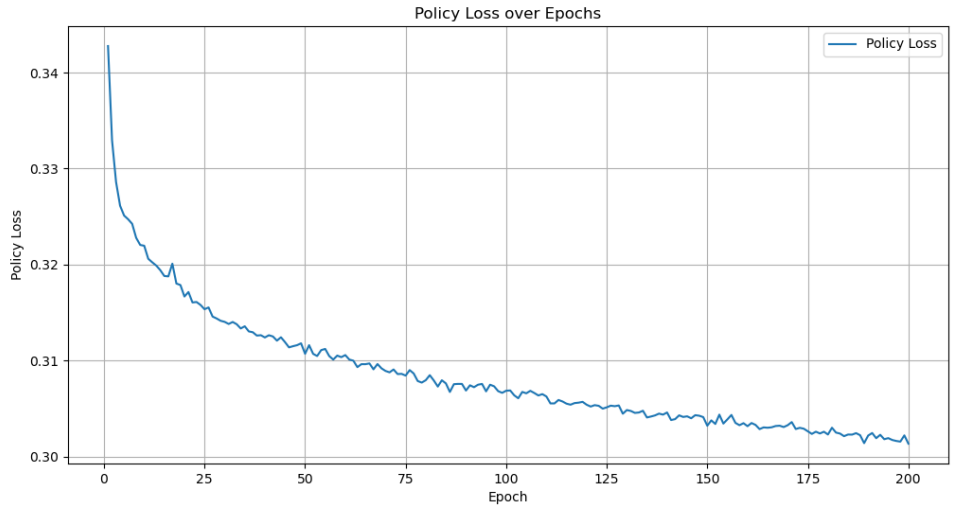
Map	Score vs. Untrained Model	Score vs. Baseline
Custom Simple	63.3%	51.7%
Banana	86.7%	56.7%
Owl Island	76.7%	46.7%
Overall	75.6%	51.7%

5.2.2.2 Iteration 2

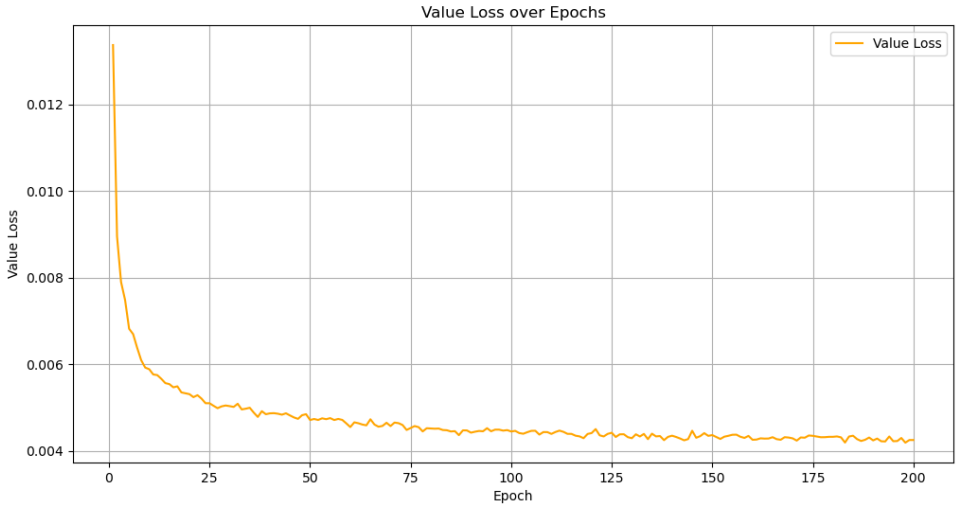
Iteration 2 improved data balance, using 49,206 turns, obtained by expanding the dataset with experiences of the updated agent (described in Table 5.5). Losses in Figure 5.3 show increased variance, particularly for value loss. Figure 5.5 gives valuable insight for comparison with the previous iteration, showing that in reality, both curves vary in a lower range than in the first iteration. This suggests improved capacity for data approximation as more turns of similar quality were added.

Evaluation results (Table 5.6) reflect stronger performance overall. Although the agent underperformed against the previous iteration on Owl Island, its improved results against the baseline on the same map suggest enhanced map comprehension due to the balanced dataset.

Figure 5.2: Policy and Value Losses For Training Iteration 1.



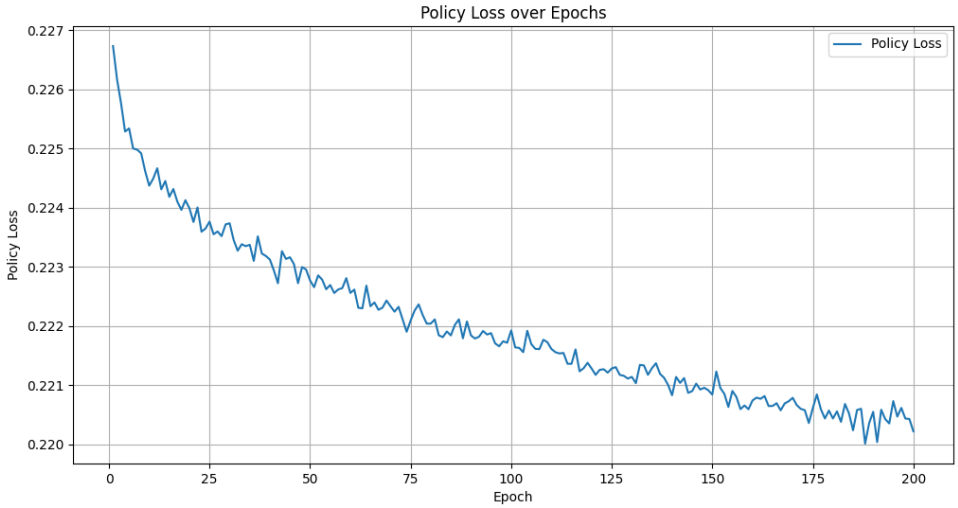
(a) Policy Loss (Iteration 1)



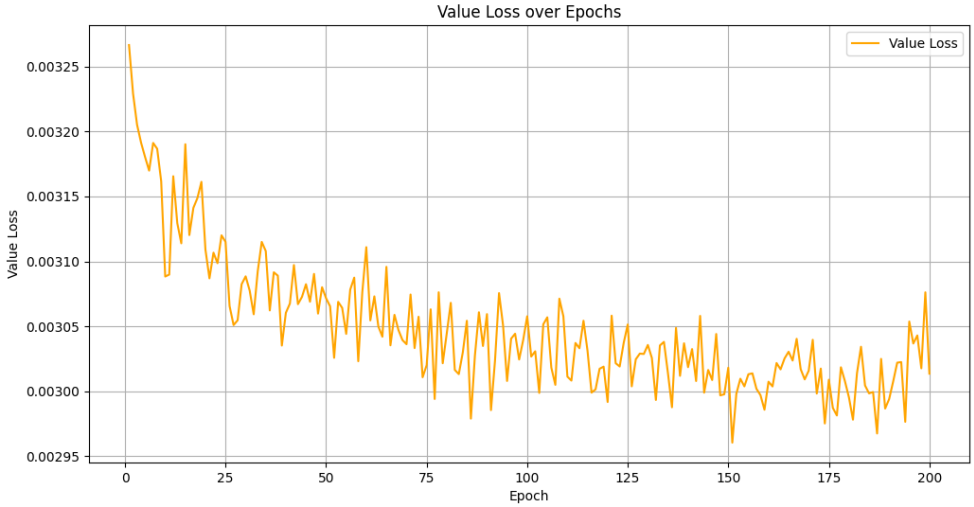
(b) Value Loss (Iteration 1)

Source: The Author.

Figure 5.3: Policy and Value Losses For Training Iteration 2.



(a) Policy Loss (Iteration 2)



(b) Value Loss (Iteration 2)

Source: The Author.

Table 5.5: Data Distribution for each map in Second Iteration training

Map	Number of Turns
Custom Simple	14,486
Banana	18,325
Owl Island	16,395
Total	49,206

Table 5.6: Evaluation Results for Training Iteration 2

Map	Score vs. Iteration 1 Model	Score vs. Baseline
Custom Map	58.3%	63.3%
Banana	63.3%	53.3%
Owl Island	40%	63.3%
Overall	53.9%	60%

5.2.2.3 Iteration 3

Iteration 3 introduced the Italy map, resulting in a dataset of 89,254 turns (Table 5.7). The loss curves (Figure 5.4) are smoother than those of the previous iteration but indicate increased overall loss values compared to earlier iterations. The value loss curve, in particular, appears less stable.

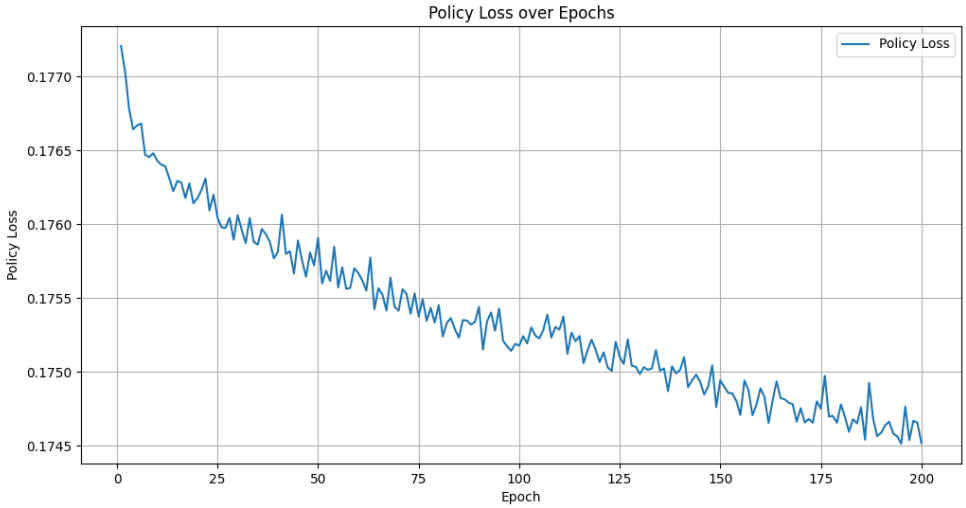
Evaluation results (Table 5.8) show a decline in general performance. The agent demonstrated some learning on the Italy map, achieving a high score against the previous iteration agent. However, it failed to outperform the baseline on this more challenging map.

The reduced dataset size for Italy (10,002 turns) compared to other maps likely contributed to the diminished performance. This aligns with the principles of Curriculum Learning (SOVIANY et al., 2022), which suggest that more complex tasks require greater quantities of high-quality data. Consequently, the model from Iteration 2 struggled to generalize its learning from smaller maps to generate data sufficient to outperform the baseline after just one iteration of training.

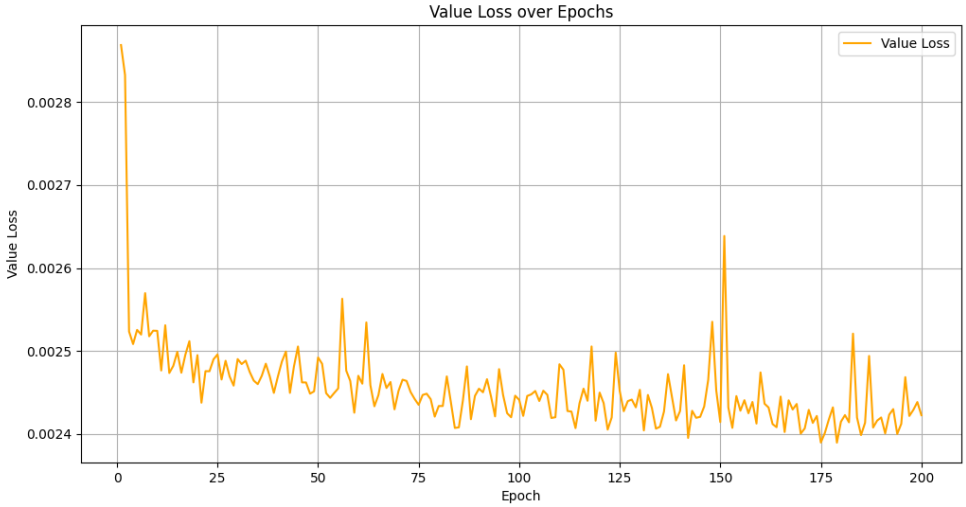
Table 5.7: Data Distribution for each map in Third Iteration training

Map	Number of Turns
Custom Simple	24,486
Banana	28,335
Owl Island	26,431
Italy	10,002
Total	89,254

Figure 5.4: Policy and Value Losses For Training Iteration 3.



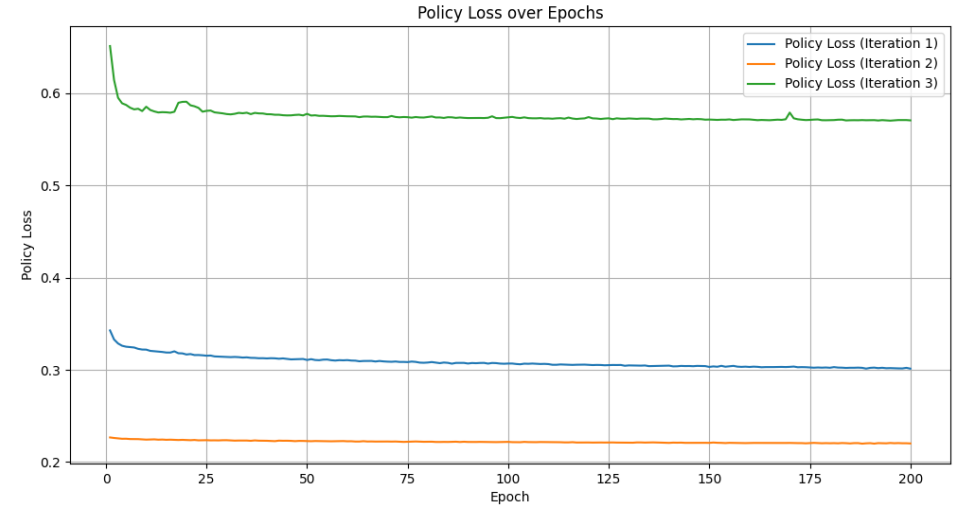
(a) Policy Loss (Iteration 3)



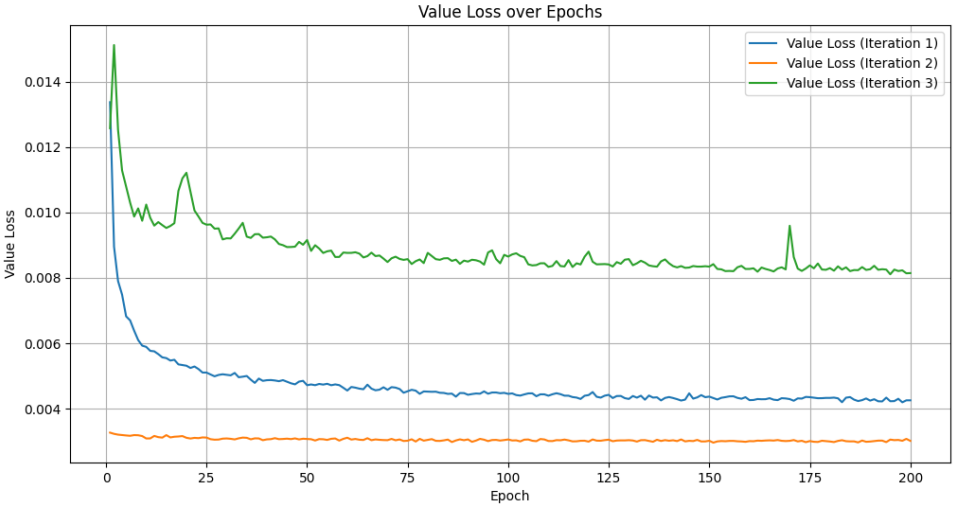
(b) Value Loss (Iteration 3)

Source: The Author.

Figure 5.5: Policy and Value Losses For All Training Iterations



(a) Policy Loss (All Iterations)



(b) Value Loss (All Iterations)

Source: The Author.

Table 5.8: Evaluation Results for Training Iteration 3.

Map	Score vs. Iteration 2 Model	Score vs. Baseline
Custom Map	70%	51.7%
Banana	43.3%	50%
Owl Island	46.7%	46.7%
Italy	90%	36.7%
Overall	62.5%	46.2%

5.3 Tournament

This section evaluates the performance of different agents through two tournaments. The first tournament uses the best-performing GNN, from the second training iteration, while the second tournament utilizes the GNN obtained in the third iteration. Subsection 5.3.1 outlines the setup and methodology, and Subsection 5.3.2 discusses the results, including insights derived from agent performance across different maps.

5.3.1 Setup

The tournaments compared the following agents:

- **Vanilla MCTS**: Baseline agent without GNN guidance.
- **GNN-Guided MCTS**: A standard MCTS agent augmented with GNN policy and value predictions, as in (SILVER et al., 2017).
- **GA Agent**: Standard genetic algorithm, similar to GG-Net (BAUER, 2024). The policy output from the GNN is ignored.
- **GA with Policy Initialization**: GA agent with populations initialized using the GNN's policy predictions.

In each tournament, every pair of agents played 50 games on each map. All agents that use a neural network share the same GNN model. In order to ensure fairness in comparing different agents controlled by different techniques, a time control of 12 seconds per move was enforced. Tables 5.9 and 5.10 contain, respectively, the parameters configuration for the MCTS and GA agents in the tournament.

Since the GA Agent and the GA with Policy Initialization share similar configurations and operations—differing only in the additional policy initialization step—the evolution process was designed to allow interruptions between steps in its lifecycle (as described in Section 4.2.3). This ensured no additional computational overhead would

unfairly benefit the GA with Policy Initialization.

Table 5.9: MCTS Parameters for Tournaments. For this configuration, the MCTS will perform as many iterations as possible under the time constraint.

	Baseline	GNN Guided
Iterations	∞	∞
Max Rollout Depth	20	20
Policy Trust	N/A	0.75
Time Limit	12s	12s

Table 5.10: GA Parameters for Tournaments. For this configuration, the GA will perform as many generations as possible under the time constraint.

	Standard GA	GA with Policy Initialization
Population Size	30	30
Elites Size	5	5
Generations	∞	∞
Sampled Individuals	N/A	90
Time Limit	12s	12s

5.3.2 Results

While the GA agents did not surpass the performance of the MCTS-based ones, consistent with prior findings (BAUER, 2024), the GA with policy initialization significantly outperformed the standard GA in our experiments. In both tournaments, the guided MCTS achieved first place (scoring 54.4%, and 55,6%), closely followed by the GA with policy initialization (scoring 53.8%, and 54.2%). While agents performance were generally close, the margin between second and third places was more significant (7.1% in the first tournament and 7.6%, in the second), suggesting that the GNN guided MCTS and the GA with policy performed significantly better than the other two competitors.

Although the overall score (Tables 5.11, 5.13) indicate tight competition, the matchup results (Tables 5.12, 5.14) reveal consistent performances. The guided MCTS scored at least 51% of the match points in all pairings. The GA with policy initialization outperformed the standard GA and the vanilla MCTS in both tournaments, having a particular wide margin over the standard GA.

Out of the 1050 games, the GA with policy scored 9.05% more points than the standard GA, achieving 60.6% in the head-to-head machups. In comparison, the Guided MCTS scored 1.05% more points than the GA with Policy Initialization, winning 53.29% in their direct encounters.

5.3.2.1 Second Iteration GNN Tournament

Table 5.11 summarizes the results for the tournament using the second iteration GNN model. The GNN-Guided MCTS and GA with policy initialization consistently outperformed other contestants.

Table 5.11: Tournament results (Second Iteration GNN) as scoring percentages. Each agent played 150 games on each of the four maps. Overall scores are shown as percentages of the maximum possible score (450 points).

Agent	Custom Map	Owl Island	Banana	**Overall**
Guided MCTS	56%	57.33%	50%	54.44%
GA with Policy Init.	48.33%	51.33%	62%	53.82%
GA Agent	51%	40.67%	48.67%	46.78%
Vanilla MCTS	44.67%	50.66%	39.33%	44.89%

Table 5.12: Matchup results for the tournament (Second Iteration GNN). Each matchup consisted of 150 games. Values represent the number of wins for the row agent (Agent 1) against the column agent (Agent 2).

Agent 1 \ Agent 2	Guided MCTS	GA with Policy Init.	GA Agent	Vanilla MCTS
Guided MCTS	–	51%	54.33%	58%
GA with Policy Init.	49%	–	57.33%	55.33%
GA Agent	45.67%	42.67%	–	52%
Vanilla MCTS	42%	44.67%	48%	–

5.3.2.2 Third Iteration GNN Tournament

Results for the tournament using the third GNN model are presented in Table 5.13. The GNN-Guided MCTS and GA with policy initialization maintained strong performance, but all network-guided models struggled on the Italy map. This reflects the limitations of this model on learning this more complex map in just one iteration of learning, as discussed in Section 5.2, third training iteration (5.2.2.3). Table 5.14 shows that the Guided MCTS and GA with Policy Initialization agents won most matches against the Vanilla MCTS.

Table 5.13: Tournament results (Third Iteration GNN) as scoring percentages. Each agent played 150 games on each of the four maps. Overall scores are shown as percentages of the maximum possible score (600 points).

Agent	Custom Map	Owl Island	Banana	Italy	**Overall**
Guided MCTS	62.33%	56.67%	57.33%	46%	55.58%
GA with Policy Init.	48%	59.33%	61.33%	48%	54.17%
Vanilla MCTS	41%	41.33%	42%	62%	46.58%
GA Agent	48.67%	42.67%	39.33%	44%	43.67%

Table 5.14: Matchup results for the tournament (Third Iteration GNN). Each matchup consisted of 200 games. Values represent the scoring percentages of the row agent (Agent 1) against the column agent (Agent 2).

Agent 1 \ Agent 2	Guided MCTS	GA with Policy Init.	Vanilla MCTS	GA Agent
Guided MCTS	–	55%	57.25%	54.5%
GA with Policy Init.	45%	–	54.5%	63%
Vanilla MCTS	42.75%	45.5%	–	51.5%
GA Agent	45.5%	37%	48.5%	–

6 CONCLUSION

6.1 Overview

This research explored a simple increment over a novel approach in Risk-like games. Our initial experiments revealed that starting the evolution with high-quality individuals can make difference in GA-based approaches. Still, this work contains many limitations;

Our experiments only included a small subset of simple maps. Different techniques may have their strengths and flaws. Having more diverse experiments and data help in understanding in which scenarios each algorithms perform better, a relevant analysis to compare different agents.

The GNN learning was only partially successful. While the network-guided agents surpassed the baseline performance, our netowrk struggled to learn a new and more complex map. Ideally, the iterations of the reinforcement learning training are run until convergence. The exact implications on training the GNN over GA self-play data on convergence speed and level of play after convergence are yet to be explored.

Parameters used for both GA and MCTS agents were not sufficiently explored. Fine-tunning parameters with grid search, and trying different designs for the GA (without total crossover, for instance) is essential to achieve the highest level of play.

Finally, a great part of our limitations (as dataset size, number of RL iterations, number of games in evaluation) are associated with the performance of our implementation.

6.2 Future Work

There is a lack of direct comparison between the different techniques proposed over the last two decades. Future research could focus on reviewing and testing more extensively the most notorious works, to understand more clearly the current state-of-the-art. This issue can be associated with the fact that the research in the field is scattered on different platforms and variations of the game, with no platform being consolidated as the best choice to develop AI agents. Despite the code of the current work being open, and allowing fully local play, refactors would be needed for it to be more extensible and easily maintainable. Additionally, critical parts could be implemented in a compiled lan-

guage to achieve efficiency in CPU operations on game simulation. Thus, the advent of a free-to-use, efficient, extensible, comprehensible, and customizable implementation of the game, that supports its most traditional variations, with no server communication required, would be a milestone on experimentation and reproducibility. Another beneficial asset would be large public datasets of games played by human experts.

Many different possibilities can be explored in terms of new agents proposal, or refinements to existing techniques. On the GNN side, different architectures can be further explored and compared to provide a hint on what is more effective in RISK. For policy gradient optimization, Proximal Policy Optimization (SCHULMAN et al., 2017) is a cutting-edge technique that can be explored.

A myriad of variations of Evolutionary Algorithms can be tested. Ranging from applying CMA-ES on the evolution, to introducing more complex methods, such as implementing RHEA (GAINA et al., 2022) framework, evaluating the actions of a fixed horizon of turns, instead of just a turn. While it could add depth to the GA moves search, a reformulation of the entities (genes, individual, and populations) of the evolutionary domain would be required, posing the challenge of correcting crossover and mutation operations on a broader scale, since a mutation of a singular order of a turn can, for example, change the ownership of a territory, thus impacting the whole upcoming sequence of turns, a consistency that should be maintained for both populations in case coevolution is maintained.

Different combinations of already documented approaches are possible. In the present work, policy predictions are used only to initialize populations, but other manners to incorporate it to the framework can be found. For example, a low-depth guided MCTS could be combined with the GA, in a hybrid framework that combines SOTA approaches. The MCTS could be used to search potential follow-ups of the moves to be evaluated on the GA, or the GA could act to refine the MCTS output.

The potential of evolutionary algorithms on neural networks training *tabula rasa* in detriment of the established MCTS, as proposed in AlphaZero (SILVER et al., 2017), is yet to be investigated. In guided MCTS, even when the neural network has random parameters, the search should eventually reach a terminal state of the game, backpropagating the results, if given enough time. In comparison to the MCTS, the GA is unable to foresee reasonable continuations of the game without relying on the GNN, as the fitness of individuals is completely given by the GNN, and no additional search is performed. Presumably, the GA as studied in this work would struggle in the first few iterations of a

reinforcement learning training. Still, if this technique outperforms MCTS with a trained GNN, it is expected that the GA can be used as a network improvement operator from a certain point. In that case, this relation could be further studied to achieve the optimal efficiency reinforcement learning framework for the game, in both convergence time, and play level reached criteria.

Other unexplored techniques, such as Hierarchical Learning, are yet to be implemented for RISK-like games.

REFERENCES

BAUER, A. Artificial intelligence with graph neural networks applied to a risk-like board game. **IEEE Transactions on Games**, v. 16, n. 2, p. 342–351, 2024.

Bauer, Andrew. **py-risk**. 2023. Accessed: Dec 2024. Available from Internet: <https://github.com/andenrx/py-risk>.

BLICKLE, T.; THIELE, L. A comparison of selection schemes used in genetic algorithms. In: . [s.n.], 1995. Available from Internet: <https://api.semanticscholar.org/CorpusID:16240839>.

BROWN, N.; SANDHOLM, T.; MACHINE, S. Libratus: The superhuman ai for no-limit poker. In: **IJCAI**. [S.l.: s.n.], 2017. p. 5226–5228.

BROWNE, C. B. et al. A survey of monte carlo tree search methods. **IEEE Transactions on Computational Intelligence and AI in Games**, v. 4, n. 1, p. 1–43, 2012.

CARR, J. Using graph convolutional networks and td(λ) to play the game of risk. **CoRR**, abs/2009.06355, 2020. Available from Internet: <https://arxiv.org/abs/2009.06355>.

DODGE, Y. Mean squared error. In: ____. **The Concise Encyclopedia of Statistics**. New York, NY: Springer New York, 2008. p. 337–339. ISBN 978-0-387-32833-1. Available from Internet: <https://doi.org/10.1007/978-0-387-32833-1_251>.

FORD, L. R.; FULKERSON, D. R. Maximal flow through a network. **Canadian Journal of Mathematics**, v. 8, p. 399–404, 1956.

GAD, A. F. Pygad: An intuitive genetic algorithm python library. **CoRR**, abs/2106.06158, 2021. Available from Internet: <https://arxiv.org/abs/2106.06158>.

GAINA, R. D. et al. Rolling horizon evolutionary algorithms for general video game playing. **IEEE Transactions on Games**, v. 14, n. 2, p. 232–242, 2022.

GIBSON, R.; DESAI, N.; ZHAO, R. An automated technique for drafting territories in the board game risk. In: . [S.l.: s.n.], 2010.

GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization, and Machine Learning**. New York: Addison-Wesley, 1989.

GOOD, I. J. Rational decisions. **Journal of the Royal Statistical Society. Series B (Methodological)**, [Royal Statistical Society, Oxford University Press], v. 14, n. 1, p. 107–114, 1952. ISSN 00359246. Available from Internet: <http://www.jstor.org/stable/2984087>.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>.

HEREDIA, L.; CAZENAVE, T. Expert iteration for risk. In: ____. [S.l.: s.n.], 2022. p. 27–37. ISBN 978-3-031-11487-8.

HOLLAND, J. H. Genetic algorithms. **Scientific American**, Scientific American, a division of Nature America, Inc., v. 267, n. 1, p. 66–73, 1992. ISSN 00368733, 19467087. Available from Internet: <http://www.jstor.org/stable/24939139>.

JOHANSSON, S. J.; OLSSON, F. Mars – a multi-agent system playing risk. In: . [s.n.], 2005. Available from Internet: <https://api.semanticscholar.org/CorpusID:17010299>.

KINGMA, D. P.; BA, J. **Adam: A Method for Stochastic Optimization**. 2017. Available from Internet: <https://arxiv.org/abs/1412.6980>.

KIPF, T. N.; WELLING, M. **Semi-Supervised Classification with Graph Convolutional Networks**. 2017. Available from Internet: <https://arxiv.org/abs/1609.02907>.

KOCSIS, L.; SZEPESVáRI, C. Bandit based monte-carlo planning. In: . [S.l.: s.n.], 2006. v. 2006, p. 282–293. ISBN 978-3-540-45375-8.

LIU, J.; PéREZ-LIéBANA, D.; LUCAS, S. M. Rolling horizon coevolutionary planning for two-player video games. In: **2016 8th Computer Science and Electronic Engineering (CEEC)**. [S.l.: s.n.], 2016. p. 174–179.

SCARSELLI, F. et al. The graph neural network model. **IEEE Transactions on Neural Networks**, v. 20, p. 61–80, 2009. Available from Internet: <https://api.semanticscholar.org/CorpusID:206756462>.

SCHULMAN, J. et al. **Proximal Policy Optimization Algorithms**. 2017. Available from Internet: <https://arxiv.org/abs/1707.06347>.

SHI, Y. et al. **Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification**. 2021. Available from Internet: <https://arxiv.org/abs/2009.03509>.

SHI, Y. et al. Masked label prediction: Unified message passing model for semi-supervised classification. In: ZHOU, Z.-H. (Ed.). **Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21**. International Joint Conferences on Artificial Intelligence Organization, 2021. p. 1548–1554. Main Track. Available from Internet: <https://doi.org/10.24963/ijcai.2021/214>.

SILVER, D. et al. Mastering the game of go without human knowledge. **Nature**, v. 550, p. 354–359, 10 2017.

Soares, Ricco. **py-risk**. 2024. Accessed: Dec 20204. Available from Internet: <{https://github.com/RiccoSoares/py-ris}.>

SOVIANY, P. et al. **Curriculum Learning: A Survey**. 2022. Available from Internet: <https://arxiv.org/abs/2101.10382>.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. Second. The MIT Press, 2018. Available from Internet: <http://incompleteideas.net/book/the-book-2nd.html>.

VELIčKOVIć, P. et al. **Graph Attention Networks**. 2018. Available from Internet: <https://arxiv.org/abs/1710.10903>.

WORDEN, K.; STASZEWSKI, W. J.; HENSMAN, J. J. Natural computing for mechanical systems research: A tutorial overview. **Mechanical Systems and Signal Processing**, v. 25, n. 1, p. 4–111, 2011. ISSN 0888-3270. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0888327010002499>.